EFOP-3.4.3-16-2016-00014

PÉLDATÁR

Webes fejlesztés

Készítette: Rusznák Attila
SZTE JGYPK
Informatika Alkalmazásai Tanszék

Lektorálta: Tóth Attila

Jelen tananyag a Szegedi Tudományegyetemen készült az Európai Unió támogatásával. Projekt azonosító: EFOP-3.4.3-16-2016-00014.

Alprojekt azonosító: AP2 – Komplex képzés- és szolgáltatásfejlesztés

Altéma azonosító: AP2_JGYPK5 Magyar és idegen nyelvű képzések oktatási innovációja az MTMI területen és tanártovábbképzés

Szegedi Tudományeg
Cím: 6720 Szeged, Du
www.u-szeged.hu
www.szechenyi2020.

Tartalomjegyzék

1. *Bevezető 1*
2. *Nyelvi alapok..... 6*
3. *Összetett tömbös feladatok 19*
4. *Objektumorientált programozás 24*
5. *A jQuery keretrendszer..... 36*
6. *Összetett jQuery feladatok..... 46*
7. *Az AJAX működése a jQuery-ben 52*

1. Bevezető

Ez a példatár a Kliens oldali webes programozás című tantárgy szemináriumának tananyagát hivatott elmélyíteni és kibővíteni. A tantárgy előfeltétele, hogy a hallgató tisztában legyen a programozás alapjaival (változók, vezérlési szerkezetek, logikai feltételek, alapvető adatszerkezetek), így ezen alapokat ismertnek tekintem a magyarázatok és a feladatok bemutatása során.

Ahhoz, hogy valaki jó fejlesztővé válhasson, fontos hogy már a tanulmányai alatt is rengeteget gyakoroljon. A programozás elsajátítása egy folyamat. A folyamat sikeréért viszont rengeteg időre, gyakorlásra és kitartásra van szükségünk. Programozni nem lehet megtanulni egyik hétről a másikra, ahogyan egy idegennyelvet sem lehet így elsajátítani, vagy megizmosodni az edzőteremben. Ahhoz, hogy valaki sikeres programozóvá válhasson, minél több példával és algoritmussal kell hogy szembesüljön. Eleinte nehéznek érezzük, mely fokozatosan válik egyre inkább könnyebbé és érthetővé. Eleinte mindenre problémaként tekintünk, mely problémának nem ismerjük az algoritmusát. Nem tudjuk, hogy van-e a problémának megoldása, ha igen, akkor hogy hány lépés vezet a megoldásra, melyik a legrövidebb út és mennyi ideig fog tartani annak a megtalálása. Idővel, amikor már elegendő tapasztalatot szerzünk, a probléma feladattá szelídül. Azaz, ha már tudjuk, hogy mit milyen algoritmus segítségével tudunk megoldani és pillanatok alatt képesek vagyunk azt előhívni vagy előkeresni a megfelelő helyről, a probléma csupán egy feladat számunkra, hisz tudjuk a megoldást, illetve a megoldáshoz vezető utat is. A cél az, hogy minél több problémára feladatként tudjunk tekinteni, mert ezáltal gyorsabban és hatékonyabban tudunk megoldani programozási feladatokat.

Azok számára, akik ezelőtt csak erősen típusos nyelvvel foglalkoztak (pl.: Java, C#), valószínűleg új lesz a script-ek világa, amilyen a Javascript is. A script nyelvek többnyire gyengén típusosak, mely azt a szabadságot adja nekünk, hogy kevésbé kell odafigyelnünk a típusokkal járó kötöttségekre. Egy változó bármilyen típust tartalmazhat, függetlenül attól, hogy néhány másodperce mit tároltunk el benne, ahogy egy tömbön belül is elhelyezhetünk string, boolean vagy épp integer értékeket egymás szomszédságában. A gyengén típusosságnak azonban az az ára, hogy könnyelművé válhatunk a munkánk során. Ez azt jelenti, hogy megszokjuk a kevésbé kötött szabályokat, így idővel visszatérve a egy erősen típusos nyelvhez, például a Java-hoz, könnyen szembesülhetünk azzal, hogy folyton szintaktikai hibákat vétünk. Éppen ezért a példák mindegyikében betartom az alapvető szabályokat még akkor is, ha a nyelv azok megszegése esetén is megengedő lenne. Ilyen például, hogy a Javascript-ben nem kötelező az egyes utasítások végére pontosvesszőt írni, vagy nem kötelező egy változót deklarálni. Természetesen ezekkel a lehetőségekkel bárki élhet, de a Java-ban megismert kötöttségek miatt kialakult megszokások megtartása érdekében ettől mindenkit óvva intenék.

A *Nyelvi alapok* fejezetben olyan feladatok vannak, amelyek végig vezetik a hallgatót az alapvető vezérlő- és adatszerkezetek használatán (elágazások, ciklusok, tömbök), illetve kitérünk a nyelvspecifikus beépített függvényekre, egyszerű objektumok készítésére. A feladatok megoldásához nincs szükség objektumorientált szemléletmódra, sőt még csak metódusszervező képességre sem. Ez a bevezetőfeladat azok számára is nagy segítséget jelent, akik a Javascript által szeretnék elsajátítani a problémamegoldó és algoritmikus gondolkodás alapjait.

Az *Összetett tömbös feladatok* fejezet az előző fejezetben lévő analízis jellegű gondolatműveletek mellett megjelenik a szintézis is, ahol a hallgatónak már nem egy-egy konkrét részfeladatra kell koncentrálnia, hanem szükséges, hogy holisztikusan átlássa a teljes feladat működését, hiszen az egyes részfeladatok egymásra épülnek.

A harmadik témakörünk az *Objektumorientált programozás*. A nyelv csak az elmúlt pár évben kezdte erősebben támogatni az objektumorientált szemléletmódot. Ez leginkább az osztályszervezés megjelenésében jelent újdonságot. Mostantól a prototípusokon (konstruktor alapú tervezés) és az objektumliterálokon túl megjelenik az osztályszervezés lehetősége is. Ebben a fejezetben a hallgató megismerkedhet az egyszerűbb osztályok felépítésével, gyakorolhatja a betokozás és az öröklés működését és használatát. De olyan témakörök is megjelennek a példákban, mint a statikus tagok működése. A feladatok feltételezik, hogy a hallgató már ismeri a prototípus alapú működést, így a fejezet első részében ilyen jellegű, összetettebb feladatokat kell megoldania. Ezt követően ellenőrizheti a megszerzett tudását az osztályszervezésben is, ahol az egyszerűbbtől a bonyolultabb felé haladás elvét követjük. Természetesen minden feladathoz megtalálható annak a megoldása, így a hallgatók könnyedén ellenőrizhetik a munkájukat. Fontos azonban megjegyezni, hogy a programozás nem egzakt tudomány, ahogy a magyar irodalom sem. Egy levelet és egy programot is megírhatunk sokféleképpen, nincs rá egységes recept.

A következő fejezetben a hallgató megismerkedik a *jQuery* keretrendszer legújabb verziójával. Tekintve, hogy a natív nyelvben bonyolultabb, hosszabb kódokkal és refaktorálással valósítható meg egy összetettebb program a DOM (Dokumentum Objektummodell) témakörében, így a nyelv ezen részét ebben a fejezetben ismertetjük. A DOM segítségével a hallgató a DOM fában (HTML elemek) történő lépkedést valósíthatja meg. Képessé válik egy HTML elem létrehozására, a HTML dokumentum tetszőleges helyére történő beszúrására, az elemek egyedi vagy csoportos kikeresésére, módosítására, manipulálására. A Javascript dinamizmusa valójában itt nyilvánul meg. Ma már elvárt, hogy a hallgató ismerjen minden általa használt nyelvhez legalább egy keretrendszert, melyek gyakran leegyszerűsítik a nyelvet részben vagy egészben, akár szintaktikai változtatások árán. Természetesen itt is szerepelnek ZH szintű összetett gyakorlófeladatok.

Az utolsó fejezetben *Az AJAX működése a jQuery-ben* megismerkedünk az AJAX működésével a jQuery szemüvegén keresztül. Megnézzük hogyan lehet egy weboldalon aszinkron hívásokat lefolytatni a szerverrel a háttérben. Adatokat küldünk és fogadunk. A visszakapott információt elemezzük, illetve elhelyezzük a DOM fában, sőt még az olyan finomságokra is kitérünk, mint a hibakezelés, a várakozás vizuális jelzése a felhasználó felé, az AJAX alapú űrlapbeküldés vagy a JSON, mint fájlformátum működése.

A példatár legfőképpen a NAT-ban meghatározott 9 kulcskompetenciák közül a digitális kompetenciát fejleszti. Olyan komponensekre koncentrálva, mint: az algoritmikus gondolkodás, az adatmodellezés, a valós világ modellezése, a problémamegoldás, az alkalmazói képesség, és a rendszerszintű gondolkodás.

Tudás	Képesség	Attitűd	Autonómia
Ismeri az alapvető vezérlési szerkezeteket és adatszerkezeteket (elágazások, ciklusok, tömbök).	Képes egyszerű és összetett feladatokban alkalmazni a vezérlőszerkezeteket, elágazásokat és adatszerkezeteket.	Törekszik a szerkezetek pontos megvalósítására.	Képes a tömböket, ciklusokat és elágazásokat önállóan használni.
Ismeri az alapvető beépített függvényeket, azok felépítését és működését.	Képes alkalmazni feladatoktól függően a megfelelő metódust.	A metódusok hívását, az azok által visszaadott értékeket pontosan használja.	Betartja a paraméterátadás és visszatérés szabályait.
Tisztában van a függvényírás szabályaival, illetve a kapcsolódó fogalmakkal (hatókör, lokális változók, visszatérési érték).	Saját függvényt hoz létre.	A függvény létrehozásának szabályait alkalmazza és betartja.	A saját függvényét képes bármilyen környezetben alkalmazni, azt továbbfejleszteni.
Ismeri az objektumliterál, a prototípus és az osztály fogalmát, felépítését és deklarációit.	Prototípusokat, objektumliterálokat és osztályokat hoz létre, példányosít.	Fogékony az objektumorientált programozással kapcsolatos módszerekre. Nyitott az objektumok különböző célú alkalmazására.	Az előforduló hibákat képes önállóan megtalálni és azokat kijavítani.
Ismeri az alapértelmezés szerinti kezdeti értéket, a this kulcsszó, a konstruktor, valamint a betokozás fogalmát.	Képes objektumokat deklarálni és inicializálni, konstruktorokat, getter és setter metódusokat létrehozni és használni.	Törekszik az objektumok pontos használatára.	Önállóan végez példányosítást, elemi objektumok közötti műveleteket.
Ismeri az öröklés működését és az öröklésben részt vevő osztályok kapcsolatát.	Érti és alkalmazza az osztályok közötti öröklést, a belső folyamatok működését.	Szem előtt tartja, hogy milyen esetekben alkalmazhat öröklést.	Önállóan képes öröklődő osztályok létrehozására, azok példányosítására.
Tisztában van a DOM fogalmával, ismeri az alapvető fabejárási műveleteket.	Megérti és használja a DOM-hoz kapcsolódó beépített metódusokat.	Törekszik a DOM bejárófüggvényeinek pontos alkalmazására.	Önállóan képes natív módon vagy keretrendszer segítségével a HTML elemek dinamikus, kiválasztására, módosítására.
Ismeri az AJAX fogalmát és működését. Különbséget tesz szinkron és aszinkron hívások között.	Képes AJAX segítségével alapvető szerverkommunikációs műveletekre a háttérben.	Az AJAX hívás szabályait és műveleteit betartja.	Saját AJAX hívásokat képes készíteni, azokat módosítani, a felmerülő hibákat javítani.

Ezúton kívánok a feladatok megoldásához sok sikert, kitartást és bízom benne, hogy sokan fogják megszeretni javascript-et, ami 2020-ban jelenleg a 3. helyen álló legnépszerűbb programozási nyelv.

1.1. A Javascript kód elhelyezésének lehetőségei

A példák során böngészős alkalmazásokat fogunk készíteni, melyeknél különálló JS fájlokban fogunk dolgozni. A haladóbb szintű feladatoknál már a DOM (dokumentum objektummodell) fában is fogunk manipulálni, értékeket megjeleníteni. Éppen ezért az ezekhez kapcsolódó HTML és CSS dokumentumok a megoldásokkal együtt letölthetőek az alábbi github címről:

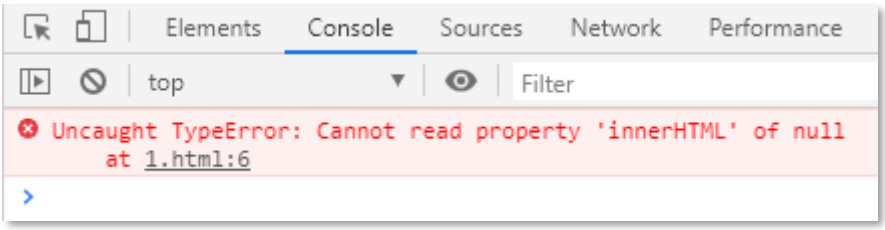
https://github.com/arusznak/javascript

A feladatmegoldások előtt ismétlésképpen nézzük át, milyen lehetőségeink vannak a script-ek elhelyezésére egy HTML dokumentumon belül beágyazva vagy külső fájlként.

Alapvetően elhelyezhető a script HTML elem a dokumentum head címkék között és a body címkék között is. Ha a head címkék között helyezzük el a kódunkat beágyazott formában vagy külső fájlként, a böngésző betöltődésekor a javascript kód előbb kerül végrehajtásra, mint a body elemben lévő további HTML kódok. Ez abban az esetben nem jelent gondot, ha a javascript kódban nem szeretnénk semmiféle manipulálást végezni a DOM-ban, azaz a HTML fában. Ha viszont szeretnénk valamilyen értéket beolvasni, kiíratni a DOM-ba, akkor a javascript ezeket a HTML elemeket nem fogja látni, mivel a programkód hamarabb kerül értelmezésre, mint az utána következő HTML kód. Futassuk le az alábbi kódot a böngészőben:

```
<!doctype html>
<html>
<head>
<script>
    let elem = document.getElementById("elem");
    alert(elem.innerHTML);
</script>
</head>
<body>
    <div id="elem">A szöveg kiolvasása a DIV-ből.</div>
</body>
</html>
```

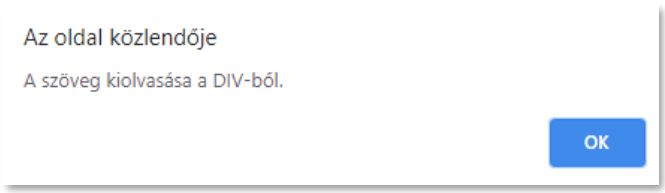
Ha megnyitjuk a böngészőben a konzolt az F12-es billentyűvel a következő hibaüzenetet fogjuk látni:



Láthatjuk, hogy hiába hivatkozunk helyesen a DOM-ban lévő DIV elemre, a script nem látja, mivel a programkód előbb kerül végrehajtásra, mint maga a HTML dokumentum body része. Arra, hogy ezt kiküszöböljük kétféle megoldás kínálkozik. Az egyik az, hogy magát a programkódot elhelyezzük egy eseményfigyelőben, mégpedig az onload-ban. Az onload azt csinálja, hogy megvárja, amíg a teljes HTML dokumentum betöltődik, majd csak ezt követően kezdi el értelmezni a javascript kódot, így a programkód már látni fogja a DOM-ban lévő HTML elemeket:

```
<script>
    window.addEventListener("load", function(){
        let elem = document.getElementById("elem");
        alert(elem.innerHTML);
    });
</script>
```

Ha ezúttal frissítjük a böngészőben az oldalt, akkor megjelenik a felugró ablak:



Ennek egy másik módja az, ha a script HTML elemet a body elem végén helyezzük el, ekkor ugyanis legutoljára kerül majd betöltésre a program.

```
<body>
    <div id="elem">A szöveg kiolvasása a DIV-ből.</div>
    <script>
        let elem = document.getElementById("elem");
        alert(elem.innerHTML);
    </script>
</body>
</html>
```

Ebben az esetben természetesen nincs szükségünk az eseményfigyelőre. Végül nézzük meg mi helyzet akkor, ha a kódunkat külső fájlban szeretnénk elhelyezni, ahogyan ezt a későbbi példák mindegyikében látni fogjuk.

Szintén megvan a lehetőségünk, hogy eseményfigyelőbe tegyük a kódunkat a külső fájlon belül. Ekkor teljesen mindegy, hogy a HTML dokumentumban hol helyezzük el a script HTML elemet. Ám arra is van lehetőségünk, hogy a head elemben helyezzük el úgy, hogy a külső fájlban ne kelljen az imént bemutatott onload eseményfigyelőt használnunk. Ehhez mindössze a script elemhez vegyük fel a defer nevű attribútumot:

```
<head>
    <script defer src="js/1b.js"></script>
</head>
```

A külső fájlban pedig csak a két sornyi kód található meg az eseményfigyelő nélkül:

```
    let elem = document.getElementById("elem");
    alert(elem.innerHTML);
```

A defer csak külső fájlok esetén használható és a segítségével megadhatjuk, hogy csak azután töltse be a külső script fájlt a böngésző, miután a teljes dokumentumot betöltötte.

A példákban eseményfigyelőt fogunk használni külső fájlokban, de természetesen bármelyik fenti megoldást használhatjuk, amelyik tetszik.

2. Nyelvi alapok

Ebben a fejezetben megnézzük, hogy hogyan tudunk egyszerű, már ismerős algoritmusokat írni a nyelvben. Milyen lehetőségeink vannak a kiíratásra, a kód tesztelésére. Olyan alapvető beépített lehetőségeket fogunk kipróbálni, mint a véletlenszám generálás, adatok bekérése a felhasználótól, különféle kiírási módok, dátumkezelés, string kezelés és összetettebb tömbös feladatok is lesznek.

2.1. Számoljuk ki egy téglalap kerületét és területét

Egy tipikus, valószínűleg mindenki számára ismert bemelegítőfeladattal kezdünk. Számoljuk ki a téglalap kerületét és területét úgy, hogy ezeket az adatokat bekérjük a felhasználótól, majd kiírjuk a végeredményt. A program végrehajtásakor a következőkre figyeljünk oda:

- Az értékeket a beépített prompt() függvény segítségével végezzük el.
- Fontos, hogy ha két számot úgy adunk össze, hogy azok string-ek, akkor nem összeadás, hanem konkatenáció fog történik, azaz 2 + 3 = 5, de „2” + „3” = „23”
- Ne felejtsük el, hogy mindennemű bemenet a nyelvben string értékként érkezik be, azaz konvertálnunk kell a műveletek végrehajtásához!
- Az eredményt alert() függvény segítségével írjuk ki.

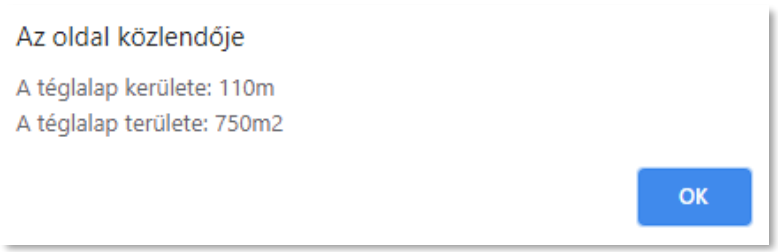
A feladat megoldása:

```
window.addEventListener("load", function() {
    const A = parseInt(prompt("Kérem a téglalap a) oldalát:"));
    const B = parseInt(prompt("Kérem a téglalap b) oldalát:"));

    const K = 2 * (A + B);
    const T = A * B;

    alert("A téglalap kerülete: " + K + "m\nA téglalap területe: "
    + T + "m2");
});
```

Nézzük meg a megoldás egyes részeit részletesebben. Alapvetően minden értéket konstansként fogunk eltárolni, amit nem változtatunk meg. Változóként azokat az értékeket fogjuk bevezetni, melyek változhatnak a program futása során, így a téglalap A és B oldalait konstansként deklaráljuk.

