TÖMBÖK VII.

Feladat

Készíts egy időjárás widgetet a hamburgerező weboldalának tetejére! A widget egy JavaScript fájlban eltárolt hőmérsékleti adatokból jeleníti meg a kiválasztott naphoz tartozó hőmérsékletet.

1. Készíts egy HTML szakaszt az **order-form.html** lap tetejére, amelyben egy legördülő lista, egy gomb és egy a hőmérséklet megjelenítésére alkalmas rész van! A legördülő listával a hét napjai közül lehessen választani. A napokhoz társított érték a nap sorszáma legyen 0-tól 6-ig! A szakaszban jelenítsd meg az "Időjárás" szöveget is!
2. Készíts egy JavaScript fájlt, amelyben egy globális tömb típusú változóban eltárolod a hét hőmérsékleti adatait!
3. Írj egy olyan függvényt, amely kiolvassa a legördülő listából a kiválasztott értéket, és ez alapján megjeleníti az adott napi hőmérsékletet! Ahhoz, hogy működjön, a gomb lenyomására futtasd le a függvényt! A gombban állítsd be az onclick attribútum értékének a függvényed nevét zárójelekkel! (Például ha a függvényed neve weatherWidget, akkor a HTML fájlba a gombba ez kerül: onclick="weatherWidget()".)
4. Formázd meg a widgetet, hogy megjelenésében illeszkedjen az oldalhoz!
5. Illeszd be az elkészült widgetet az összes oldal tetejére!

## Listák és Tömbök

Amikor a számítógépet kifejlesztették, nem játszottak rajta, nem rajzoltak vagy cseteltek, egész nap óriási listákat dolgoztak fel. A mai napig a programozás egyik alapvető feladata az adatokkal való munka. Az adathalmazokat a programok gyakran tömbökben tárolják, amelyek értékek listái.

## Array - tömbösítsünk!

Értékek indexelt halmaza. Ez a tömb. Úgy kell elképzelni, mint egy polc, ahová felpakoljuk a dolgokat.

Az index az a mutató, amivel eléred a tömb egyes elemeit. Tehát minden elemnek van egy száma a tömbön belül, amivel le tudod kérni az értékét, vagy módosítani tudod.

A tömbökbe szabadon elhelyezhetsz Number, String, Boolean, Object sőt akár Array típusú elemeket is. Ha tömbbe tömböt teszel, akkor azt **többdimenziós** tömbnek is nevezik.

## Dinamikus

JavaScript esetén a tömb dinamikus, ami azt jelenti, hogy bármikor vehetsz fel új elemet, vagy törölhetsz meglévőket. Ez nem minden programnyelv esetén van így, a legtöbbször előre meg kell adnod a tömböd hosszát. De szerencsére mi js-ben dolgozunk, úgyhogy ez legyen a Java-sok és a C#-osok gondja! :)

## Mixed

Másik különbség például a Java nyelvvel szemben, hogy lehet mixelt a tömb, azaz egy tömbön belül különböző típusú elemeket is tárolhatsz.

## Tömb létrehozása

Új tömböt legegyszerűbben a szögletes zárójelek használatával hozhatsz létre. Az alábbi példában egy három elemű tömböt definiálok:

Új tömböt legegyszerűbben a szögletes zárójelek használatával hozhatsz létre. Az alábbi példában egy három elemű tömböt definiálok:

## Hozzáférés a tömb elemeihez, az index

Amikor elemeket adsz a tömbhöz, a JavaScript értelmező automatikusan megszámozza azokat. JavaScriptben a tömb indexek csak számok lehetnek.

**FONTOS: az index mindig 0-val kezdődik!**  
Azaz a tömb első elemének indexe 0, és nem 1.

A példa azt mutatja meg, hogy egy tömb elemeit hogyan éred el és hogyan módosíthatod az indexük segítségével:

## Többdimenziós tömbök

Ezeknél egy tömböt egy másik tömbbe ágyaznak. Úgy képzeld el, mint egy Excel táblát, ahol a sorok a külső tömb elemei, a belső tömbök pedig a cellákat tartalmazzák.