Beolvasást követően a beérkező értékeket átkonvertáljuk a parseInt() függvény segítségével. Így a string értékek ezt követően egészként lesznek kezelve. Ha tört számokat szeretnénk konvertálni, használjuk a parseFloat() függvényt.

Kiíratásnál az alert() függvényben használhatnk védőkaraktereket, amilyn a \n is. Ennek segítségével külön sorokra tördelhetjük a kimenetet.



2.2. Magyar dátumformátum

Készítsünk olyan programot, ahol magyar formátumban írjuk ki az aktuális dátumot dátumkezelő függvények segítségével. A formátum legyen a következő: 2020. május 09.

- A feladat megoldásához a W3Schools weblapon számos dátumkezelő getter függvényt találunk, mely segítségével le tudjuk kérdezni egy dátum objektum különböző értékeit.
- A hónapnevek magyarul történő kiírásához használjunk switch szerkezetet.
- A pontos dátum jelenjen meg egy üres DIV-ben a HTML oldalon.

A feladat megoldása:

```
const dt = new Date();
const ev = dt.getFullYear();
let ho = dt.getMonth() + 1;
const nap = dt.getDate();

switch(ho) {
    case 1 : ho = "január"; break;
    case 2 : ho = "február"; break;
    ...
    case 12 : ho = "december"; break;
}

const datum = document.getElementById("datum");
datum.innerText = ev + ". " + ho + " " + nap + ".";
```

Először példányosítanunk kell a Date() osztályt, azaz szükségünk van egy dátum objektumra, ugyanis azon tudjuk meghívni a gettereket. A hónap lekérdezésénél van egy kis trükk, ugyanis a hónapokat a nyelv alapból 0-11-ig határozza meg, így ehhez hozzáadtunk egyet.

A swtich szerkezetben pedig 12 ágat definiálunk, ahol megadjuk a hónapok neveit magyarul, majd ezzel a szöveges értékkel írjuk felül a ho változó tartalmát, ami épp emiatt nem konstansként került bevezetésre. A kódban nem a teljes szerkezet látható helyhiány miatt.

A végén pedig eltároljuk a datum azonosítóval rendelkező DIV-et a HTML DOM-ból egy változóban objektumként. Ezen az objektumon meghívjuk az innerText tulajdonságot, amivel a DIV címkéi közé tudunk szöveges értékeket beleírni. String összefűzéssel, azaz konkatenálással összeállítjuk a dátumot, majd megjelenítjük a HTML dokumentumban.

2.3. Számláló

Most készítsünk egy számlálót, ami visszaszámlál 60 másodperctől nulláig. A következőkre figyeljünk:

- A feladat megoldása során ne használjunk dátumkezelő függvényeket!
- A visszaszámlálást a setInterval() függvénnyel oldjuk meg, mely másodpercenként automatikusan kerüljön meghívásra.
- A függvény automatikusan álljon le, ha az érték eléri nullát. Ehhez használjuk a clearInterval() függvényt.

A feladat megoldása:

```
let mp = 60;
const szamlalo = document.getElementById("szamlalo");

const ismetel = setInterval(function() {
    szamlalo.innerText = mp--;

    if(mp < 0) clearInterval(ismetel);
},1000);
```

Először eltároljuk a DIV objektumot a HTML dokumentumból egy konstansban, ahol megjelenítjük majd a számláló értékét.

Ezután eltároljuk szintén egy konstansban a setInterval() függvényt, amire azért van szükség, hogy le tudjuk állítani a folytonos, végtelenségig tartó ismétlődését. A függvénynek két paramétere van, egy anonim függvény, illetve egy számérték. Az anonim függvény kerül ismételten végrehajtásra a számértéknek megfelelő sűrűségben. A számérték ms-ban van megadva, azaz 1 másodperc 1000 ms-ot jelent.

A függvény először kiírja a DIV belsejébe az mp változó egy egésszel csökkentett értékét. Ha ott már szerepelt valamilyen érték, akkor az felülírásra kerül.

Ezután egy elágazással ellenőrizzük, hogy ha az mp változó értéke kisebb, mint 0, akkor egyszerűen töröljük az intervallumot. Ezt a clearInterval() függvénnyel tehetjük meg, paraméterként a változót kell átadnunk, amibe eltároltuk a programunkban a setInterval()-t.

2.4. Digitális óra

Az előző két feladat alapján már képesek vagyunk elkészíteni egy digitális órát is. Most írjuk ki konzolra a pontos időt a következő formátumban: 16:23:55.

- Függvény segítségével kezeljük le azt a lehetőséget is, amikor 10-nél kisebb egy számérték, azaz jelenjen meg előtte egy nulla. Például: 6 helyett legyen 06.
- Használjuk a setTimeout() függvényt a setInterval() helyett.
- Az idő változtatásához folyamatosan új dátum példányra van szükségünk.

A feladat megoldása:

```
const ora = document.getElementById("ora");
let dt;

function ido() {
    dt = new Date();
    ora.innerText = tiznelKisebb(dt.getHours()) + ":" +
    tiznelKisebb(dt.getMinutes()) + ":" + tiznelKisebb(dt.getSeconds());

    setTimeout(ido,1000);
}

function tiznelKisebb(szam) {
    return szam < 10 ? "0" + szam : szam;
}

ido();
```

A setTimeout() függvényről azt kell tudnunk, hogy csak egyszer fut le. Ha szeretnénk vele helyettesíteni a setInterval()-t, akkor rekurzívan kell hivogatni a függvényt, amiben elhelyezzük. Esetünkben ez az ido() függvény.

A függvényben először létrehozunk egy dátum objektumot, majd innerText-el kiírjuk a DIV-be a megfelelő formátumban a pontos időt. Végül a setTimeout() másodpercenként rekurzívan hívja ezt a függvényt, így a futás sosem fog leállni.

A tiznelKisebb() függvény vár egy paramétert, azaz egy egész értéket. Azt vizsgálja egy ternary segítségével, hogy a kapott szám kisebb-e, mint 10. Ha kisebb, akkor elé írunk egy string nulla értéket, egyébként visszatérünk az eredeti számértékkel.

2.5. Gondoltam egy számra játék

Készítsünk egy egyszerű játékot, ahol a számítógép gondol egy számra 1 és 20 között. Ezt véletlenszám generátor segítségével fogjuk meghatározni. A program készítésekor a következőkre figyeljünk:

- Kérjünk be egy számot a felhasználótól.
- A találgatás addig folytatódjon, amíg nem találja el a játékos a gondolt számot vagy ha rossz bemenetet adott meg a felhasználó.
- A felhasználónak adjunk lehetőséget a kilépésre, ha a prompt() felugró ablakában a Mégse feliratra kattint. Ekkor a függvény null értékkel tér vissza. Kilépéskor köszönjünk el a felhasználótól egy alert() segítségével.
- Ha a felhasználó eltalálta a játékot, írjuk ki neki, hogy hány próbálkozásból találta ki, illetve hogy mi volt a helyes megfejtés.
- A véletlenszám generáláshoz a következő képletet használjuk:

$$Math.floor(Math.random() * (max - min + 1)) + min;$$

A feladat megoldása:

```
const general = Math.floor(Math.random() * (20 - 1 + 1)) + 1;
let tipp, probalkozasok = 0;

do {
    tipp = prompt("Melyik számra gondoltam 1 és 20 között?");
    if(tipp !== null) {
        tipp = parseInt(tipp);
        probalkozasok++;
    } else {
        alert("Köszönjük a játékot!");
        break;
    }
} while(tipp !== general);

alert("A gondolt szám: " + general +
"\nA próbálkozások száma: " + probalkozasok);
```

Az olyan jellegű feladatokat, amikor mindenképpen jó bemenetet várunk, vagy valamilyen végjelig olvasunk be értékeket, általában do-while vagy while ciklusokkal végezzük el. Esetünkben a hátultesztelő ciklus a kézenfekvőbb megoldás, hisz egyszer mindenképpen fog tippelni a felhasználó vagy megszakítja a ciklus futását a mégse gombra kattintva.

Az első sorban generáltunk egy véletlenszámot 1 és 20 között a képlet segítségével. A floor() függvény segítségével válthatjuk ki a Java-ból ismert integer értékké történő kasztolás műveletét, azaz itt egyszerűen a tört számot lefelé kerekítjük függetlenül annak az értékétől. Bizonyára emlékszünk rá, hogy a véletlenszám generálás egy 0 és 1 között tört szám, ami sosem lesz egy.

Ezután felveszünk két segédváltozót. A tipp a játékos tippjeit fogja tárolni, míg a probalkozasok változó a próbálkozások számát.

A ciklus először bekéri a tippet a játékostól, majd egy elágazás ellenőrzi, hogy a kapott érték null-e. Ugyanis, ha null értéket kapunk, a felhasználó szeretné leállítani a ciklust, azaz a mégse gombra kattintott.

Ha az érték nem null, akkor átkonvertáljuk a beolvasott értéket egészre, ami ha nem egész, akkor NaN (not a number) értéket fogunk kapni, illetve növeljük a próbálkozások számát.

Ha null értéket kaptunk, elbúcsúzunk a felhasználótól, majd megszakítjuk a ciklus futását egy break utasítással.

A ciklus feltétele egyszerű, addig fut, amíg nem találjuk el a gondolt számot.

A program végül kiírja az adatokat, amiket a feladat kért.

2.6. Lottószám generátor

Generáljunk öt darab számot 1 és 90 között, majd írjuk ki a számokat növekvő sorrendbe a konzolra.

- A generált számokat tároljuk el egy tömbben.
- Kizárólag tömbkezelő függvényeket használjunk a tömb manipulálásához (elem hozzáadása).
- Egy számot csak egyszer húzhatunk ki, tehát a tömbben nem szerepelhet két egyforma szám.
- A rendezéshez szintén használjuk a megfelelő tömbkezelő függvényt.

A feladat megoldása:

```
const lotto = [];
let general;

do {
    general = Math.floor(Math.random() * (90 - 1 + 1)) + 1;
    if(lotto.indexOf(general) === -1) {
        lotto.push(general);
    }
} while(lotto.length < 5);

lotto.sort();

console.log("A generált számok: " + lotto.join(", "));
```

Készítünk egy üres tömböt. A tömb konstans lesz, hisz magát a tömböt nem fogjuk kicserélni objektumra vagy string értékre vagy számra, tehát semmire. A függvényhívás egy adott konstanson pedig nem befolyásolja annak az értékét.

Szintén egy do-while ciklust használunk, ahol minden futáskor generálunk egy 1 és 90 közötti egész számot. Ezután mielőtt a generált számot a tömbbe töltenénk, megvizsgáljuk az indexOf() tömbkezelő függvény segítségével, hogy szerepel-e a tömbben már ilyen szám. Ha igen, akkor visszaadja az index helyet, ahol már szerepel a generált szám, ha nem, akkor viszont a -1 értékkel tér vissza. Azért a -1-el,

mivel ez az első olyan érték, ami nem lehet indexszám. Ugyanis a programozásban 0-tól indexelünk és az indexérték nem vehet fel negatív számot.

A tömböt a push() metódussal lehet feltölteni, paraméterként a beszúrandó értéket kell átadni.

A ciklus addig folytatja a futást, amíg a tömb hossza el nem éri az ötöt, azaz nem került kihúzásra öt szám.

A sort() metódussal rendezzük a tömb elemeit.

Végül pedig kiírjuk az eredményt. A join() metódus a paraméterben átadott módon fogja a tömb elemeit elválasztani egymástól, majd string értékké átkonvertálni. Így nem lesz szükségünk ciklusra a tömb elemeinek a kipörgetéséhez.

2.7. Palindrom

Bizonyára ismerősek az olyan mondatok, amelyek oda-vissza olvasva egyformán ugyanazt jelentik. Nos, egy ilyen programot fogunk írni ebben a példában.

- Kérjünk be egy tetszőleges szöveg a felhasználótól.
- A kapott szöveget tároljuk el visszafelé.
- A szóközöket és az írásjeleket az összehasonlítás során ne vegyük figyelembe!
- Ha palindrom, zöld színnel, ha nem palindrom, pirossal jelenítsük meg egy DIV elemben.
- A programnak nem kell kezelnie a kettős magánhangzókat (pl.: sz, ny, zs stb.)
- Néhány mondat:
    - A Kupa apuka.
    - Angol pap lógna.
    - A kultúra rút luka.

A feladat megoldása

```
const palindrom = document.getElementById("palindrom");
const mondat = prompt("Kérek egy mondatot:");
let betuk = "", forditott = "";
const tiltott = [" ", ".", ",", "?", ":", "!"];

if(mondat !== null) {
    for(let i=0;i<mondat.length;i++) {
        if(tiltott.indexOf(mondat.charAt(i)) === -1) {
            betuk += mondat.charAt(i).toLowerCase();
        }
    }

    for(let j=betuk.length-1;j>=0;j--) {
        forditott += betuk.charAt(j);
    }

    palindrom.innerHTML = forditott

    if(betuk === forditott) palindrom.style.color = "green";
    else palindrom.style.color = "red";
}
```

Először eltároljuk a DOM-ban lévő DIV elemet egy konstansba. Ezután bekérünk egy mondatot a

felhasználótól, majd létrehozunk két segédváltozót. A tiltott karaktereknek létrehozhatunk egy tömböt, amibe felsoroljuk azokat az érétkeket, amiket nem fogadunk el.

Egy elágazással ellenőrizzük, hogy a bemenet értéke null-e. Ugyanis null esetén a ciklus nem tudja lekérdezni a változó length értékét.

Az első ciklus végigmegy a beolvasott mondaton karakterenként, majd megnézi, hogy az aktuálisan vizsgált karakter szerepel-e a tiltott karakterek tömbben. Ha szerepel, akkor nem szúrja be, ha nem szerepel, akkor beszúrja a betuk változóba a vizsgált karaktert.

A második ciklus pedig a már szóközöktől, írásjelektől mentes betuk változó tartalmát eltárolja visszafelé a forditott változóban.

Ezután megjelenítjük a forditott változó tartalmát a palindrom DIV elemben.

Végül egy elágazással vizsgáljuk a két új változónk értékét, ha azok megegyeznek, akkor zöld színnel jelenítjük meg a szöveget, ha nem, akkor pirossal. A CSS beállításokat a style tulajdonságon keresztül tudjuk beállítani a javascript-ben.

2.8. Páros és páratlan tömbök

Kérjünk be számokat a felhasználótól nulla végjelig, azaz amíg nulla értéket nem ír be.

- A páros számokat, illetve a páratlanokat külön tömbben gyűjtsük.
- Csak szám típusú értéket tároljunk el a tömbben!
- A program írja ki mindkét tömb tartalmát úgy, hogy a nulla ne kerüljön be sehová.
- Írjuk ki, a hosszabb tömb elemeinek számát, a páros tömb szorzatát és a páratlan átlagát.
- Használjunk forEach() tömbkezelő függvényt a feladat megoldásához.
- Az eredményeket egy DIV-be írjuk ki az oldalon.

A feladat megoldása:

```
let szam = 0, paros = [], paratlan = [], szorzat = 1, atlag = 0;
const eredmeny = document.getElementById("eredmeny");

do {
    szam = parseInt(prompt("Kérek egy számot:"));
    if(!isNaN(szam) && szam !== 0) {
        (szam % 2 === 0 ? paros.push(szam) : paratlan.push(szam));
    }
} while(szam !== 0);

const max = Math.max(paros.length, paratlan.length);

paros.forEach((elem) =>  szorzat *= elem);

paratlan.forEach((elem) => atlag += elem);
atlag /= paratlan.length;

eredmeny.innerHTML = "A páros számok: " + paros.join(", ") +
"<br>A páratlan számok: " + paratlan.join(", ") +
"<br>A hosszabb tömb elemeinek száma: " + max +
"<br>A páros számok szorzata: " + szorzat +
"<br>A páratlan számok átlaga: " + atlag;
```

Deklarálunk néhány változót a program elején. A szam változó fogja tárolni a bekért számot, lesz két tömbünk, amik külön-külön tárolják a beolvasott értékeket, illetve lesz a szorzat és az atlag segédváltozó. Ne felejtsük el, hogy szorzatképzésnél 1-re kell inicializálni a változót, különben mindig nulla eredményt fogunk kapni, hisz a nullával történő szorzás mindig nullát eredményez.

Egy do-while ciklus segítségével bekérjük a felhasználótól a számokat. Csak akkor tároljuk el a megfelelő tömbben a bekért számot, ha az valóban szám. A nulla érték nem kerül eltárolásra, hisz az a kilépési szándék jelzésére van fenntartva.

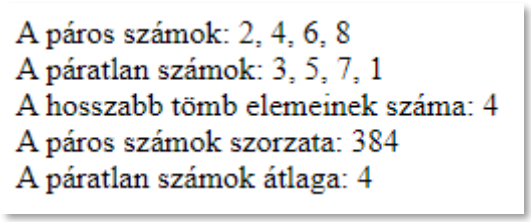
A páros és páratlan számok ellenőrzését egy ternary operátor végzi, így egyetlen sorban megvalósítható az elágazás.

A ciklus akkor fog leállni, ha nulla bemenetet kap a felhasználótól.

Ezután készítünk egy konstanst, ami a max függvény segítségével eldönti a két tömb közül, hogy melyik elemszáma a nagyobb. A feladat nem kérte a hosszabb tömb megnevezését, csak a hosszát.

Ezután következik a két forEach() függvény, ami egy beépített tömbkezelő függvény a javascript-ben. Ez a függvény végigiterálja a tömb elemeit, majd meghív minden egyes elemre egy függvényt, amit vagy anonim függvényként adtunk meg a programban. A függvénynél alkalmaztuk az ún. „fat arrow” szintaktikát, amivel lerövidíthetjük a függvény meghatározását. Amennyiben egyetlen sor utasítás kerül egy névtelen függvényben, úgy nem szükséges a kapcsos zárójelek kiírása.

Végül kiírtuk az eredményeket a DIV elembe.



2.9. Generált számok

Generáljunk számokat, amíg azok összege el nem éri a 100-at.

- Egy tömbben gyűjtsük a bekért számokat.
- A kapott számot vizsgáljuk meg, hogy ha hozzáadnánk a tömbhöz, akkor túllépnénk-e a 100-at vagy nem. Ha igen, akkor már ne adjuk hozzá és lépjünk ki a ciklusból.
- Az eredményt konzolon jelenítsük meg.

A feladat megoldása

```
const szazig = [];
let osszeg = 0, szam = 0;

do {
    szam = parseInt(prompt("Kérek egy számot:"));
    if(!isNaN(szam)) {
        if(osszeg + szam <= 100) {
            szazig.push(szam);
            osszeg += szam;
        } else break;
    }
} while(osszeg <= 100);

console.log("A tömb elemei: " + szazig.join(", ") +
"\nA számok összege: " + osszeg);
```

Készítünk egy tömböt szazig névvel, illetve két változót. Az egyikben a számok összegét fogjuk tárolni, a másikban pedig a bekért számot.

Egy do-while ciklus segítségével kérünk be számokat a felhasználótól addig, amíg a bekért számok összege meg nem haladja a 100-at, azaz kisebb vagy egyenlő, mint 100.

Egy egymásba ágyazott elágazással ellenőrizzük, hogy a bekért szám szám-e egyáltalán, majd azt, hogy a jelenlegi összeghez ha hozzáadnánk a bekért számot, akkor vajon túllépnénk-e a 100-at. Ha nem, akkor feltöltjük a tömböt elemekkel, illetve növeljük az összeg változó érétkét. Ellenkező esetben megszakítjuk a ciklus futását.

Végül kiírjuk az eredményeket.

A feladat megoldható lett volna úgy is, ha utólag számoljuk ki a tömbelemek alapján a számok összegét.

2.10. Hacker írásmód

Adjunk meg egy tetszőleges mondatot, melyet alakítsunk át hacker írásmódra a következők alapján:

- A következő betűket alakítsuk át számmá: l > 1; o > 0; e > 3; s > 5; ó > 6; t > 7; g > 9
- Az így átalakított mondatot írjuk ki konzolra.

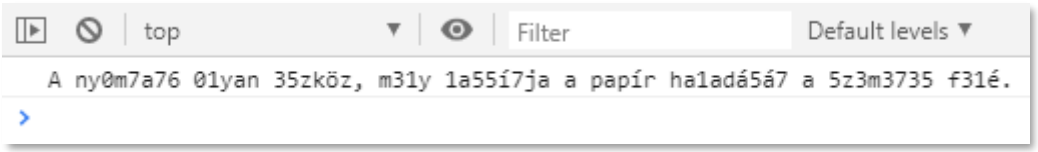
A feladat megoldása:

```
const mondat = "A nyomtató olyan eszköz, mely lassítja a papír haladását a
szemetes felé.";
let ujmondat = "";
const hacker = {l : 1, o : 0, e : 3, s : 5, ó : 6, t : 7, g : 9};
const hackerKeys = Object.keys(hacker);

for(let i=0;i<mondat.length;i++) {
    if(hackerKeys.indexOf(mondat.charAt(i)) !== -1) {
        ujmondat += hacker[mondat.charAt(i)];
    } else {
        ujmondat += mondat.charAt(i);
    }
}

console.log(ujmondat);
```

Először létrehoztunk egy példa mondatot, illetve létrehoztunk egy új változót az átalakított mondat számára. A hacker objektumban eltároltuk a kulcs-érték párokat az egyszerűbb kereshetőség érdekében. Ezt természetesen két külön tömbben is eltárolhattuk volna. Ezután az objektumunkból a kulcsokat átalakítottuk egy tömbbé, ahol az indexekhez kerülő értékek lettek az objektum kulcsai. Tehát a 0. indexen a hackerKeys tömbben az l betű szerepel, míg az 1. indexen az o betű és így tovább.

Egy for ciklussal végigmegyünk a mondat minden egyes karakterén. Ha az objektumkulcsokat tartalmazó tömbben megtalálható az aktuálisan vizsgált karakter, akkor az új mondatba lekérjük az objektum kulcsához kapcsolódó értéket. Ha nem, akkor egyszerűen beszúrjuk az eredeti mondatban lévő karaktert.

A feladat végén konzolra kiírjuk a megoldást.



2.11. Védett e-mail cím

Kérjünk be a felhasználótól egy e-mail címet, majd alakítsuk át védett formába a következők szerint:

- Az e-mail cím ASCII karakterekre legyen átalakítva, ehhez használjuk a charCodeAt()-et.
- Az egyes karaktereket ne csak ASCII kódban hagyjuk, hanem alakítsuk HTML karakterkóddá. Egy HTML karakterkód &# betűkkel kezdődik és pontosvesszővel végződik.
- A bekért e-mail címet írjuk ki a konzolra és jelenítsük meg egy DIV-ben is.

A feladat megoldása:

```
const email = document.getElementById("email");
let mail = "labamkozottvanegy@gmail.com";
let encrypted = "";

for(let i=0;i<mail.length;i++) {
    encrypted += "&#" + mail.charCodeAt(i) + ";";
}

email.innerHTML = encrypted;
console.log(encrypted);
```

Először elmentjük a DIV objektumot egy konstansba, majd megadunk egy példa e-mail címet. A segédváltozóban lesz az e-mailcím kódolt változata.

Egy ciklussal egyszerűen végig pörgetjük karakterenként a megadott e-mailt, majd minden egyes karakterét kicseréljük a charCodeAt() függvény segítségével. Ez azonban önmagában nem elég, mert csak számokat ad vissza. Ahhoz, hogy HTML karakterkódokká tudjuk alakítani ezeket a számokat, kell elé az &# jel, illetve a végére a pontosvessző.

Az eredményből láthatjuk, hogy valójában kódoltan kerül eltárolva kódszinten, a böngésző azonban képes megjeleníteni ennek ellenére is.

2.12. Az e-mail cím domain címe

Egy e-mailcím alapján határozzuk meg a címhez kapcsolódó domaint, azaz a kukac és a domainvégződés közötti részt.

- Stringkezelő függvények segítségével vágjuk ki dinamikusan egy e-mail cím domain nevét.
- A program bármilyen e-mail cím esetén működőképes kell, hogy legyen, akkor is, ha a domain tartalmaz pontot.
- Próbáljuk megoldani kétféleképpen a feladatot (indexOf(), substr() és substring() segítségével).
- Az eredményt írjuk ki a konzolra.

A feladat megoldása

```
let email = prompt("Kérek egy e-mail címet:"),
kukac = email.indexOf("@") + 1,
pont = email.lastIndexOf(".");

console.log("Az első megoldás: " + email.substring(kukac, pont));
console.log("A második megoldás: " +
email.substr(kukac ,Math.abs(kukac - pont)));
```

Először bekérünk egy e-mail címet, majd meghatározzuk a kukac helyét a címben az indexOf() függvény segítségével. Mivel ez a karakter pontos helyét határozza meg, ezért itt hozzáadunk plusz egyet, hogy a kivágást majd a rákövetkező karaktertől kezdje. Ezután meghatározzuk a cím hátuljától kezdve a pontot, mivel szerepelhet a domain címben is pont, így hátulról kezdjük el vizsgálni az első pont előfordulását.

Az első kiírásnál a substring() segítségével vágjuk ki a domain-t, azaz a változónevek alapján láthatjuk, hogy a kukac és a pont közötti részt.

A második kiírásnál viszont a substr() függvényt használtuk, ott pedig szintén a kukactól indultunk el, viszont a kukac és a pont indexértékének különbségét kell itt meghatároznunk, mivel ennél a függvénynél nem tól-ig történik a vágás, hanem a kezdőponttól meghatározott karakterszámot fogunk kivágni. Ez a különbség a substring() és a substr() függvények között.

2.13. HTML táblázat

Generáljunk HTML táblázatot a felhasználó által megadott dimenzióban.

- A program kérje be a táblázat oszlopainak és a sorainak a számát.
- Generáljuk le a táblázatot ciklusok segítségével.
- Az egyes cellákba generáljunk véletlenszámokat 1 és 100 között.
- A páros számokat zölddel, a páratlanokat pirossal jelöljük a táblázatban.
- A táblázatot jelenítsük meg a BODY elemben a HTML dokumentumban.

A feladat megoldása

```
let general, tablazat = "<table border=\"1\">";
const oszlopok = parseInt(prompt("Kérem az oszlopok számát:")),
sorok = parseInt(prompt("Kérem a sorok számát:"));

for(let i=0;i<oszlopok;i++) {
    tablazat += "<tr>";
    for(let j=0;j<sorok;j++) {
        tablazat += "<td>" + (Math.floor(Math.random() *
        (100 - 1 + 1)) + 1) + "</td>";
    }
    tablazat += "</tr>";
}
tablazat += "</table>";

document.body.innerHTML = tablazat;
```

Felvettünk két segédváltozót, egyet a generálandó számoknak, egyet pedig a HTML táblázatnak. Létrehoztunk egy konstanst a beolvasott sorok és oszlopok számának. A program nem kezeli azt a lehetőséget, ha a felhasználó nem számértéket ad meg.

Egymásba ágyazott ciklusok segítségével beleírjuk a segédváltozóba a HTML táblázatot. A külső ciklus az oszlopokat generálja le, míg a belső ciklus a sorokat, illetve a cellákban lévő számokat.

A program végül a BODY elembe beszúrja a változóban lévő HTML kódot.

13	99	83	72	25	59	16	82
10	12	36	32	100	86	75	30
14	49	44	46	64	34	75	89
51	87	37	88	58	86	36	23

2.14. Ékezettelenítő program

Számos esetben szükségünk lehet arra, hogy lecseréljük az ékezetes vagy különleges karaktereket tartalmazó szövegekben ezeket a karaktereket. Például egy e-mail cím esetében. Írjunk programot, ami képes egy megadott teljes név alapján e-mail címet generálni egy cég számára.

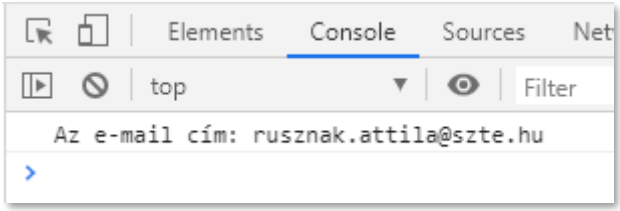
- Kérjük be a felhasználó teljes nevét.
- A nevet alakítsuk át e-mail címmé úgy, hogy a névben szereplő ékezeteket cseréljük ékezet nélkülire, a szóközöket pedig pontra.
- Ha szerepel a névben titulus, azt a név végére fűzzük hozzá szintén ponttal.
- Például: Dr. Bekre Pál > dr.bekre.pal@szte.hu

A feladat megoldása

```
const nev = prompt("Kérem a teljes neved:").toLowerCase(),
tiltott = {
    á: "a", é: "e", í: "i", ó: "o", ö: "o", ő: "o",
    ú: "u", ü: "u", ű: "u", ".":"", " ": "."
},
tiltottKulcsok = Object.keys(tiltott);
let ujnev = "";

for(let i=0;i<nev.length;i++) {
    if(tiltottKulcsok.indexOf(nev.charAt(i)) !== -1) {
        ujnev += tiltott[nev.charAt(i)];
    } else {
        ujnev += nev.charAt(i);
    }
}

console.log("Az e-mail cím: " + ujnev + "@szte.hu");
```

Először bekérjük a felhasználó nevét, majd azonnal át is alakítjuk csupa kisbetűsre. Az e-mail címeknél nyilván nem számít a kis és a nagybetű, viszont az átalakításnál így elég csak a kisbetűk kulcs-érték párjait megadnunk. Szintén objektumot használunk, hasonlóan egy régebbi feladathoz.

Egy ciklussal ellenőrizzük egyesével a névben szereplő karaktereket, és ha talál a tiltott listán lévő karaktert, akkor kicseréli a megfelelő értékre. A végén kiírjuk az e-mail címet úgy, hogy mögé írjuk a kukacot és a fennmaradó részét a címnek.



3. Összetett tömbös feladatok

Ebben a fejezetben olyan feladatokat fogunk gyakorolni, ahol leginkább tömbökkel történő műveletvégzés lesz. Egy feladat több részből áll, mely részek egymásra épülnek. Ezek a feladatok segítenek begyakoroltatni a tömbökben történő gondolkodást és a tömbműveleteket.

3.1. „A” feladatsor

- Kérjünk be a felhasználótól 5 számot, melyeket egy tömbben tároljunk le. Bekéréskor írjuk ki a felhasználónak, hogy hányadik számot kérjük. Pl.: "Kérem a(z) 1. számot:", és mindaddig ne lépjen a program a következő számra, amíg nem kaptunk helyes bemenetet.
- A bekért számok értéke csak egy 0 és 50 közötti szám lehet.
- Minden szám értékét a tömbben írjuk át %-os értékre, ahol az intervallum maximuma legyen a 100%, azaz az 50. A százalékos értékeket ne szövegesként, hanem egészként tároljuk el. A 100 százalék tehát 100 legyen, ne pedig „100%”.

A részfeladat megoldása:

```
let szamok = [], n = 1, szam = 0;

do {
    szam = parseInt(prompt("Kérem a(z) " + n + ". számot
    (0 és 50 között):"));

    if(!isNaN(szam) && (szam >= 0 && szam <= 50)) {
        szamok.push(szam);
        n++;
    }
} while(szamok.length < 5)

for(let i=0; i<szamok.length;i++) {
    szamok[i] *= 2;
}

console.log("Eredmény: " + szamok.join(", "));
```

- A tömbben lévő számok közül csak azokat az elemeket hagyjuk meg, melyek a tömb 0. indexétől számolva maximum 55%-ot adnak ki. A többi elemet töröljük!
- Törlésnél magát az elemet is távolítsuk el, ne csak az értéket.

A részfeladat megoldása:

```
let osszeg = 0;
const tombHossz = szamok.length;

for(let i=0; i<tombHossz;i++) {
    osszeg + szamok[i] <= 55 ? osszeg += szamok[i] : szamok.pop();
}
console.log("Eredmény: " + szamok.join(", "));
```

Fontos, hogy ebben az esetben mentsük ki külön konstansba a változó hosszát és azt használjuk a ciklusban, mivel a tömb hossza a ciklusban történő törlés miatt változik.

- A maradék elemeket egészítsük ki egy olyan %-os értékkel, melyet ha hozzáfűzünk a tömb jelenlegi elemeihez, ismételten 100%-ot kapjunk.

A részfeladat megoldása:

```
osszeg = 0;
for(let i=0;i<szamok.length;i++) {
    osszeg += szamok[i];
}
szamok.push(100-osszeg);
console.log("Eredmény: " + szamok.join(", "));
```

- Készítsünk a tömbből egy másolatot. Az új tömbben a %-os értékeket számoljuk vissza eredeti értékükre, majd mindkét tömböt jelenítsük meg konzolon.

A részfeladat megoldása:

```
let szamok2 = [];

for(let i=0;i<szamok.length;i++) {
    szamok2.push(szamok[i] /= 2);
}

console.log("Eredmény:\nSzázalék értékek: " + szamok.join(", ") +
"\nEredeti értékek: " + szamok2.join(", "));
```

3.2. „B” feladatsor

- Kérjünk be a felhasználótól pontosan 5db. számot. Más bemenet nem elfogadott.
- A kapott számokat rendezzük növekvő sorrendbe.
- Írassuk ki, hogy a középső elemtől balra lévő elem mennyivel kisebb, illetve a tőle jobbra lévő elem mennyivel nagyobb.
- A középső elem meghatározásának bármekkora elemszámmal működnie kell! Meghatározásához használjuk a tömbnek a hosszát, a kapott eredményt kerekítsük!

A részfeladat megoldása:

```
const szamok = [];
let n = 1, szam = 0, kozepso, balra, jobbra;

do {
    szam = parseInt(prompt("Kérem a(z) " + n + ". számot:"));

    if(!isNaN(szam)) {
        szamok.push(szam);
        n++;
    }

} while(szamok.length < 5)

szamok.sort(function(a, b) { return a-b; });

kozepso = Math.round(szamok.length / 2)-1,
balra = Math.abs(szamok[kozepso]-szamok[kozepso-1]),
jobbra = Math.abs(szamok[kozepso]-szamok[kozepso+1]);

console.log(szamok.join(", "));
console.log("Balra lévő: " + balra + ", jobbra lévő: " + jobbra);
```

A sort() függvényt már ismerjük, viszont alapértelmezetten nem rendez jól számokra. Ezért ilyenkor át kell neki adni egy anonim függvényt, ami megcserélgeti a tömb szomszédos elemeit.

- Az előző két elemet módosítsuk úgy, hogy a középső elemtől balra, ill. jobbra lévő elemek egyformán 3 szám „távolságra” legyenek.
- Ha a középső elem 6, akkor a 6-nál kisebb szám és a 6-nál nagyobb szám is egyforma messzire legyen. A kisebb szám mondjuk ha 3, akkor a nagyobbnak 9-nek kell lennie. Így 3-3 távolságnyira lesznek.

A részfeladat megoldása:

```
const elteres = szamok[kozepso];
szamok[kozepso-1] = elteres - 3;
szamok[kozepso+1] = elteres + 3;

console.log("Eredmény: " + szamok.join(", "));
```

- Számoljuk át %-os értékekre a tömbelemeket úgy, hogy a 100%-nak a legnagyobb tömbelem feleljen meg. A számokat kerekítsük!

A részfeladat megoldása:

```
const legNagy = szamok[szamok.length-1];

for(let i=0;i<szamok.length;i++) {
    szamok[i] = Math.round(szamok[i] / (legNagy/100));
}

console.log("Eredmény: " + szamok.join(", "));
```

A tömb mivel rendezve van, így a legnagyobb elem mindig az utolsó eleme lesz.

- Kérjünk be a felhasználótól egy számot a következő megkötésekkel: a szám csak 10 és 100 közötti lehet. A megadott számérték alapján írjuk ki konzolra, hogy hány elemre lenne szükségünk legalább, hogy elérjük (de ne lépjük túl) a felhasználótól kapott %-os értéket. Azaz a tömbben hány százalékos értékre lenne szükségünk a 0. indextől számolva, ami még belefér a felhasználó által megadott értékbe, de nem lépi azt túl.

A részfeladat megoldása:

```
let osszeg = 0, elemek = 0;

do {
    szam = parseInt(prompt("Kérem a számot:"));
} while(isNaN(szam) || (!isNaN(szam) && (szam > 100 || szam < 10)))

for(i=0;i<szamok.length;i++) {
    if((osszeg + szamok[i]) <= szam) {
        osszeg += szamok[i];
        elemek++;
    }
}
console.log("A felhasználó bemenete: " + szam +
"\nA szükséges elemek: " + elemek);
```

3.3. „C” feladatsor

- Kérjünk be egy számot a felhasználótól, majd készítsünk belőle ötöt úgy, hogy felszorozzuk az értékét. Ha a kapott szám 5, akkor a további 4 számot így alakítjuk ki: 5 × 2, 5 × 4, 5 × 8 – azaz a szorzószám négyzetesen növekszik.
- A kapott számokat tömbben tároljuk le.

A részfeladat megoldása:

```
const szamok = [];
let szam = 0;

do {
    szam = parseInt(prompt("Kérek egy számot:"));
} while(isNaN(szam));

for(let i=0; i<5; i++) {
    szamok.push(szam * Math.pow(2,(i+1)));
}

console.log("Eredmény: " + szamok.join(", "));
```

A számok felszorzását a pow() függvénnyel csináltuk meg, ami megkönnyíti a hatványozás műveletét.

- A számok közül rakjuk középre a legnagyobb elemet.

A részfeladat megoldása:

```
const legNagy = szamok.length-1,
kozepso = Math.ceil(szamok.length/2)-1;
let temp = szamok[kozepso];

szamok[kozepso] = szamok[legNagy];
szamok[legNagy] = temp;

console.log("Eredmény: " + szamok.toString());
```

A ceil() függvényét használtuk a Math osztálynak, ami felfelé kerekít minden alkalommal. Öt elemnél ez teljesen jó megoldás, ugyanis 5/2 = 2.5 → 3 felfelé kerekítve és a -1 azért van ott, mert 0. indextől számoljuk az elemeket.

Másik megoldás kerekítésre lehetne még a *`Math.floor(szamok.length/2);`* Ez esetben a középső elem meghatározható floor() függvénnyel is, azaz állandó jellegű lefelé kerekítéssel. Ez öt elemnél: 5/2 = 2.5 → 2 lefele kerekítve. Itt nem kell már kivonni értéket, mert tökéletesen megfelel a kezdeti 0. indexnek, így valóban a középső elemet fogjuk kiválasztani.

- Írjuk ki, hogy mennyit kell hozzáadni a legkisebb elemhez, hogy megkapjuk a legnagyobbat!