Például:

var arr = [true, ['elso', 'masodik', 'harmadik'], 'Joe'];

A példában a tömb második eleme szintén egy tömb.  
A beágyazott tömb második elemének elérése:

arr[1][1]

## isArray(), tömb azonosítása

A munkád során előfordul, hogy meg kell állapítanod egy változóról, hogy Array típusú-e? A typeof utasítás a tömböket Object típusúnak ismeri fel, ezért nem alkalmas az azonosításukra. Erre a célra az Array objektum isArray metódusa szolgál. Ha megadsz neki egy változót, akkor megállapítja, hogy tömb-e?

A példában a tömb esetén true értéket kapsz vissza, ami azt jelenti hogy igaz - azaz tömbbel van dolgod. Ellenkező esetben az eredmény false - hamis - azaz nem tömböt adtál át az isArray metódusnak:

# **A tömbök metódusai**

# **Array metódusok**

A tömbök esetén számos metódus áll rendelkezésre az adatok kereséséhez, ellenőrzéséhez, a tömbök vágásához vagy egyesítéséhez. Most csak a legalapvetőbbeket mutatom meg, később a függvények megismerése után visszatérünk még a magasabb szintű metódusokra is.

## push()

Segítségével új elemet tudsz felvenni a tömb végére. Azaz az így felvett elem a tömböd utolsó eleme lesz. Fontos, hogy ez a metódus megváltoztatja az eredeti tömböt, azaz nem kell felüldefiniálnunk ahhoz, hogy a módosítások érvénybe lépjenek. Miután a metódus lefutott, a tömb új hosszát adja vissza.

**FONTOS: módosítja az eredeti tömböt.**

## pop()

A push()-tól eltérően nem hozzáad, hanem elvesz egy elemet a tömb végéről. A törölt elemet adja vissza, és szintén azonnal módosul az eredeti tömb.

**FONTOS: módosítja az eredeti tömböt.**

A példában a push() és a pop() működését is bemutatjuk:

## unshift()

A push()-hoz hasonlóan működik, csak a tömb elejére ad hozzá új elemet, azaz eggyel hátrébb tolja az elemeket. Hatására minden meglévő tömbelem indexe eggyel nőni fog.

**FONTOS: módosítja az eredeti tömböt.**

## shift()

Ez pedig a pop() párja, egy elemet kivesz a tömb elejéről, és a többit eggyel előre tolja. Hatására minden meglévő tömbelem indexe eggyel csökkenni fog.

**FONTOS: módosítja az eredeti tömböt.**

A példában az unshift() és a shift() működését is megmutatom neked:

# **A tömbök vágása és összeillesztése**

Tömbök vágása

A tömbök több ezer vagy millió elemmel is rendelkezhetnek. Hasznos ha hatékonyan tudod őket vágni, vagy adott helyre új elemeket tudsz beilleszteni. Most erről lesz szó.

slice()

A tömböt vágja. Két számot vár, start és end a nevük.

* **start:** azt adja meg, hogy hol kezdje a vágást. Ha 0, akkor a tömb elején kezdi, ha mínusz szám, akkor hátulról fogja számolni.
* **end:** azt adja meg hogy melyik elemnél fejezze be a vágást. Ugyanazok igazak rá, mint a startra.

**FONTOS: nem módosítja az eredeti tömböt, hanem egy új tömbbel tér vissza.**

splice()

A slice-szal ellentétben nem csak kivenni lehet elemeket vele, hanem hozzá is lehet adni a tömbhöz.

A szintaxisa a következő:

array.splice(index, howmany, item1, ....., itemX)

Paraméterei:

* **index:** kötelező, azt adja meg hogy honnan induljon a vágás, olyan mint a slice esetén.
* **howmany:** azt adja meg, hogy hány elemet akarsz eltávolítani.
* **item1, ..., itemX:** azok az elemek, amiket az eltávolított elemek helyére akarsz tenni. Nem kötelező megadni, akkor csak eltávolítás lesz, hozzáadás nem.

**FONTOS: nem módosítja az eredeti tömböt, hanem egy új tömbbel tér vissza.**

**Megjegyzés:**a splice művelet után a fruits tömb így fog kinézni: ["Banana","Orange","Lemon","Kiwi","Apple","Mango"]. Azért mert nem vettünk ki egy elemet sem és a 2 indexűtől (azaz a harmadiktól), "Lemon"-tól kezdődően beszúrtam két új elemet.

join()

A tömbből String-et készít, az elemeket vesszővel elválasztva fűzi egymás után.