A részfeladat megoldása:

```
let minElem = maxElem = szamok[0];

for(i=0; i<szamok.length; i++) {
    minElem = Math.min(minElem,szamok[i]);
    maxElem = Math.max(maxElem,szamok[i]);
}

console.log("A két szám közötti különbség: " + Math.abs(minElem-maxElem));
```

Ez egy különleges példája a minimum- és maximumkiválasztás algoritmusának. Itt ugyanis a két beépített függvényt használjuk a tömb elemeivel történő összehasonlításra ezzel megspórolva az elágazást, ami vizsgálja az értékeket. Az abszolútérték függvény azt a célt szolgálja, hogy mindenképpen pozitív eredményt fog visszaadni, így nem kell odafigyelnünk a kivonás sorrendjére.

- Írjuk ki, hány % különbség van a legkisebb és a legnagyobb tömbelem között!
- Csak kiírásnál használjuk a % jelet! A számokat számként tároljuk!

A részfeladat megoldása:

```
let kisSzazalek = 100;
for(i=0;i<szamok.length;i++) {
    szamok[i] /= (maxElem/100);
    if(szamok[i] < kisSzazalek) {
        kisSzazalek = szamok[i];
    }
}
console.log(100-kisSzazalek +
"% különbség van a legnagyobb és legkisebb elem között.");
```

Tekintve, hogy hatványozunk, az értéknek mindig 93,75%-nak kellene lennie.

4. Objektumorientált programozás

A javascript az utóbbi évek során egyre szélesebb támogatást kapott az objektumorientáltságban. Körülbelül 2 éve már megjelentek benne az osztályok is, támogatja a konstruktorokat, az osztályváltozókat, a gettereket és a settereket, sőt még az öröklést és a statikus tagokat is. Ebben a fejezetben a nyelv régi és új objektumorientált képességeit fogjuk megnézni és feladatok formájában kipróbálni.

4.1. Fájlkezelő prototípus

Készítsünk egy fájlok prototípust a következő adatokkal:

- Tulajdonságok:
  - ***fajlok:*** tömb, melyben fájlnevek szerepelnek kiterjesztéssel (ezt kívülről kapja meg)
  - ***hanyFajl:*** a fájlok számát adja eredményül
- Metódusok:
  - ***listaz():*** kilistázza a fájlok nevét
  - ***torol():*** kitöröl egy tetszőleges fájlt a fájlnév alapján
  - ***atnevez():*** átnevez egy tetszőleges fájlt
  - ***letrehoz():*** létrehoz egy tetszőleges fájlt

A részfeladat megoldása:

```
function fajlok(fajlok) {
    this.fajlok = fajlok;
    this.hanyFajl = this.fajlok.length;
    this.listaz = function() {
        console.log(this.fajlok.join(", "));
    };
    this.torol = function(fajl) {
        if(this.fajlok.indexOf(fajl) !== -1) {
            this.fajlok.splice(this.fajlok.indexOf(fajl),1);
            this.hanyFajl--;
        } else {
            console.log("Hibás fájlnév!");
        }
    };
    this.atnevez = function(reginev, ujnev) {
        if(this.fajlok.indexOf(reginev) !== -1) {
            this.fajlok.splice(this.fajlok.indexOf(reginev),1, ujnev);
        } else {
            console.log("Hibás fájlnév!");
        }
    };
    this.letrehoz = function(fajl) {
        this.fajlok.push(fajl);
        this.hanyFajl++;
    }
}
```

Létrehoztunk egy prototípust, ami lényegében egy javascript alapú konstruktor leegyszerűsített változata. Úgy néz ki, mint egy függvény és vár egy paramétert, ami egy fájlneveket tartalmazó tömb lesz majd.

Tartalmaz egy hanyFajl tulajdonságot is, ami a kapott tömb elemeinek a számát kérdezi le és tárolja el.

Ezután van egy listaz() metódus, ami egyszerűen string fromátumra alakítja a már ismert join() függvény segítségével a tömb elemeit.

A torol() metódus a kapott fájlnevet megkeresi az indexOf() függvény segítségével a tömbben. Ha az megtalálható, akkor a splice() beépített tömb manipuláló függvény kitörli az indexOf() által visszaadott indexértékről. Végül ugyanitt csökkentjük a tömbben lévő elemeinek a számát, bár itt lekérdezhettük volna a tömb új hosszát is. Ha a fájlnév nem található, konzolra kiírjuk hogy hibás a fájlnév.

Az atnevez() metódus vár egy jelenlegi fájlnevet és egy új nevet. A jelenlegi név alapján ismét az indexOf() függvény segítségével megnézzük, hogy szerepel-e a keresett név a tömbben. Ha igen, akkor megint csak a splice() függvénnyel kicseréljük a régi fájlnevet az új névre.

Végül a letrehoz() függvény vár egy fájlnevet, amit beszúr a tömb végére a push() segítségével és növeljük a tömb hosszát tároló hanyFajl tulajdonságot is.

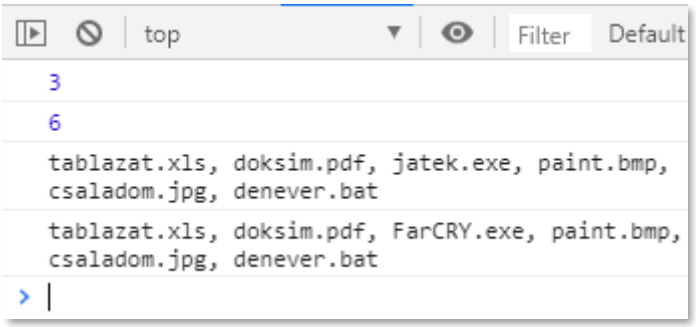
- Most az előző prototípusból hozzunk létre egy objektumot, majd hajtsuk végre a következő feladatokat:
  - Kérdezzük le a fájlok számát
  - Hozzunk létre 3 fájlt
  - Kérdezzük le a fájlok számának új állapotát
  - Listázzuk ki a fájlokat
  - Nevezzünk át egy nem a fájlok elején vagy végén lévő fájlt

A részfeladat megoldása:

```
let TCM = new fajlok(["tablazat.xls", "doksim.pdf", "jatek.exe"]);
console.log(TCM.hanyFajl);

TCM.letrehoz("paint.bmp");
TCM.letrehoz("csaladom.jpg");
TCM.letrehoz("denever.bat");
console.log(TCM.hanyFajl);

TCM.listaz();
TCM.atnevez("jatek.exe","FarCRY.exe");
TCM.listaz();
```

Először készítünk egy saját objektumot a new kulcsszóval, majd a konstruktornak átadjuk a fájlneveket tartalmazó tömböt. Ezt utána kilistázhatjuk a konzolra, hogy megnézzük a fájlok számát.

Hozzáadunk még 3db új fájlt a tömbhöz. Ismét kiírjuk, hogy hány fájlunk van. Majd listázzuk a fájlneveket, végül pedig átnevezzük az egyik fájl nevét.



4.2. Lottószám generátor prototípus

Írjunk egy „lotto” konstruktort, ami a következő értékeket várja: a húzások számát (hány számot húzzon a gép), a játékos tippjeit tömb formában, a tippelhető lottószámok legkisebb és legnagyobb értékét. A következőképpen építsük fel a konstruktorunkat:

- Tulajdonságok:
  - ***huzas:*** a húzások maximális száma (kívülről jön)
  - ***min:*** a generálandó számok minimuma (kívülről jön)
  - ***max***: a generálandó számok maximuma (kívülről jön)
  - ***gepi:*** a general() függvény által feltöltendő tömb, kezdetben üres tömb
  - ***tipp:*** tömb, a játékos tippjeivel (kívülről jön)
  - ***talalat:*** az ellenoriz() függvény által meghatározott szám, kezdetben 0
  - ***sorsolasok:*** az eddigi összes sorsolás száma (egy objektumra nézve)
- Metódusok:
  - ***general():*** a huzas tulajdonságnak megfelelő számú tömböt generál véletlen szám generátor segítségével. Egy számot csak egyszer húzzunk ki!
  - ***ellenoriz():*** ellenőrzi, hogy a generált tömbben lévő számok közül hány szám egyezik meg a játékos tippjeivel
  - ***kiir()***: kiírja az eredményeket, de itt jelenjen meg az is, hogy hány sorsolás volt eddig. A tippek és a sorsolt számok rendezett formában jelenjenek meg.

A részfeladat megoldása:

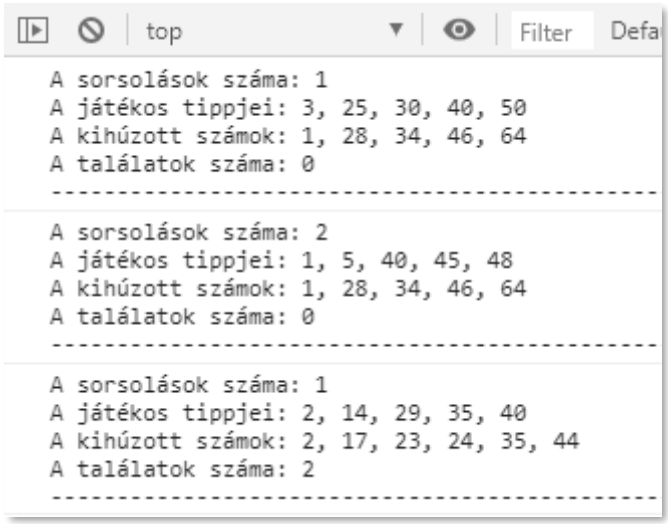
```
function lotto(huzas, min, max, tipp) {
    this.huzas = huzas;
    this.min = min;
    this.max = max;
    this.tipp = tipp;
    this.gepi = [];
    this.talalat = 0;
    this.sorsolasok = 0;
    this.general = function() {
        while(this.gepi.length < this.huzas) {
            let gen = Math.floor(Math.random() *
            (this.max - this.min + 1)) + this.min;
            if(this.gepi.indexOf(gen) === -1) this.gepi.push(gen);
        }
        this.sorsolasok++;
    };
    this.ellenoriz = function() {
        for(let i=0;i<this.gepi.length;i++) {
            if(this.gepi.indexOf(tipp[i]) !== -1) this.talalat++;
        }
    };
    this.kiir = function() {
        console.log("A sorsolások száma: " + this.sorsolasok +
        "\nA játékos tippjei: " +
        this.tipp.sort(function(a,b) {return a-b;}).join(", ") +
        "\nA kihúzott számok: "
        + this.gepi.sort(function(a,b) {return a-b;}).join(", ") +
        "\nA találatok száma: " + this.talalat +
        "\n-----------------------------------------------------");
    }
}
```

- Hozzunk létre egy objektumot, majd hajtsuk végre a következő feladatokat:
  - 5-ös lottó legyen, 5 számot húzzunk ki 1 és 90 között
  - hajtsuk végre a sorsolást
  - írassuk ki az eredményt
  - adjunk meg új tippeket, és ismét sorsoltassunk a fentiek alapján
- Hozzunk létre egy másik objektumot:
  - 6-os lottó legyen, 6 számot húzzunk ki 1 és 45 között
  - hajtsuk végre a sorsolást, majd módosítsuk a játékos tipp tömb elemeit
  - sorsoljunk ismét, majd írassuk ki a sorsolások számát

A részfeladat megoldása:

```
let otos = new lotto(5, 1, 90, [3, 25, 30, 40, 50]);

// 1. sorsolás
otos.general();
otos.ellenoriz();
otos.kiir();

// 2. sorsolás
otos.tipp = [5, 1, 45, 40, 48];
otos.general();
otos.ellenoriz();
otos.kiir();

// létrehozom a 6-os lottó objektumát
let hatos = new lotto(6, 1, 45, [2, 14, 29, 35, 40]);

// 1. sorsolás
hatos.general();
hatos.ellenoriz();
hatos.kiir();
```

A végeredmény pedig a következő:



4.3. Újabb fájlkezelő prototípus

Készítsünk egy újabb fajlok prototípust a következő adatokkal:

- Tulajdonságok:
  - ***fajlok:*** tömb, melyben fájlnevek szerepelnek kiterjesztéssel (kívülről kapja meg)
- Metódusok:
  - ***nevHossz()***: vár egy fájlnevet, és kiírja hány karakter hosszú, írja ki ha nincs ilyen fájl
  - ***csakNev():*** vár egy fájlnevet és levágja a kiterjesztését, tehát csak a fájl nevét írja ki, írja ki ha nincs ilyen fájl
  - ***rendez()***: a teljes fájlokat tartalmazó tömböt rendezi név szerint
  - ***tipus()***: vár egy fájlnevet és az adott fájlnév alapján levágja a kiterjesztést, és meghatározza, hogy milyen típusú fájlról van szó (docx, exe, mp3, avi legyen), a függvény kezelje azt a lehetőséget is, ha ismeretlen a fájl kiterjesztése

A részfeladat megoldása:

```
function fajlok(fajlok) {
    this.fajlok = fajlok;
    this.nevHossz = function(fajl) {
        if(this.fajlok.indexOf(fajl) !== -1)
            console.log("A fájl (" + fajl + ") " + fajl.length +
            " karakter hosszú.");
        else console.log("Nem található a fájl!");
    };
    this.csakNev = function(fajl) {
        if(this.fajlok.indexOf(fajl) !== -1)
            console.log(fajl.substr(0,fajl.lastIndexOf(".")));
        else
            console.log("Nem található a fájl!");
    };
    this.rendez = function() {
        this.fajlok.sort();
    };
    this.tipus = function(fajl) {
        if(this.fajlok.indexOf(fajl) !== -1) {
            let kiterj = fajl.substr(fajl.lastIndexOf(".")+1);
            let tipus = "";

            switch(kiterj) {
                case "docx" : tipus = "szöveges"; break;
                case "exe" : tipus = "futtatható"; break;
                case "mp3" : tipus = "zene"; break;
                case "avi" : tipus = "videó"; break;
                default : tipus = "ismeretlen";
            }
            console.log("A fájl (" + fajl + ") típusa: " + tipus);
        } else console.log("Ismeretlen fájltípus!");
    }
}
```

A fenti feladatban nem használtunk ismeretlen függvényeket. Az első függvényben a tömbben az indexOf() segítségével a legegyszerűbb kideríteni, hogy van-e olyan nevű fájl, ami érkezett paraméterként.

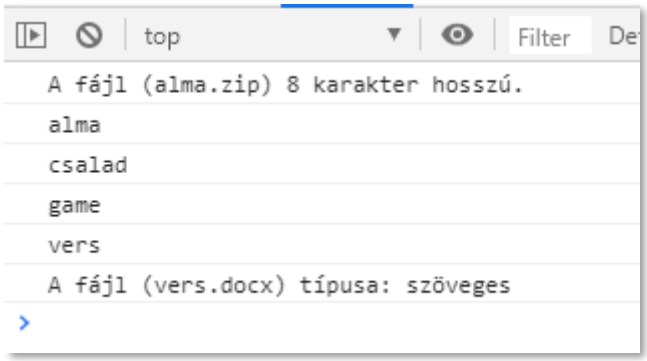
A fájl típusának meghatározásánál egy switch szerkezettel vizsgáljuk a megadott kiterjesztés típusát.

- Hozzunk létre egy objektumot, majd hajtsuk végre a következő feladatokat:
    - Kérdezzük le az egyik fájl karaktereinek hosszát
    - Rendezzük a fájlokat név szerint és írjuk ki őket
    - Írjunk ciklust, ami meghívja minden fájlra a csakNev() függvényt
    - Teszteljük a fájltípust egy szöveges (docx) típusú fájllal

A részfeladat megoldása:

```
let TCM = new fajlok(["vers.docx", "alma.zip", "csalad.jpg", "game.exe"]);
TCM.nevHossz(TCM.fajlok[1]);
TCM.rendez();

for(let i=0;i<TCM.fajlok.length;i++) {
    TCM.csakNev(TCM.fajlok[i]);
}

TCM.tipus(TCM.fajlok[3]);
```

A kimenet pedig a következő:



4.4. Objektumorientált kisállat és gazdi

Készítsünk egy Kisallat osztályt, melynek van egy konstruktora. A konstruktor a következő paramétereket várja:

- A kisállat típusa (pl.: kutya, macska stb.), életkora, fajtája, illetve a hang, amit ki tud adni
- Legyen egy beszel() metódusa, amit meghívva hangot ad ki.
- Készítsünk egy tevekenyseg() gettert, ahol dátumkezelő függvények segítségével meghatározzuk hogy most épp hány óra van. Ha 9 és 20 óra között van, akkor írja ki hogy játszik a kisállat, egyéb időben aludjon.
- Az osztálynak legyen egy tulajdonos() settere, amivel meg lehet adni a kisállat tulajdonosát. Fontos, hogy a setter neve nem lehet osztályváltozó is egyben, erre figyeljünk!
- Legyen egy gettere is a tulajdonos()-nak, ami egyszerűen visszaadja a tulajdonos nevét.

A részfeladat megoldása:

```
class Kisallat {
    constructor(kisallat, kor, fajta, hang) {
        this.kisallat = kisallat;
        this.kor = kor;
        this.fajta = fajta;
        this.hang = hang;
    }

    beszel() {
        console.log(this.hang);
    }

    get tevekenyseg() {
        const ma = new Date();
        const ora = ma.getHours();
        return (ora > 8 && ora <= 20 ? 'játszik' : 'alszik');
    }
    set tulajdonos(gazdi) {
        this.gazdi = gazdi;
    }
    get tulajdonos() {
        return this.gazdi;
    }
}
```

- Példányosítsuk a Kisallat osztályt a következők szerint:
    - A példányosítást a HTML dokumentumban végezzük el.
    - Legyen egy buksi objektumunk, akinek a neve Buksi, 1 éves, tacskó és a kiadott hangja: "Yip, yip!"
    - Készítsünk egy cirmi objektumot is, a neve legyen Cirmi, 2 éves, cirmos és a kiadott hangja: "Meoooowww!"
    - Beszéltessük mindkét kisállatot.
    - Írjuk ki a macska tevékenységét.
    - Állítsuk be a kutyának a saját keresztnevünket gazdiként, majd kérdezzük le a gazdit.

A részfeladat megoldása:

```
const buksi = new Kisallat('Buksi', 1, 'tacskó', 'Yip, yip!');
const cirmi = new Kisallat('Cirmi', 2, 'cirmos', 'Meooow!');

buksi.beszel();
cirmi.beszel();

console.log(cirmi.tevekenyseg);

buksi.tulajdonos = 'Attila';
console.log(buksi.tulajdonos);
```

A javascript nem vizsgálj egy változó értékét, így gazdinak nem csak string értéket adhatunk át, hanem objektumot is. Készítsünk egy új Gazdi osztályt a következők szerint:

- Legyen egy konstruktora, ami a gazdi nevét és címét várja. Ezeket beállítja osztályváltozóként.

- Legyen egy telefonszam() gettere és settere ami szintén beállítja ezt az osztályváltozót. A telefonszámnál használjunk szabályos kifejezést, ami átalakít mindenféle formátumot számra, azaz ami nem számként érkezik be, azt eldobja.
- Ezután példányosítsunk egy új tulajdonost a következő adatokkal: Kornél, Szeged, Nyírfa u. 23.
- A tulajdonos legyen a kutya új gazdija, illetve adjuk meg a gazdi telefonszámát is.
- Végül írjuk ki a kutya gazdiját, mint objektumot a konzolra.

A részfeladat megoldása:

```
class Gazdi{
    constructor(nev, cim) {
        this.nev = nev;
        this.cim = cim;
    }
    set telefon(telszam) {
        this.telszam = telszam.replace(/[^0-9]/g, '');
    }
    get telefon() {
        return this.telszam;
    }
}
```

A HTML fájlban pedig az előző példányosítás után a gazdi példányosítása.

```
buksi.tulajdonos = new Gazdi('Kornél', 'Szeged, Nyírfa u. 23.');
buksi.tulajdonos.telefon = '(30) 123-4567'
console.log(buksi.tulajdonos);
```

4.5. Objektumorientált könyvtár

Készítsünk egy olvasó osztályt, aminek a konstruktora várja az olvasó nevét és e-mail címét. Egy olvasónál ezen túl tároljuk le, hogy milyen könyv van nála. A példa egyszerűsége kedvéért egy olvasó egyszerre egy könyvet kölcsönözhet ki. Minden olvasónak lesz egyenlege, ami a késedelmi díjat fogja számon tartani. Kezdetben egyik olvasó sem rendelkezik könyvvel (null), illetve a kezdőegyenlege nulla.

- Minden olvasónak legyen egy kolcsonoz() metódusa, ami vár egy könyv objektumot. A metódus beállítja az olvasó osztályváltozójához a kikölcsönzött könyv objektumot. Ugyanitt állítsuk be a könyv objektum olvaso tulajdonságánál, hogy az aktuális olvasó kölcsönöz ki, azaz a könyv objektumnál ott lesz az olvasó objektum, az olvasó objektumban pedig a könyv objektum. Végül állítsuk be szintén a könyv objektum kolcsonozve tulajdonságát igaz logikai értékre, ezzel jelezve hogy az adott könyv épp ki van kölcsönözve.
- Az osztály rendelkezzen egy visszahoz metódussal is, ami szintén egy könyv objektumot vár. Itt az olvasó konyv tulajdonságát állítsuk vissza üresre (null), továbbá a könyv objektumon belül az olvaso tulajdonságnál töröljük az olvasó objektumot, majd állítsuk be a könyv objektumban, hogy nincs kikölcsönözve.

Természetesen megoldhatjuk tömbök segítségével azt is, hogy egy olvasó több könyvet tudjon kölcsönözni, ebben az esetben be kell építenünk a könyvek keresésének a lehetőségét, illetve a tömbből történő objektum törlését és beszúrását is.

A részfeladat megoldása:

```
class Olvaso {
    constructor(nev, email) {
        this.nev = nev;
        this.email = email;
        this.konyv = null;
        this.egyenleg = 0;
    }
    kolcsonoz(konyv) {
        this.konyv = konyv;
        konyv.olvaso = this;
        konyv.kolcsonozve = true;
    }
    visszahoz(konyv) {
        this.konyv = null;
        konyv.olvaso = null;
        konyv.kolcsonozve = false;
    }
}
```

Most készítsünk egy Konyv osztályt, melynek a konstruktora várja a könyv példány címét, szerzőjét, ISBN számát, illetve hogy melyik évben került kiadásra a könyv. Ezen túlmenően egy könyvről tároljuk el az olvaso osztályváltozóban az olvasó objektumot, ha épp kikölcsönözte valai. Legyen egy lejar osztályváltozó is, ami a kölcsönzés lejáratát fogja majd dátumként tárolni. Végül pedig szintén legyen egy osztályváltozó, ami megadja, hogy épp ki van-e kölcsönözve az adott könyv.

Készítsünk egy gettert és egy settert az osztályhoz kolcsonozve() néven. A setter várjon egy logikai értéket, azaz hogy épp kikölcsönzik-e a könyvet vagy visszahozzák. Ez alapján fogjuk beállítani a kolcsonozve osztályváltozó logikai értékét is, azaz az érkezett paraméter alapján. Ha a könyvet épp kikölcsönzik, akkor készítsünk egy aktuális dátum objektumot, majd állítsuk előre a kikölcsönzés határidejét 14 nappal a mostani időponthoz képest a setDate() függvény segítségével. A kapott dátumot a lejar osztályváltozóban tároljuk el. Ha a könyvet visszahozták, töröljük a lejar osztályváltozóból a dátumot. A getter pedig egyszerűen adja vissza a kolcsonozve osztályváltozó értékét.

A részfeladat megoldása:

```
class Konyv {
    constructor(cim, szerzo, isbn, kiadva) {
        this.cim = cim;
        this.szerzo = szerzo;
        this.isbn = isbn;
        this.kiadva = kiadva;
        this.olvaso = null;
        this.lejar = null;
        this._kolcsonozve = false;
    }
    set kolcsonozve(kolcsonozve) {
        this._kolcsonozve = kolcsonozve;
        if(this._kolcsonozve) {
            const ujLejarat = new Date();
            ujLejarat.setDate(ujLejarat.getDate() + 14);
            this.lejar = ujLejarat;
        } else
            this.lejar = null;
    }
    get kolcsonozve() {
        return this._kolcsonozve;
    }
}
```

Az utolsó osztályunk egy Konyvtar osztály legyen, melynek a konstruktora nem vár semmit. Tartalmazzon egy konyvek osztályváltozót, ami egy üres tömb lesz kezdetben, illetve egy olvasok tömböt, ami szintén üres kezdetben. Végül legyen egy napiBuntetes osztályváltozónk is, ami 50 forintra állítja be a büntetés napidíját.

- Legyen egy ujKonyv() metódusa az osztálynak, ami a kapott könyv objektumot beszúrja a könyveket tartalmazó tömbbe.
- Legyen egy ujOlvaso() metódusa is, ami az olvasó objektumot fogja beszúrni az olvasókat tartalmazó tömbbe.
- Végül legyen egy lejartKonyvek() metódus, amivel készítünk egy új dátum objektumot és megnézzük az olvasó objektumokat tartalmazó tömb alapján, hogy melyik olvasónak kell késedelmi díjat fizetnie. Ehhez használjuk a filter() beépített tömbkezelő függvényt. Hasonlítsuk össze a mostani dátumot a kint lévő könyvek lejárati dátumával, majd mentsük el ezeket az olvasókat egy objektumba. Végül egy for ciklussal menjünk végig a késett olvasókat tartalmazó objektumokon, és számoljuk ki, hogy hány napot késtek. Annyiszor szorozzuk rá a késedelmi díjat az osztályváltozóból.

A részfeladat megoldása:

```
class Konyvtar {
    constructor() {
        this.konyvek = [];
        this.olvasok = [];
        this.napiBuntetes = 50;
    }
    ujKonyv(konyv) {
        this.konyvek.push(konyv);
    }
    ujOlvaso(olvaso) {
        this.olvasok.push(olvaso);
    }
    lejartKonyvek() {
        const most = new Date();
        const kesettOlvasok = this.olvasok.filter(olvaso =>
        (olvaso.konyv !== null && olvaso.konyv.lejar < most));

        for(let olvaso of kesettOlvasok) {
            const datumKulonbseg = new Date(most - olvaso.konyv.lejar);
            const kesettNapok = datumKulonbseg.getDate();
            olvaso.egyenleg += this.napiBuntetes * kesettNapok;
        }
    }
}
```

Végül példányosítsuk az osztályokat a HTML fájlban a következők alapján:

- Legyen egy david nevű objektumunk a következő adatokkal: Dávid, david@gmail.com.
- Készítsünk egy metro nevű könyv objektumot a következőkkel: Metró 2033, a szerzője: Dmitry Glukhovsky, ISBN száma: 9789630791397, a kiadás éve pedig 2019.
- Hozzunk létre egy könyvtár példányt, majd állítsuk be a könyvtárnak az új könyvet és az olvasót.
- Dávid kölcsönözze ki a könyvet, majd kérdezzük le egy ternary operátor segítségével, hogy ki van-e kölcsönözve a metro c. könyv.
- Végül listázzuk ki a lejárt könyveket és írjuk ki Dávid adatait.

A részfeladat megoldása:

```
const david = new Olvaso("Dávid", "david@gmail.com");
const metro = new Konyv("Metró 2033", "Dmitry Glukhovsky",
"9789630791397", "2019");
const tik = new Konyvtar();

tik.ujKonyv(metro);
tik.ujOlvaso(david);
david.kolcsonoz(metro);

console.log("A metró ki van-e kölcsönözve: " +
(metro.kolcsonozve ? "Igen\nKi kölcsönözte ki: " +
metro.olvaso.nev : "Nem"));

tik.lejartKonyvek();
console.log(david);
```

4.6. Öröklés és statikus tagok

Készítsünk egy Jarmu osztályt. A konstruktora várjon egy rendszámot. A rendszám mellett állítsa be a a GPS-t bekapcsolt állapotra egy gpsBekapcsolva osztályváltozó segítségével (logikai változó).

- Az osztály rendelkezzen egy start() metódussal, ami kiírja, hogy a játművet elindítjuk.
- Legyen egy cegNev() statikus metódusa is, ami a "Jarmű ACME" szöveget fogja kiírni.

Készítsünk egy Auto osztályt, ami a Jarmu osztály leszármazottja. A konstruktora szintén várjon egy rendszámot, amit állítson be a szülőnek. A konstruktor a szülőtől kapott GPS beállító osztályváltozót írja felül, ugyanis az autókban nem lesz bekapcsolva a GPS.

- A leszármazott osztály írja felül a start() metódust, itt az autó indítása üzenet jelenjen meg.
- Végül írjuk felül a cegNev() statikus metódust is "Autó ACME" kiíratással.

A részfeladat megoldása:

```
class Jarmu {
    constructor(rendszam) {
        this.rendszam = rendszam;
        this.gpsBekapcsolva = true;
    }
    start() {
        console.log('A jármű indítása...');
    }
    static cegNev() {
        console.log('Jármű ACME');
    }
}

class Auto extends Jarmu {
    constructor(rendszam) {
        super(rendszam);
        this.gpsBekapcsolva = false;
    }
    start() {
        console.log('Az autó indítása...');
    }
    static cegNev() {
        console.log('Autó ACME');
    }
}
```

- Készítsünk egy auto és egy jarmu példányt, melyeknek adjunk át különböző rendszámokat.
- Írjuk ki az autó cégnevét, illetve hogy az autóban be van-e kapcsolva a GPS.
- Végül hívjuk meg az auto objektumra a start() metódust.

A részfeladat megoldása:

```
const auto = new Auto('ABC-123');
const jarmu = new Jarmu('XDD863');

console.log('Az autó cégneve:');
Auto.cegNev();
console.log("A GPS be van kapcsolva az autóban: " + auto.gpsBekapcsolva);
jarmu.start();
```

5.A jQuery keretrendszer

Eddig a natív javascript nyelvvel foglalkoztunk. Egy nyelvet viszont sajnos ma már nem elég önmagában ismerni, ehhez érdemes elsajátítanunk már valamilyen keretrendszert is. A keretrendszerek célja az, hogy felgyorsítsák a munkánkat, azaz kevesebb kóddal, egyszerűbb szintaktikával megoldhassuk mindazt, amit a natív nyelvben sokkal bonyolultabb lenne. Ebben a fejezetben éppen ezért a jQuery keretrendszerrel kapcsolatos gyakorlatokat fogunk végezni. Innentől kezdve már előre elkészített HTML és CSS kódokat tartalmazó fájlokban fogunk dolgozni, azaz manipulálni fogunk a DOM-ban, amit természetesen megtehetünk a natív nyelvben is, ám ott ez sokkal bonyolultabb és hosszabb lenne. A feladatokban használt forrásállományok és természetesen a megoldások is megtalálhatóak a példatár elején lévő github címen. A példákban a jQuery legújabb, 3.5.1-es verzióját használjuk.

5.1. A DOM módosítása eseményfigyelővel

A következő feladatokban különféle problémákra fogunk megoldásokat keresni a jQuery keretrendszer segítségével. A kapcsolódó fájlt megnyitva (5.1.html) a következőt látjuk:

Itt az a feladatunk, hogy beszúrjuk az utazáshoz kapcsolódó árat a gomb helyére. Ehhez vegyük szemügyre a HTML kódot, hogy a DOM-ba tudjunk új elemet beszúrni:

```
<h1>Vakációs csomagok</h1>
<ul id="boxes">
<li class="vacation">
  <h2>New York, New York</h2>
  <button>Egyedi ár</button>
  <p class="details">Ha szereted nyüzsgést.</p>
</li>
</ul>
```

Láthatjuk, hogy az alapvető felépítése egy számozatlan lista, melynek a listaelemei jelképezik az egyes utazásokhoz kapcsolódó dobozokat, amiből jelenleg csak egy van (később lesz több is). Egy ilyen utat jelképező dobozban van egy kettes szintű címsor, egy gomb, illetve egy bekezdés.

- Írjunk eseményfigyelőt, azaz ha megnyomjuk a gombot, kerüljön a gomb helyére az utazás ára.
- Az utazás árát érdemes felvenni egy külön változóba, az ár legyen egy bekezdésben (p elem). Ez a szöveg szerepeljen benne: Már $399.99-tól
- Beszúrást követően töröljük ki a gombot, azaz úgy szúrjunk be a DOM fába, hogy a gomb helyére kerüljön a bekezdés.

A feladat megoldása:

```
$(function() {

    $('button').on('click', function() {
        let price = $('<p>Már $399.99-tól</p>');

        $('.vacation').first().find('h2').after(price);
        $('button').remove();
    });

});
```

Az első példánál beemeltük a function()-t is, ami a jQuery rövidített eseményfigyelője, ami megegyezik a korábbi load típusú addEventListener() függvény működésével. Tehát itt ezzel jelezzük azt, hogy akkor kerüljön végrehajtásra a kód, amikor már betöltődött a HTML dokumentum. Ez a kódrész a további példákban nem lesz benne, de a forráskódokban szerepel és ajánlott a használata a saját kódunkban is. Természetesen a példatár elején szereplő defer attribútum továbbra is használható.

Az eseményfigyelőnél először megadjuk, hogy mi az esemény forrása, ami egy button típusú HTMl elem lesz. Megadjuk az esemény típusát: click, ami annyit jelent, hogy kattintáskor kerüljön végrehajtásra, vagyis ha rákattintunk egy button HTML elemre. Ez elég általános meghatározás a kódban, viszont a későbbiekben majd nézünk rá szigorúbb megfogalmazásokat is. Ebben az esetben ugyanis az oldalon bármennyi button HTML elem szerepelne, mindegyikre rákötnénk ezt az eseményt, ami egy bonyolultabb oldalon nem túl hatékony megoldás, viszont ebben az esetben ez tökéletesen működik.

Ezután deklarálunk egy változót, amibe jQuery objektumként beleírunk egy bekezdés elemet, melynek a tartalma megegyezik a feladatleírásban közölttel.

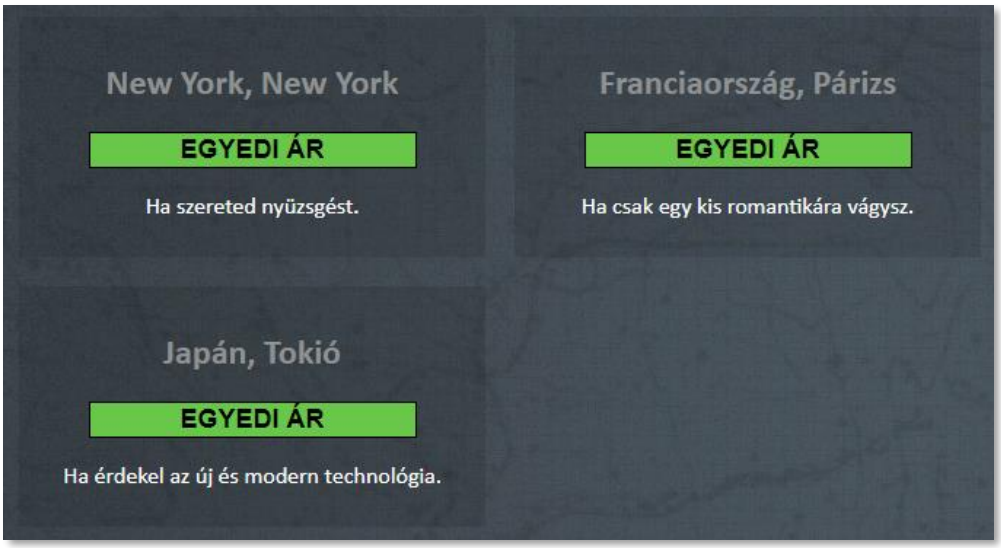
Ezután elindulunk a DOM fában, azaz a kiindulópontunk a vacation osztállyal rendelkező elemünk, azaz a listaelem első eleme (first() függvény). Ezután ebben a listaelemben megkeressük a gyermekek között a kettesszintű címsort (h2) a find() függvény segítségével. Majd a rákövetkező testvérelemre lépünk, azaz meghatározzuk az after() segítségével, hogy hová szeretnénk beszúrni a price változó értékét, azaz az újabb bekezdést.

Végül töröljük a gombot, ez szintén egy elég általános meghatározás. Ez ugyanis minden gomb HTML elemet töröl az oldalon, de a példánkban csak egy ilyen gomb szerepel, így ez szerencsére nem gond.

Végül az alábbi képen láthatjuk a végeredményt, illetve a forráskód megváltozását is:

5.2. Eseményfigyelő több vakációs csomagnál

Ebben a példában már több vakációs csomag is megjelenik, azaz a számozatlan listánkban immáron három különböző listaelem szerepel.



Itt már nem olyan egyszerű a dolog, hisz több gomb van, több vakációs csomag.

- A feladatunk az, hogy amelyik gombra rákattintunk, abba a listaelembe kerüljön beszúrva a nyaralás egyedi ára és az ahhoz kapcsolódó gomb kerüljön törlésre.
- A feladatot a $(this) segítségével hajtsuk végre, ami az esemény forrását jelenti, azaz a DOM-ban történő bejárás kiindulópontja lesz, ehhez képest kell relatívan beszúrnunk az árat.
- Ebben az esetben az eseményünk forrás a gomb, azaz a this maga.

A feladat megoldása:

```
$('button').on('click', function() {
    let price = $('<p>Már $399.99-tól</p>');

    $(this).after(price);
    $(this).remove();
});
```

A feladat megoldásában láthatjuk, hogy az eseményfigyelő forrása továbbra is egy button elem a HTML dokumentumban, azaz még mindig túl általános, de ez még mindig nem jelent gondot.

A $(this) itt tehát azt jelenti, hogy az esemény forrása, azaz amin kiváltottuk az eseményt, ez a kiindulópontja a relatív bejárásnak. Ez itt az a gomb, azaz button HTML elem lesz, amelyikre rákattint a felhasználó, így minden doboz esetében más és más. Ezzel tudjuk garantálni, hogy ugyanazt az eseményfigyelőt tudjuk rákötni több különböző HTML elemre is.

5.3. Az árak beszúrása testvérelemként

Kicsit csavarjunk a feladaton. A gomb és az utána lévő bekezdés minden csomagban bekerült egy DIV elembe, ez a vörössel jelölt terület:

- A feladatunk az, hogy az utazás árát jelenítsük meg a vörös terület mögött, de még továbbra is az utazás dobozában.
- Továbbra is töröljük ki a gombot a DOM-ból.

A feladat megoldása:

```
$('button').on('click', function() {
    let price = $('<p>Már $399.99-tól</p>');

    $(this).parents('.vacation').append(price);
    $(this).remove();
});
```

Itt egyetlen egy sor az ami újdonság, mégpedig a parents() függvényes. Itt az történik, hogy a gomb helyétől elindulunk, azaz az esemény forrásától. Felmegyünk annyi szülőt, amíg el nem jutunk a vacation osztályig, azaz a listaelemig. Ekkor meghívjuk az append() függvényt, ami annyit csinál, hogy a listaelem utolsó gyermekelemeként beszúrja a bekezdést, ami az utazás árát tartalmazza.

```
▼<li class="vacation">
   <h2>Franciaország, Párizs</h2>
  ▼<div>
     <p class="details">Ha csak egy kis romantikára vágysz.</p>
   </div>
   <p>Már $399.99-tól</p> == $0
 </li>
▼<li class="vacation">
   <h2>Japán, Tokió</h2>
  ▼<div>
     <button>Foglalás</button>
     <p class="details">Ha érdekel az új és modern technológia.</p>
   </div>
 </li>
```

A fenti feladat egyébként megoldható a következő kódok bármelyikével is:

- *$(this).closest('.vacation').append(price);*
- *$(this).parent().parent().append(price);*
- *$(this).parent().after(price);*

5.4. Különböző utak különböző árakon

Ebben a példában megoldjuk azt a problémát, ami eddig fennállt, azaz minden egyes utazásnak külön árat kellene meghatároznunk. Ha megnézzük a HTML kódban a listaelemeket, láthatjuk a következőt:

```
<li class="vacation" data-price="$399.99">
```

Azaz megjelent egy data-price attribútum. A data kezdetű attribútumokról azt kell tudnunk, hogy azért vezették be, hogy saját értékeket tárolhassunk a HTML dokumentumban úgy, hogy az ne jelenjen meg a felhasználóknál, de mégis kéznél legyen a fejlesztőknek, ha azzal bármiféle műveletet szeretnének végezni és nem külső forrásból akarják kiolvasni vagy épp eseményhez szeretnék kötni a kiolvasását és felhasználását. A data attribútum után kötőjellel bármilyen értéket megadhatunk, azaz mi döntjük el ennek az attribútumnak az elnevezését, a példában price-nak neveztük el.

A jQuery rendelkezik egy ehhez kapcsolódó beépített data() függvénnyel, ami paraméterként várja az egyedi elnevezését az ilyen attribútumoknak, amiből szeretnénk kiolvasni az értéket. Esetünkben tehát a price paramétert kell szöveges típusként átadnunk a függvénynek. Igen ám, de meg is kellene találnunk az attribútum helyét, tehát ismételten a DOM-ban a megfelelő bejárási függvényeket kell alkalmaznunk.

- Oldjuk meg, hogy az egyes utazások gombjaira kattintva jelenjen meg az utazás egyedi ára.
- A feladat megoldásához használjuk az egyes utazások listaelemeinél eltárolt data attribútumok értékeit.
- Az ár a gomb helyén jelenjen meg, azaz töröljük ki a gombot a DOM-ból, miután rákattintott a felhasználó és jelenítsük meg a helyén az egyedi árat.

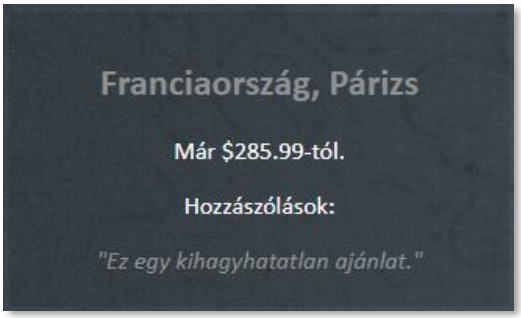
A feladat megoldása:

```
$('button').on('click', function() {
    let amount = $(this).closest('.vacation').data('price');
    let price = $('<p>Már $' + amount + '-tól.</p>');

    $(this).after(price);
    $(this).remove();
});
```

A megoldásban létrehoztunk két változót. Az első változóban megkeressük a gombhoz képest relatív módon a legközelebbi vacation listaelemet, mint szülőt és kiolvassuk a data attribútumából a hozzá kapcsolódó árat. Ezután a price változóba beszúrjuk ezt a kikeresett árat, mint változót.

Ezt követően a gomb mögé beszúrjuk egy after() függvénnyel az egyedi árat.

Végül töröljük a gomb elemet a remove()-al.



5.5. Eseményfigyelő abszolút eléréssel

A következő feladatban találunk az oldal tetején két újabb gombot:

Ebben a feladatban azt oldjuk meg, hogy mi van akkor, ha a DOM-ban történő bejárás és az esemény forrása teljesen különböző helyeken történik. Azaz ebben az esetben nem tudjuk a $(this)-t használni.

- Oldjuk meg, hogy az akciós utak gombra kattintva az onsale osztállyal ellátott listaelemeknél jelenjen meg egy új osztály is highlighted névvel. Ez sárga háttérrel fogja beszínezni az akciós utakat.
- Továbbá az is a feladatunk, hogy a lejáró ajánlatok gombra kattintva az expiring osztállyal rendelkező listaelemeknél jelenítsünk meg egy highlighted osztályt pluszban.
- Amikor megnyomjuk valamelyik gombot, előtte töröljük az összes listaelemről a meglévő kijelöléseket, azaz az összes highlighted osztályt, hogy ne keveredjenek a háttérszínek.
- Egy HTML elemhez újabb osztályt az addClass() függvénnyel tudunk hozzáadni, míg a removeClass() függvénnyel eltávolítani. A függvények paramétere ne tartalmazzon pontot!
- A megoldásban két különböző eseményfigyelőt kell írnunk a két gombra.

A feladat megoldása:

```
const vacation = $('.vacation');

$('#filters').on('click', '.onsale-filter', function() {
    vacation.removeClass('highlighted');
    vacation.filter('.onsale').addClass('highlighted');
});

$('#filters').on('click', '.expiring-filter', function() {
    vacation.removeClass('highlighted');
    vacation.filter('.expiring').addClass('highlighted');
});
```

Az első sorba konstansként létrehozzuk a vakációs listaelemeket tartalmazó objektumok tömbjét. Ez azért hasznos, mert a jQuery szempontjából hatékonyabb, ha változókban tároljuk el az egyes hivatkozásokat, ahelyett hogy újra kikeresnénk őket.

A két eseményfigyelő működése itt különbözik az eddig megszokottól. Érdemes úgy írni eseményfigyelőket, hogy kijelöljük az eseményfigyelő forrásának egy szülőelemét. Esetünkben ez egy DIV, ami a filters azonosítóval rendelkezik. Az on() függvényben pedig ezután jön az esemény típusa, és ezt követően határozzuk meg az esemény forrását, azaz az egyes gombokon lévő osztálynevekre hivatkozva tudjunk megkülönböztetni egymástól a két gombot. Itt már tehát konkrétabban történik az eseményfigyelő megfogalmazása.

Az eseményfigyelőkön belül először töröljük az összes listaelemről a highlighted osztályt, majd rákeresünk a kapcsolódó osztályokra a filter() függvénnyel és hozzáadjuk az új osztályt.

5.6. Rejtett elem animációs megjelenítése

Ebben a feladatban a HTML kódban van egy újabb számozatlan lista, ami az utazással kapcsolatos információkat tartalmazza, azaz hogy hová érkezik a repülő, mikor száll fel és mi a repülő száma:

```
<ul class='ticket'>
  <li><span>Érkezés:</span> John F. Kennedy IA</li>
  <li><span>Felszállás:</span> 2018. február 20. 09:30-kor</li>
  <li><span>Repülő száma:</span> 815</li>
</ul>
```

Ez a lista alapértelmezetten rejtve van. A feladatunk a következő:

- A gombra kattintva jelenítsük meg a repülés részleteit, azaz a teljes számozatlan listát.
- A lista slide jellegű animációval jelenjen meg.
- Ha ismét a gombra kattintunk, akkor a lista ismételten animálva tűnjön el.
- Készítsünk egy újabb eseményfigyelőt, ami akkor is megjeleníti vagy elrejti (állapottól függően) a listát, ha a felhasználó ráviszi a kurzort a H2-es elemre.

A feladat megoldása:

```
const vacation = $('.vacation');

vacation.on('click', 'button',showTicket);
vacation.on('mouseenter', 'h2', showTicket);

function showTicket() {
    $(this).closest('.vacation').find('.ticket').slideToggle();
}
```

Ismét létrehozunk egy konstanst, amiben eltároljuk a hivatkozást a vakációs listaelemekre, hogy ne kelljen többször meghívni feleslegesen.

Mivel két eseményfigyelőnk van és mindkét esemény ugyanazt az egy kódsort tartalmazná, ezért érdemes ilyenkor egy külön függvényt létrehozni nekik és hivatkozni a függvény nevére az eseményfigyelőn belül. Ilyenkor, ha nem anonim függvényt készítünk, hanem egy külső függvényre hivatkozunk, nem adhatjuk meg a függvénynek a kerekzárójeleit, azaz nem adható át paraméter. Ha paramétert szeretnénk átadni, akkor anonim függvényen belül kell meghívnunk a külső függvényt.

Érdekesség, hogy ebben az esetben is tudni fogja a külső függvény, hogy mi is az a $(this), azaz mi az eseményünknek a forrása. Nézzük meg mi a helyzet, ha szeretnék átadni egy saját paramétert:

```
vacation.on('click', 'button', function() {
    showTicket($(this));
});
vacation.on('mouseenter', 'h2', function() {
    showTicket($(this));
});

showTicket = (that) =>
that.closest('.vacation').find('.ticket').slideToggle();
```

Egy anonim függvényen belül történik a hívás, ám ez esetben a $(this)-t át kell adni, különben a külső függvény nem fogja tudni mit jelent. A showTicket() függvény ennél a példánál át van alakítva az új arrow syntax-ra, azaz mellőzhetjük segítségével a function kulcsszót és a kapcsos zárójeleket.

5.7. Billentyű események

Természetesen nem csak egér alapú eseményfigyelőket írhatunk, hanem figyelhetjük a lenyomott billentyűket is akár egyesével, vagy általánosságban azt a tényt, hogy valaki lenyomott, felengedett, vagy ténylegesen lenyomta és felengedte bármelyik billentyűt. Ezekre ugyanis külön-külön eseményfigyelők léteznek. A következő feladatban a jegyek számát adhatja meg a felhasználó:

- Írjunk eseményfigyelőt, ami az INPUT elem megváltozása esetén átírja a végösszeg értékét a jegyek számának megfelelően.
- Használjunk egy, a feladathoz jól illeszkedő billentyű eseménytípust.
- Olvassuk ki a listaelem data attribútumából a jegy árát, majd szorozzuk fel a beírt darabszámmal, hogy kiírjuk a végösszeget.
- Végezzük el a szükséges konvertálásokat, ugyanis minden bemenet szövegesként érkezik be.

A feladat megoldása:

```
$('.vacation').on('keyup','.quantity',function() {

    let price = +$(this).closest('.vacation').data('price');
    let quantity = +$(this).val();

    $(this).parents('.vacation').find('span').text(price * quantity);
});
```

A választott eseményfigyelő a keyup, ami a billentyű felengedésekor kerül aktiválásra. Itt persze más egyéb eseményfigyelők is működnek, ez csak egy a több lehetséges választás közül.

Először megkeressük a listaelemben lévő data attribútumot és kiolvassuk belőle a jegy árát. Az utasítás előtt lévő + jel a javascript hivatalos kasztoperátora, ha számmá szeretnénk konvertálni egy szöveges értéket. Itt természetesen használhatjuk a már korábban megismert parseInt() vagy parseFloat() függvényeket is.

Ezután kiolvassuk az INPUT mező értékét, azaz a felhasználó által beírt darabszámot. Űrlapelemből a val() függvény segítségével olvashatunk ki adatot, míg egyéb elemekből a text() segítségével szöveges elemként (a HTML kódok nem kerülnek beolvasásra), míg a html() függvénnyel a HTML kód is beolvasásra kerül. Ezekkel a függvényekkel nem csak olvasni, de írni is tudunk.

Végül megkeressük relatív módon a szülő listaelemet és visszafelé megyünk benne egy SPAN elemig. Ott kiírjuk a két számérték szorzatát, azaz a jegyek után fizetendő végösszeget.

5.8. Az alapértelmezett viselkedés elkerülése

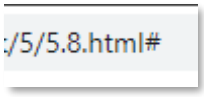
Vannak olyan HTML elemek, amelyek rendelkeznek egy ún. alapértelmezett viselkedéssel. Ezek azt a célt szolgálják, hogy ne kelljen külön eseményfigyelőt írniuk rájuk, mivel az esetek nagyrészében ezt az alapértelmezett viselkedést használjuk és ezt a fajta viselkedést várjuk el tőlük mind fejlesztőként, mind pedig felhasználóként. Mi is ez az alapértelmezett viselkedés?

Ha találkozunk egy hivatkozással egy weblapon, nagyon jól tudjuk, hogy ha rákattintunk, akkor az továbbít bennünket egy másik oldalra. Ahogyan azt is, hogy ha megnyomunk egy beküldő gombot egy űrlap alján, akkor az beküldi az adatainkat és a legtöbb esetben továbbít szintén egy új oldalra. Ezeket a mechanizmusokat nevezzük alapértelmezett viselkedésnek.

Igen ám, viszont fejlesztőként van, hogy szeretnénk másféle viselkedést megvalósítani, ahogyan ebben a feladatban is ilyen célunk van. Vannak hivatkozásaink, viszont én nem szeretnék új oldalt megnyitni, amikor rákattint a felhasználó, hanem azt szeretném, hogy jelenjen meg animációval az összes komment, ha a linkre kattint a felhasználó. Ehhez meg kell gátolnunk az alapértelmezett viselkedést.

Ilyen és ezen kívül még számtalan sok esetben tudjuk segítségül hívni az event objektumot. Ez egy különleges objektum, ami paraméterként érkezhet, de csak akkor ha mi szeretnénk használni. Azaz nem kell létrehoznunk, csak bírni paraméterként, amikor arra szükségünk van rá. Sosem küldjük, csak kapjuk. Nos, ennek az esemény objektumnak van egy ún. preventDefault() függvénye, ami meggátolja az ilyen alapértelmezett viselkedéseket.

- Oldjuk meg, hogy ha a felhasználó a hozzászólások mutatása linkre kattint, akkor ne kerüljön végrehajtásra a link alapértelmezett viselkedése, azaz a href attribútumba megjelenő # miatt ne ugorjon fel a lap a böngészőablak tetejére és ne jelenjen meg a címsorban egy # jel.



- Ezen túl animációval jelenítsük meg a hozzászólásokat az adott utazásnál.

A feladat megoldása:

```
$('.vacation').on('click','.expand',function(event) {
    event.preventDefault();
    $(this).closest('.vacation').find('.comment'). fadeToggle ();
});
```

Láthatjuk, hogy kívülről érkezik egy event paraméter jóformán a semmiből. Bárhogyan elnevezhetjük, de csak eseményfigyelőknél alkalmazható. A példában ezen az esemény objektumon keresztül hívtuk meg a preventDefault() függvényt, azaz meggátoltuk az anchor elem alapértelmezett viselkedését. Sőt, a címsorban nem jelent meg a kettőskereszt sem, ahogy az oldal sem ugrik fel a lap tetejére.

5.9. CSS kódok a jQuery-ben

Ebben a feladatban megnézzük hogyan alkalmazhatunk egy vagy több CSS utasítást az egyes HTML elemekre. Megnézünk többféle megoldást, melyek közül lesz szebb és kevésbé szép is.

- Az egyes listaelemekre történő kattintáskor állítsuk be a css() függvény segítségével azok háttérszínét a következő színkódra: #252b30
- Emellett adjunk keretet is a listaelemnek, ami legyen 1px vastag, folytonos vonalú a következő színkóddal: #967
- Jelenítsük meg az árakat az is utazásoknál, ezeket az egyes listaelemek tartalmazzák (.price).

A feladat megoldása:

```
$('#boxes').on('click','.vacation',function() {
    $(this).css('background-color', '#252b30');
    $(this).css('border', '1px solid #967');
});
```

Nos, természetesen ez nem túl szép megoldás, hiszen kétszer írtuk le ugyanazt a kódot csak amiatt, mert két CSS utasítást adunk meg a listaelemre. Képzeljük el milyen csúnya lenne a kód, ha 10-15 utasítást kellene így megadnunk. A következő példa már egy szebb megoldás:

```
$('#boxes').on('click','.vacation',function() {
    $(this).css('background-color', '#252b30')
    .css('border', '1px solid #967');
});
```

Viszont ez sem az igazi, mivel így is elég hosszú utasítást kapnánk 5-10 kódsor esetén. A következő megoldással már javascript objektumonként tudunk egyetlen függvényen belül megadni több tulajdonság-érték párt:

```
$('#boxes').on('click','.vacation',function() {
    $(this).css({'background-color': '#252b30',
    'border': '1px solid #967'});
});
```

Viszont a legszebb megoldás továbbra is az, ha a CSS kódba írjuk bele a CSS kódokat és egyszerűen csak osztályt rakunk arra az elemre, amit másképpen szeretnénk megjeleníteni:

```
$(this).toggleClass('selected');
```

A CSS fájl már tartalmazza a kapcsolódó kódokat, így azt nem kell beírnunk. A toggleClass() függvény segítségével viszont megtehetjük, hogy egyszer hozzáfűzi, majd legközelebb leveszi az osztályt az elemről. Mivel itt az eseményfigyelőnk kattintás alapú, így egy kattintásra rákerül a CSS osztály, a következő kattintáskor viszont törlésre kerül, így megszűnik a háttérszín és a keret az elemen.

A végleges kód, ami már megjeleníti az árakat is pedig a következő (kommentben egy csúnyább, de szintén működő megoldást láthatunk):

```
$('#boxes').on('click','.vacation',function() {
    $(this).toggleClass('selected');
    $(this).find('.price').show();

    //$(this).find('.price').css('display','block');
});
```

6. Összetett jQuery feladatok

Mivel a jQery számos újdonságot hoz a javascript natív nyelvébe, így fontos, hogy ne csak részleteiben, hanem összességében, összetett, bonyolultabb feladatok esetén is értsük a működését és képesek legyünk megoldani olyan feladatokat is, ahol egyszerre több eseményfigyelőt kell leprogramoznunk vagy bonyolultabb HTML elemeket kell beszúrnunk vagy épp áthelyeznünk a DOM-ba. Ezeknek a feladatoknak az a célja, hogy a hallgató megfelelő módon begyakorolja a keretrendszer működését ezzel felkészülve a zárthelyi dolgozatokra.

6.1. „A” összetett feladat

Adottak a már ismert vakációs csomagok, ám ezúttal mindegyiknél egy szövegmező szerepel külön-külön, alatta pedig hozzászólások. Ez egy leegyszerűsített adminisztrációs felület hozzászólási lehetőséggel.

A feladatunk a következő:

- Írjunk eseményfigyelőt, ami a felhasználó által beírt kommentet beszúrja a hozzászólások bekezdést követően egy számozatlan lista elembe. Ennek az elemnek adjunk egy comments nevű CSS osztálykijelölőt. A program az első komment megírásakor ellenőrizze le, hogy az adott vakációs listaelemben (.vacation) szerepel-e már ilyen comments nevű számozatlan listaelem (ul). Ha nem, akkor hozzuk létre és szúrjuk be a DOM-ba a megfelelő helyre. Fontos, hogy ezt az elemet csak egyszer hozzuk létre. Az eseményfigyelő focusout legyen, azaz ha kikattint a mezőből a felhasználó, akkor szúrjuk be a kommentet.
- A beírt kommentet LI elemként szúrjuk be a comments listába. Az új kommentek legalul jelenjenek meg. Minden komment tartalmazzon egy hivatkozást is egy [x] felirattal. Példa:

```
<li>Komment szövege <a class='delete' href='#'>[x]</a></li>
```

- Írjunk eseményfigyelőt, ami kitöröl egy teljes kommentet (azaz listaelemet LI), ha a felhasználó rákattint valamelyik komment végén lévő [x] hivatkozásra. Kapcsoljuk ki a link alapértelmezett viselkedését.

A kommentek beszúrását a DOM-ba megoldhatjuk egyetlen string-ként is, ahogy a price változó tartalmát készítettük el a feladatmegoldások során, de megoldhatjuk külön jQuery objektumokkal is. Pl.: `$('<a></a>').attr('hef', '#');` ami egy anchor elemet hoz létre egy href attribútummal.

Azt, hogy létezik-e egy vakációs listaelemben a comments osztálykijelölővel ellátott számozatlan lista egy elágazás segítségével ellenőrizhetjük. Kérdezzük le a létező listaelemek hosszát a DOM-ból.

A feladat megoldása:

```
$('.vacation').on('focusout','.myComm',function() {
    let vacation = $(this).closest('.vacation');

    let comment = $("<li></li>").text($(this).val() +
    ' ').append($('<a></a>').addClass('delete')
    .attr('href','#').text('[x]'));

    if(vacation.find('.comments').length == 0) {
        vacation.append($('<ul></ul>').addClass('comments'));
    }

    vacation.find('.comments').append(comment.hide().fadeIn());

    $(this).val("");
});

$('.vacation').on('click','.delete',function(event) {
    event.preventDefault();
    $(this).parent().fadeOut();
});
```

Az első eseményfigyelőt a myComm osztályú INPUT elemre írjuk, azaz ez lesz az esemény forrása. Az esemény típusa focusout, ami azt jelenti, hogy ha elveszíti a forrás a fókuszt, akkor kerül kiváltásra az esemény. Ez a gyakorlatban annyit jelent, hogy kikattint belőle a felhasználó, vagy a TAB billentyűvel ugrik a következő elemre.

Ezután egy comment változóba összerakjuk a feladatban bemutatott komment kódot. Először elkészítjük a listaelemet (li), majd ennek a belsejébe beszúrjuk az INPUT elemből kiolvasott komment szövegét. Utána hozzáfűzünk az append() függvénnyel egy anchor elemet, azaz hivatkozást készítünk a szöveg mögé. Ennek a hivatkozásnak adunk egy delete osztályt (a feladat példakódja alapján), illetve egy href attribútumot. Végül megadjuk a text() függvény segítségével a feliratot, amire a felhasználó rá tud majd kattintani.

Ezt követi egy elágazás, ami ellenőrzi, hogy a comments osztálykijelölők száma a vacation listaelemen belül nulla-e. Ha nulla, akkor tudjuk, hogy még nincs létrehozva az ehhez kapcsolódó UL elem, így létre kell hoznunk. Ezt közvetlenül be is szúrjuk a DOM fába és hozzáadjuk a megfelelő osztálynevet is.

Majd megkeressük ezt a comments listaelemet és append() segítségével utolsó gyermekelemként beszúrjuk a komment változó tartalmát. Ha ezt animációval szeretnénk beszúrni, akkor érdemes először a hide() függvénnyel elrejteni, majd a fadeIn() animáló függvénnyel megjeleníteni.

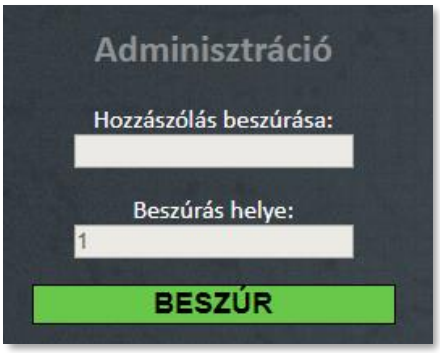
Végül pedig töröljük a szövegmező tartalmát úgy, hogy a val() függvény értékét üres string-re állítjuk.

A második eseményfigyelőt a kommentek törlésére írjuk. Azaz a delete nevű osztálykijelölő elemre történő kattintáskor váltjuk ki. Itt két dolog fog történni:

- Meggátoljuk az alapértelmezett viselkedését a hivatkozásoknak
- Animációs függvény segítségével töröljük a kommentet a DOM-ból, azaz nem elég itt csupán a $(this), mert az csak a hivatkozás anchor elemét jelenti, hanem fel kell menni a szülőhöz, a listaelemhez és a komment egy LI elemét kell kitörölnünk.

6.2. „B” összetett feladat

Ez a feladat hasonló lesz, mint az előző, ám itt külön helyen történik a hozzászólás beszúrása, azaz ehhez van egy külön űrlap:



Alapértelmezetten az input elemek értéke disabled állapotban van, azaz nem lehet őket szerkeszteni. A feladatunk az, hogy a beszúr gombra kattintva beszúrjuk a beírt hozzászólást, illetve megadjuk az azonosítóját a megfelelő utazásnak. Itt két azonosító létezik: 1 és 2. Valósítsuk meg a következőket:

- Írjunk eseményfigyelőt arra az esetre, ha a felhasználó ráviszi a kurzort valamelyik szövegmezőre, automatikusan váljon szerkeszthetővé, azaz a disabled állapot kapcsoljon ki (egéresemény).
- Írjunk egy másik eseményfigyelőt, amit a beszúr gombra kattintáskor kerül végrehajtásra. Ekkor olvassuk be a két input mezőből az adatokat az azonosítókijelölő segítségével (id).
    - A gomb megnyomásakor ha üres a hozzászólás mező, írjuk ki alert() segítségével, hogy "A hozzászólás nem küldhető be!".
    - Ha a második input mezőben kiválasztott azonosító nem 1 vagy 2, akkor írjuk ki szintén alert() segítségével, hogy "A beszúrás helye nem megfelelő!".
    - Állítsuk össze a következő kódot, amiben a komment fog szerepelni, majd szúrjuk be a megfelelő vacation osztálykijelölővel ellátott számozatlan listán belüli comments osztálykijelölővel rendelkező listába. Az új hozzászólás első gyermekelemként kerüljön beszúrásra.
    - Ne felejtsük el beszúrást követően törölni a szövegmező tartalmát, hogy ne a felhasználónak kelljen.
    - Az összeállítandó kód:

```
<li><span>A hozzászólás szövege.</span> <a class="del" href="#">[d]</a></li>
```

- Írjunk egy újabb eseményfigyelőt, ami a del osztálykijelölővel rendelkező hivatkozásokra kattintva kerül végrehajtásra és kitörli a kommentet, azaz a listaelemet a benne lévő teljes tartalommal. Kezeljük a hivatkozás alapértelmezett viselkedését is.

A feladat megoldása:

```
$('.vacation').on('mouseenter','input',function() {
    $(this).attr('disabled',false);
});

$('.vacation').on('click','button',function() {
    let opinion = $('#opinion');
    let id = $('#id').val();

    if(opinion.val() != "") {
        if((id > 0 && id < 3)) {
            let span = $("<span></span>").text(opinion.val() + " ");
            let a = $("<a></a>").addClass("del")
            .attr("href","#").text("[d]");

            let li = $("<li></li>").append(span).append(a);

            $('.vacation').eq(id).find('.comments').prepend(li);
            opinion.val('');
        } else alert("A beszúrás helye nem megfelelő!");
    } else alert("Üres hozzászólás nem küldhető be!");
});

$('.vacation').on('click','.del',function(e) {
    e.preventDefault();
    $(this).parent().remove();
});
```

Az első eseményfigyelőben az attr() függvény segítségével kapcsoljuk ki a szövegmezőkből a disabled értéket, azaz így tehetjük őket szerkeszthetővé.

Elágazások segítségével ellenőrizzük, hogy üresen akar-e valaki kommentet beküldeni, illetve hogy a megfelelő azonosítójú utazáshoz szeretnénk-e beszúrni. Az azonosítót kiolvassuk az input elemből, majd az eq() függvénynek adjuk át ezt az id változó tartalmat. A jQuery-ben az eq() függvény segítségével tudjuk meghatározni egy, a DOM-ban lévő HTML elem sorszámát, azaz ha több van belőle, akkor az 1. index jelenti a fában az első előfordulását, a 2. a másodikat. Nekünk itt pont erre van szükségünk.

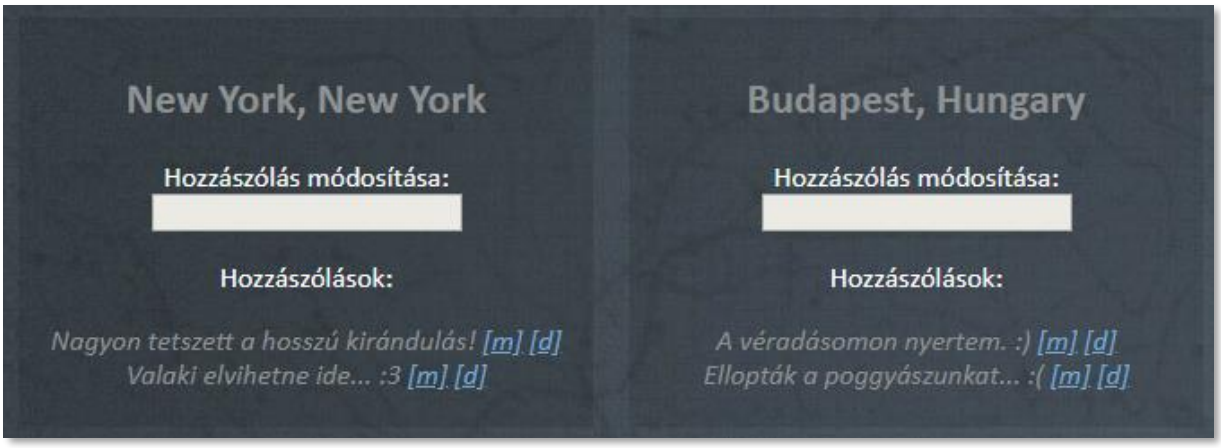
Végül a törlést megvalósító eseményfigyelő pedig szinte teljesen megegyezik az előző feladatban látott eseményfigyelővel.

A DOM-ba beszúrt új hozzászólás így néz ki a kódban:

```
▼<li class="vacation">
   <h2>Budapest, Hungary</h2>
   <p>Hozzászólások:</p>
  ▼<ul class="comments">
    ▼<li>
       <span>2-es azonosítóval ellátott hozzászólás... </span> == $0
       <a class="del" href="#">[d]</a>
     </li>
```

6.3. „C” összetett feladat

Az utolsó feladatban szintén adminisztrátorként tevékenykedünk, ám ezúttal a kommentek szerkesztésének a megoldása lesz a feladatunk.



Vegyük észre, hogy minden hozzászólásnak egyedi azonosítója van a HTML kódban:

```
<ul class="comments">
    <li data-id="6">A véradásomon (...)
    <li data-id="11">Ellopták a (...)
</ul>
```

Ezek az azonosítók fognak abban segíteni nekünk, hogy szerkesztéskor a megfelelő helyre írjuk vissza az átszerkesztett hozzászólást. Végezzük el a következő feladatokat:

- Írjunk eseményfigyelőt, ha valaki rákattint az [m] feliratú hivatkozásra.
    - Gátoljuk meg az alapértelmezett viselkedését a hivatkozásnak.
    - Olvassuk be a hozzászólást, illetve a hozzá kapcsolódó egyedi azonosítót, hogy majd vissza tudjuk írni a megfelelő helyre.
    - Szüntessük meg a disabled állapotát az input mezőnek a szerkesztés idejére.
    - Másoljuk be a hozzászólás szövegét a szerkesztőmezőbe úgy, hogy a hozzá kapcsolódó hivatkozás ne jelenjen meg, csak maga a hozzászólás szövege.
- Írjunk egy újabb eseményfigyelőt, ami visszaírja a kommentet a megfelelő helyre, ha a felhasználó kikattint az input mezőből (focusout).
    - Ekkor állítsuk vissza disabled állapotra az input mezőt.
    - Ne írjuk vissza a kommentet a megfelelő helyre, ha üresen próbálja visszaírni a felhasználó. Ilyenkor állítsuk az input mező hátterét pirosra jelezve, hogy így nem küldheti be és ne írjuk felül az eredeti hozzászólást. Oldjuk meg, hogy ha ezután ír bele szöveget és visszaírja, akkor a piros háttér változzon fehér színűre (ezt másik eseményfigyelőben is megadhatjuk).
    - Nézzünk utána a jQuery each() függvény működésének és próbáljuk meg ennek segítségével végig iterálni a megfelelő listaelemeket ellenőrizve az egyedi azonosítót, hogy megtaláljuk a megfelelő kommentet, ahová vissza szeretnénk írni a szöveget.
    - A szöveg mögé hozzunk létre egy hivatkozást is, ami pont úgy néz ki, mint az eredeti kommentekben a HTML forráskódban. Visszaíráskor töröljük az input tartalmát.
- Végül írjunk egy eseményfigyelőt, amely törli az egyes kommenteket a megfelelő hivatkozásra kattintva. Itt is figyeljünk az alapértelmezett viselkedés megszüntetésére.

A feladat megoldása:

```
let id;

$('.vacation').on('click', '.modify', function(e) {
  e.preventDefault();
  let li = $(this).parent();
  let input = li.parents('.vacation').find('input');
  id = li.data('id');
  input.css('background-color', 'white');

  input.attr('disabled', false);
  input.val(li.text().substring(0,li.text().length-8));
});

$('.vacation').on('focusout', 'input', function() {
  let that = $(this);
  that.attr('disabled', true);
  let vacation = that.parents('.vacation');

  if(that.val().length !== 0) {
    vacation.find('.comments').find('li').each(function() {
      if(id === $(this).data('id')) {
        $(this).html(that.val() + " <a class='modify' href='#'>[m]</a>
        <a class='del' href='#'>[d]</a>");
      }
    });
    that.val('');
  } else {
    that.css('background-color', 'red');
  }
});

$('.vacation').on('click', '.del', function(e) {
  e.preventDefault();
  $(this).parent().remove();
});
```

A program elején készítünk egy globális id változót, mivel az egyik eseményfigyelő kiolvassa, a másik pedig megkeresi. Tehát ebben tároljuk le a komment egyedi azonosítóját.

Az első eseményfigyelő akkor kerül végrehajtásra, amikor rákattintunk a módosítás hivatkozásra. Itt eltároljuk az id globálisba a kiolvasott data értéket a függvény segítségével.

Ezután beállítjuk az input mező hátterét CSS segítségével fehérre. Ez azért kell, mert ha a visszaíráskor véletlenül üresen küldené be a felhasználó és piros hátteret adtunk meg, akkor ilyenkor mindig visszaállítja majd a program alapértelmezett fehér színűre.

Ezután megszüntetjük a disabled állapotát a szerkesztőmezőnek.

Végül pedig beleírjuk a kommentet az input mezőbe úgy, hogy substring() segítségével levágjuk azt a részt, ami csak a kommentet tartalmazza. Ezt könnyen ki tudjuk számolni a HTML kódban lévő hivatkozás karakteriből, hisz mindegyik pont ugyanannyi karakterből áll.

A másik eseményfigyelőben az each() segítségével végig iteráljuk a megfelelő vacation listaelemen belül lévő comments listán belüli LI elemeket egyesével. Ha az adott listaelem azonosítója megegyezik a globális változónkban tárolt értékkel, akkor felülírjuk a tartalmát úgy, hogy ismét összeállítjuk a hivatkozásokat, amiket a komment után fogunk besúrni.

7.Az AJAX működése a jQuery-ben

A következő feladatokhoz egy webszerverre lesz szükségünk, ugyanis a mai modern böngészők szinte teljesen letiltják az AJAX technológiának a használatát a file:// protokoll útján. Ez a protokoll jelenik meg a böngészőben, amikor a saját gépünkről nyitunk meg egy HTML dokumentumot.

Ahhoz, hogy ezeket a feladatokat végre tudjuk hajtani vagy egy saját gépre telepített webszerverre lesz szükségünk pl.: XAMPP, vagy pedig egy tárhelyszolgáltatóra, ahová feltölthetjük a projekthez mellékelt PHP fájlokat. Ezeknek a PHP fájloknak a működése egyszerű, ám a gyakorlófeladatok megoldásához nem kell ismernünk a PHP programozási nyelvet, így nem is célja ennek a kiadványnak a nyelv vagy a mellékelt fájlok működésének a bemutatása.

Feltételezzük, hogy a hallgató tisztában van az AJAX működésével, fogalmával és fontosságával a webes világban, így ennek definiálására és példák segítségével történő elmagyarázására jelen kiadványunkban nem térünk ki. A kapcsolódó részleteket a kliensoldali webes programozás c. kurzus kapcsolódó óráin tárgyaljuk.

Az AJAX nem a jQuery találmánya, mindössze a jQuery az egyik olyan keretrendszer, ami nagymértékben leegyszerűsítette ezt a csodálatos technológiát. A natív javascript már nagyon régóta képes erre, viszont az ilyetén történő megvalósítása az AJAX-nak sokkal bonyolultabb és jóval hosszabb kódokat igényelnek, a bemutatásuk és a gyakorlati feladatok éppen ezért jQuery-n keresztül történnek bemutatásra.

7.1. Egyszerű AJAX hívás

Az első feladatban egyetlen utazást fogunk látni, ahol szeretnénk egy repülés részleteit lekérni, ám ezúttal nem a DOM-ból, hanem szerverről AJAX segítségével.

- Írjunk eseményfigyelőt, ami a gombra való kattintáskor AJAX hívás segítségével megjeleníti az ajax_1.php fájl kimenetét a HTML dokumentumban.
- Az AJAX hívás által visszaadott HTML kódokat szúrjuk be a ticket osztályú számozatlan listába gyermekelemekként. A PHP kód listaelemeket fog visszaadni string értékként:

```
<li><span>Érkezés:</span> London</li>
<li><span>Felszállás:</span> 2018. január 5.</li>
<li><span>Repülő száma:</span> 134</li>
```

- A szöveg slide típusú animációval jelenjen meg a gomb megnyomását követően.
- Az AJAX hívás csak egyszer történjen meg a gomb megnyomását követően, azaz ne lehessen egymás után többször behívni a HTML dokumentumba a PHP oldal által visszaadott listaelemeket. Ezt egy elágazás segítségével tudjuk vizsgálni.

A feladat megoldása:

```
$('.vacation').on('click', 'button', function() {
  let button = $(this);
  let vacation = button.closest('.vacation');
  let ticket = vacation.find('.ticket');

  if(ticket.children().length === 0) {
    $.ajax('ajax/ajax_1.php', {
      success: function(response) {
        ticket.html(response).slideDown();
      }
    });

    /*$.get('ajax/ajax_1.php', function(response) {
      ticket.html(response).slideDown();
    });*/
  }
});
```

Először kimentjük külön változókban az egyes jQuery hivatkozásokat, majd egy elágazás segítségével ellenőrizzük, hogy a ticket osztályú lista elemnek van-e gyermekeleme. Ha nincs, akkor megtörténik az AJAX hívás. Ezzel elkerülhetjük azt az esetet, hogy egymás után többször behívjuk ugyanazokat az értékeket.

Produkciós környezetben ez a megoldás akkor lehet problémás, ha arra számítunk, hogy az adatok bármikor frissülhetnek és szeretnénk a friss információkat megjeleníteni az oldal frissítése nélkül is. Ebben az minden alkalommal töröljük ki a számozatlan lista tartalmat, mielőtt lekérjük ismételten a szervertől az új listaelem információkat.

Az AJAX hívás során megadhatunk relatív, illetve abszolút elérési útvonalat is. Amennyiben távoli szerverről szeretnénk meghívni egy fájlt, a teljes webcím elérési útvonalát kell megadnunk. Ha a HTML dokumentum, illetve a JS fájl is ugyanott található, ahol az AJAX által hívott fájl, akkor használhatunk relatív elérési útvonalat is, ahogyan azt a példában is láthatjuk.

A kommentek között lévő AJAX hívás teljesen megfelel a fenti példának, csak ez egy rövidített változata. A GET típusú hívás akkor hasznos, ha csak fogadunk a szervernek és nem küldünk adatokat.

Az AJAX hívás tartalmaz egy objektumot, melyben szerepel a success tulajdonság. Ennek az értéke egy anonim függvény, ami egy response változóban tárolja a szerver által küldött információkat, amennyiben sikeres a hívás. Ennek a változónak a tartalmát jelenítjük meg a számozatlan lista belsejében a html() függvény segítségével.

Az alábbi ábrán láthatjuk, hogy sikeresen bekerült a DOM fába a hívás eredménye:

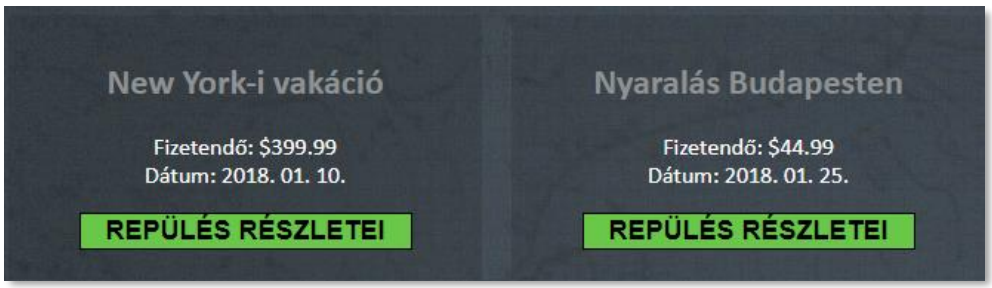
7.2. AJAX hívás GET paraméterrel

Az előző példában csak egy általános hívást csináltunk, azaz mindig ugyanazt a fájlt szólítottuk meg. Most kicsit dinamikusabban fogjuk megcsinálni az AJAX hívásunkat, azaz a PHP fájlt GET paraméterek segítségével fogjuk megszólítani. Ezáltal, hogy különböző értékeket adunk át paramétereken keresztül más-más eredményre számíthatunk.

Próbáljuk ki a böngészőben a következő két címet:

- http://localhost/jsbook/7/ajax/ajax_1.php?flightNum=1
- http://localhost/jsbook/7/ajax/ajax_1.php?flightNum=2

Láthatjuk, hogy a PHP által visszaadott eredmény más és más, azaz különböző repülőgépek adatait kapjuk eredményül. Ez azért hasznos nekünk, mert a következő példában két külön utazáshoz kell lehívnunk különböző adatokat és ezt a paraméterek segítségével tudjuk megtenni.



- Kiindulásképpen használhatjuk az előző kódunkat. Szintén lesz egy eseményfigyelőnk, ami a gomb megnyomásakor lekéri a repülés adatait a szervertől AJAX hívás segítségével.
- A megfelelő vacation listaelemekből olvassuk ki a data értékét, ami utazásonként különböző azonosítót tárol.
- A kiolvasott azonosító segítségével a fenti példában adjuk át a kiolvasott id értéket a webcímnek és hívjuk meg a megfelelő repülés adatait a szerint, hogy a felhasználó az 1-es, vagy a 2-es út gombjára kattintott-e rá.
- Az eredményt az előző feladat alapján slide animáció segítségével jelenítsük meg.

A feladat megoldása:

```
$('.vacation').on('click', 'button', function() {
  let button = $(this);
  let vacation = button.closest('.vacation');
  let ticket = vacation.find('.ticket');
  let id = vacation.data('id');

  if(ticket.children().length === 0) {
    $.ajax('ajax/ajax_1.php?flightNum=' + id, {
      success: function(response) {
        ticket.html(response).slideDown();
      }
    });
  }
});
```

A kódban ami új, az az id változó tartalma, ami kiolvassa a megfelelő nyaraláshoz kapcsolódó egyedi

azonosítót, illetve az AJAX hívásban lévő paraméteralapú hívás. Ezt a megoldást szebben is végre lehet hajtani, azaz nem csak a webcímen keresztül adhatjuk át a paramétert, hanem az objektumoknál is:

```
$.ajax({
    url: 'ajax/ajax_1.php',
    data: {
        "flightNum" : id
    },
    success: function(response) {
        ticket.html(response).slideDown();
    }
});
```

Talán a fenti megoldás áttekinthetőbb is, ha az URL címet nem első paraméterként helyezzük el, hanem azt is az objektumban kulcs-érték párként. A példában látszik, hogy a paramétert nem a PHP fájl kiterjesztését követően adtuk át, hanem külön egy data objektumon keresztül, ahol megadtuk a paraméter nevét és a kapcsolódó értéket beágyazott javascript objektumként. Az id itt továbbra is az előző kódban kiolvasott data érték.

Ha jól dolgoztunk, láthatjuk, hogy a két úthoz különböző repülőgép adatok kapcsolódnak:

7.3. Hibakezelés és várakozás

Az AJAX segítségével felkészülhetünk a nem várt eseményekre is, illetve tájékoztathatjuk a felhasználót is a töltés alatt különféle módon. Biztos találkoztunk már olyan weblapokkal, amikor megjelent egy körbe-körbe forgó kör és várnunk kellett a töltésre néhány másodpercet. Ilyet és ehhez hasonló várakozással kapcsolatos animációkat vagy kiírásokat mi is megjeleníthetünk a felhasználók számára. A következő PHP kódot ha lefuttatjuk a böngészőben, látni fogjuk hogy 3 másodpercig fog tölteni:

- http://localhost/jsbook/7/ajax/ajax_2.php

Nos, ez szándékosan van így megoldva, ezzel tudjuk majd demonstrálni a várakozást. Készítsük el a következő programot:

- Az előző példákat alapul véve írjunk olyan AJAX hívást, ami a success mellett tartalmazza az error tulajdonságot is. Ennek a segítségével tudjuk megjeleníteni az esetleges hibaüzeneteket. Itt három különböző paramétert várhatunk a függvénnyel, amiket bárhogyan elnevezhetünk. Írjuk ki ezeket a konzolra és nézzük majd meg hiba esetén mi történik.
- Adjuk hozzá a híváshoz a beforeSend tulajdonságot is, melyhez megint csak egy anonim függvény kapcsolódik. A beforeSend segítségével előkészíthetjük a várakozásra a webes alkalmazásunkat. Itt a gomb feliratát írjuk át "Töltés..."-re, illetve változtassuk a háttérszínét fehérre a css() függvénnyel. Továbbá állítsuk a gombot a töltés idejére disabled állapotra.
- Végül adjuk hozzá a complete tulajdonságot is az AJAX hívás objektumában, amihez szintén írjunk egy anonim függvényt. Ez a függvény akkor kerül végrehajtásra, ha befejeződött az AJAX hívás. Itt tudjuk visszaállítani a gomb feliratát arra, hogy "Repülés részletei", illetve a gomb színét a következő kódra: #69C84A. Végül vegyük le a disabled beállítást is a gombról.

A feladat megoldása:

```
$.ajax({
    url: 'ajax/ajax_2.php',
    success: function(response) {
        ticket.html(response).slideDown();
    },
    error: function(request, errorType, errorMessage) {
        console.log("Hiba típusa: " + errorType +
        "\nHibaüzenet: " + errorMessage);
    },
    beforeSend: function() {
        button.text('Töltés...');
        button.css('background-color', 'white');
        button.attr('disabled', true);
    },
    complete: function() {
        button.text('Repülés részletei');
        button.css('background-color', '#69C84A');
        button.attr('disabled', false);
    }
});
```

A fenti példában csak az AJAX hívást látjuk, az eseményfigyelő azonos az előző példában látottakkal, illetve az elágazás is szintén azonos. Kivéve, hogy itt nem használjuk az id változó tartalmát. Ha kipróbáljuk a kódot, akkor a következőt fogjuk látni három másodpercig:

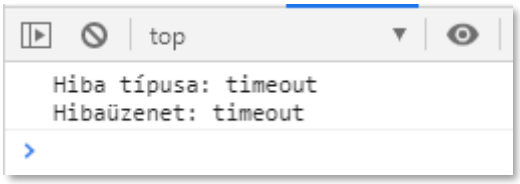
A beforeSend-hez rendelt függvény segítségével tényleg előkészíthetjük az alkalmazásunk egyes részeit a várakozás idejére. Itt akár egy várakozó animációt is megjeleníthetnénk, amit a jelenlegi jQuery és javascript tudásunkkal könnyedén meg tudunk tenni.

- A hívás teszteléséhez adjuk hozzá timeout tulajdonságot, aminek az értéke legyen 2000, azaz állítsuk be, hogy az AJAX hívásunk maximum 2 másodpercig hajlandó várakozni a válaszra. Mivel a PHP fájl 3 másodpercig fog tölteni, ezért timeout hibát fogunk tudni generálni ezzel a beállítással.

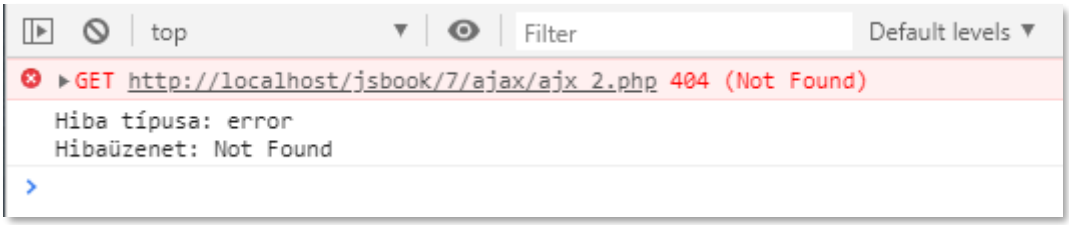
A feladat megoldása:

```
timeout: 2000
```

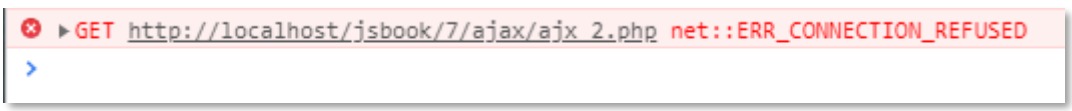
Az eredmény pedig a következő lesz:



Azaz a konzolra történő hiba kiírásából egyértelmű a hiba típusa. Most próbáljuk meg úgy lefuttatni az AJAX hívást, hogy szándékosan elrontjuk a hívás URL címét. Ekkor a következő hibát kapjuk:



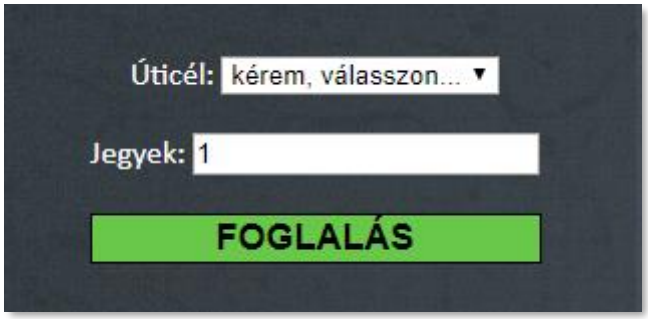
Azaz nem található a hívott fájl. Ha helyi szerveren dolgozunk, akkor állítsuk le a webszervert és ismét futtassuk le a kódot:



Ilyenkor nem is írt ki semmit a konzolra, mivel nem kaptunk vissza semmiféle választ a szervertől, csak egy hibaüzenet van, hogy a kapcsolat visszautasításra került.

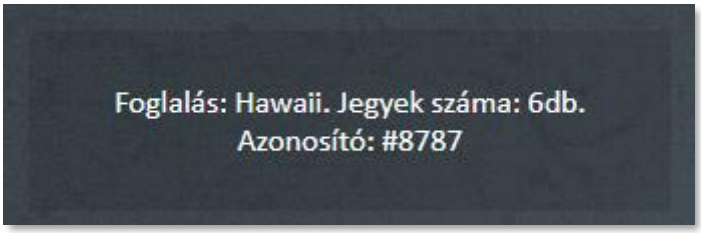
7.4. Űrlap beküldése AJAX segítségével

Ebben a példában egy űrlapot fogunk POST típusú kéréssel beküldeni. A GET típusú kérések arra valók, hogy információkat fogadjunk a szervertől, míg a POST típusúak segítségével küldeni szoktunk jellemzően adatokat a szerver felé.



A példában a felhasználó kiválaszt egy úticélt, majd megadja, hogy hány jegyet szeretne foglalni és megnyomja a foglalás gombot. Ekkor a szerverre felküldjük a felhasználó által megadott adatokat, majd a visszaküldött eredményt megjelenítjük a HTML dokumentumban. Ehhez a 3-as PHP fájlt fogjuk használni, amit ha meghívunk a böngészőben, hibát fog kiírni.

- Készítsünk egy eseményfigyelőt, ami a gomb nyomásakor végrehajtja az eseményfigyelőhöz kapcsolt függvényünket. Az űrlapok beküldése is tartalmaz egy alapértelmezett viselkedést. Ezt gátoljuk meg a tanult módon az event objektum segítségével.
- Íjunk egy AJAX hívást, melynél beállítjuk a type tulajdonság értékét POST típusra, ezzel jelezzük, hogy a szervernek szeretnénk adatot küldeni. A már ismert data tulajdonság segítségével egy beágyazott objektum segítségével megadjuk a kulcs-érték párokat, amiket szeretnénk felküldeni a szervernek.
- A PHP fájl a destination és a quantity értékeket várja. Ezeket olvassuk ki jQuery segítségével, majd írjunk tulajdonságneveket az itt megadott nevekkel.
- Adjunk egy success tulajdonságot is az AJAX híváshoz, ahol egy függvény segítségével paraméterül várjuk a szerver válaszát. A kapott kódot a vacation osztállyal rendelkező listaelemben jelenítsük meg a következőképpen:
    - Töröljük a listaelemben lévő teljes FORM elemet.
    - Ezután jelenítsük meg benne fadeIn() animációval a kapott információkat. A szerver bekezdésben küldi vissza az adatokat, így nincs szükség semmilyen befoglalóelemre.



A feladat megoldása:

```
$('.vacation').on('submit', 'form', function(e) {
  e.preventDefault();
  let form = $('form');
  let vacation = $(this).parent();

  $.ajax({
    url: 'ajax/ajax_3.php',
    type: 'POST',
    data: {
      "destination" : $('#destination').val(),
      "quantity" : $('#quantity').val()
    },
    success: function(response) {
      form.remove();
      vacation.hide().html(response).fadeIn();
    }
  });
});
```

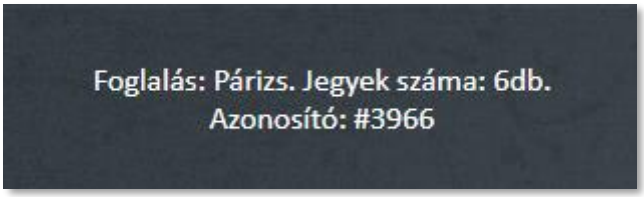
A data tulajdonsághoz kapcsolódó beágyazott objektumnál a kulcs-érték párok működése nagyon fontos. A kulcsok nevei mindig azok, amiket a PHP fájl vár. E kiadványnak nem témája a szerveroldali programozás, így ezt most külön megadtuk a feladatleírásban. A tulajdonsághoz kapcsolódó érték pedig mindig a HTML DOM-ból kiolvasott érték lesz, ami származhat egy űrlapmezőből vagy bármilyen más HTML elemből, de akár érkezhet más külső forrásokból is.

A fenti megoldás jól működik, viszont az a baj vele, hogy egy összetettebb, mondjuk 8-10 form elemet tartalmazó űrlap esetében elég körülményes lenne felsorolni minden mezőt és egyesével kiolvasni a kapcsolódó értékeket, bár ciklusok segítségével az is megoldható volna. Viszont szerencsére kitaláltak egy egyszerű függvényt a teljes űrlap beküldésére. A data tulajdonságnak egyszerűen adjuk meg a következő értéket:

```
data: form.serialize(),
```

Ezzel fogja a form változót (amiben a fenti kódban eltároltuk a DOM-ból az űrlapot), majd meghívja rá a serialize() függvényt. Ez automatikusan átadja a szervernek az űrlapelemek nevét és a mezőkben lévő értékeket, azaz kiolvassa a name és a value attribútumok értékeit.

Ez a megoldás viszont csak akkor működik, ha a PHP oldalon a várt űrlapnevek megegyeznek a HTML kódban lévő űrlapelemek name mezőiben megadott nevekkel.



A háttérben futó PHP kód egyébként azt csinálja, hogy a kiválasztott értékekhez generál egy véletlenszámot, amit visszaküld nekünk azonosítóként, illetve azokat az adatokat is, amiket előzőleg megadtunk az űrlapon keresztül.

7.5. A javascript fia, a JSON

A JSON egy ún. flat file, azaz egy olyan szabvány fájlformátum, mint amilyen az XML, a TXT vagy a CSV. Ezeket a formátumokat számos alkalmazás ismer és azért terjedtek el, mert viszonylag egyszerű a felépítésük és könnyen tudnak kommunikálni egymással ilyen formátumokban a különféle alkalmazások. Például egy PHP-t futtató szerver képes JSON formátumban leküldeni adatokat, aztán a javascript ezt képes átalakítani objektumokká, így könnyedén lehet majd hivatkozni rájuk.

A mi példánkban a PHP fájl által visszaküldött JSON a következőképpen néz ki:

```
{"destination":"New York","tickets":"6","id":"#5782"}
```

Láthatjuk, hogy a JSON valójában pont olyan, mint egy javascript objektum. De miért is jó nekünk ez a formátum, amikor eddig is boldogultunk a sima szövegekkel, mert tökéletesen be tudtuk szúrni a DOM-ba azt is. A válasz egyszerű: a JSON objektumok segítségével nem kell előre felépített HTML struktúrákat fogadni a szervertől, hanem itt csak nyers szöveges értékeket kapunk. Tehát itt mi döntjük el, hogy hogyan építjük fel a kapott adatok köré a HTML kódot és hogy szúrjuk be. Nem leszünk rákényszerítve arra, hogy csak egy UL elembe szúrhatjuk be, mert listaelemekben jöttek az adatok.

- Írjunk AJAX hívást, mely POST típusú kérést küld fel a szerver felé, azaz az előző példa alapján felküldi az űrlap adatait a serialize() függvény segítségével.
- Adjuk meg az adattípust a dataType tulajdonság segítségével json típusúnak, jelezve, hogy ilyen adatot várunk a szervertől.
- A success tulajdonsághoz írjunk anonim függvényt, mely fogadja a szervertől a JSON adatokat. Ebben a függvényben töröljük ki a form elemet, majd hozzunk létre egy számozatlan listát (UL). Ebbe a kapott értékeket listaelemenként tároljuk el és szúrjuk be az UL elemet az üres vacation listaelembe.
- A szervertől kapott változó egyes értékeire pont úgy tudunk hivatkozni, mint egy objektumra, azaz a pont operátor segítségével. A szervertől a következő JSON értékeket kapjuk meg: destination, tickets és id. Jó példa erre a fent látható JSON kód.

A feladat megoldása:

```
$.ajax({
    url: 'ajax/ajax_4.php',
    type: 'POST',
    data: form.serialize(),
    dataType: 'json',
    success: function(response) {
      form.remove();
      let ul = $('<ul></ul>');
      ul.append('<li>Foglalás: ' + response.destination + '</li>');
      ul.append('<li>Jegyek száma: ' + response.tickets + '</li>');
      ul.append('<li>Azonosító: ' + response.id + '</li>');
      vacation.hide().html(ul).fadeIn();
    }
});
```