**FONTOS: nem módosítja az eredeti tömböt, hanem egy új**String**-gel tér vissza.**

# **Paraméterek átadása**

Paraméterek átadása

Amikor paramétereket adsz át egy függvénynek, akkor érdekes dolgok történnek a háttérben, ezért nem árt tisztában lenni azzal, hogy a függvények mit kezdenek a nekik átadott adatokkal.

Primitívek átadása érték szerint

Kis ismétlés:

**Primitív változó típusok:** Boolean, Number, String, Undefined, Null.

Tehát a következő szabályok rájuk vonatkoznak.

**Paraméter átadás érték szerint:** a primitív típusokat másolja a js. Azaz, amikor átadsz egy ilyen változót a függvénynek, akkkor egy másolatot készít az eredetiről és azzal dolgozik.

function callByValue(varOne, varTwo) {

console.log("A függvényen belül");

varOne = 100;

varTwo = 200;

console.log("varOne =" + varOne +"varTwo =" +varTwo);

}

let varOne = 10;

let varTwo = 20;

console.log("A függvény hívása előtt");

console.log("varOne =" + varOne +"varTwo =" +varTwo);

callByValue(varOne, varTwo);

console.log("A függvény hívása után");

console.log("varOne =" + varOne +" varTwo =" +varTwo);

**Na, kitalálod mi lesz ennek a kimenete? Nem kell, én megmondom:**

* A függvény hívása előtt: varOne = 10, varTwo = 20
* A függvényen belül: varOne = 100, varTwo = 200
* A függvény hívása után: varOne = 10, varTwo = 20

**Miért ez a kimenet?**

* Létrehoztam a varOne és varTwo változókat 10 és 20 értékekkel.
* Amikor átadtam őket a függvénynek, akkor kiolvasta az értéküket, és ahogy a paraméterlistában megadtam callByValue(varOne, varTwo) azon a néven létrehozott két lokális változót a függvényben és az eredeti értékeket bemásolta ebbe az két új változóba. Amikor kiíratom a függvényben az értékeket, látszik hogy megváltoztak.
* Mégis amikor az eredeti értékeket újra kiíratom a függvényhívás után, azok változatlanok maradnak. Azért **mert nem az eredeti változókkal dolgozik a függvény, csak a másolatukkal**.

# **Objektumok átadása referencia szerint**

Paraméterek átadása

Amikor paramétereket adsz át egy függvénynek, akkor érdekes dolgok történnek a háttérben, ezért nem árt tisztában lenni azzal, hogy a függvények mit kezdenek a nekik átadott adatokkal.

Objektumok átadása referencia szerint

Mit nevezek itt objektumoknak? Azokat a változókat, amelyek nem primitív típusok. Itt elsősorban a már tanult tömbökre gondolok és lesznek még az Object típusok, amelyeket hamarosan megismersz.

Tehát a következő szabályok rájuk vonatkoznak.

**Paraméter átadás referencia szerint:** az objektumokat nem másolja a js. Minden változónak van egy címe a memóriában. Amikor egy objektumot kap a függvény, akkor létrejön egy új név a lokális váltoozónak, de a memóriacím ahova mutat, az ugyanaz lesz. Tehát két különböző néven ugyanazt a változót éred el.

function callByReference(arr) {

console.log("Érték a függvényen belül");

arr[0] = 100;

console.log(arr);

}

let varArray = [10];

console.log("Függvényhívás előtt");

console.log(varArray);

callByReference(varArray);

console.log("Függvényhívás után");

console.log(varArray);

**Na, kitalálod mi lesz ennek a kimenete? Nem kell, én megmondom:**

* Függvényhívás előtt: [10]
* Érték a függvényen belül: [100]
* Függvényhívás után: [100]

**Miért ez a kimenet?**

* Létrehoztam a varArray tömböt [10] értékkel.
* Amikor átadtam a függvénynek, szándékosan más nevet választottam a lokális változónak: arr, mivel az teljesen mindegy, hogy mi a neve a függvényen belül. A függvényben megváltoztattam a 0 indexű elemet és kiírtam. Látszik, hogy meg is változott.
* Most jön a meglepetés: amikor az eredeti tömböt újra kiíratom a függvényhívás után, is megváltozott. Azért **mert az eredeti változó refernciájával dolgozik, tehát ugyanazt a helyet módosítja a memóriában**.

**Előnyei:**

* A függvényre nézve globális objektumokat is könnyen tudsz módosítani.
* Takarékoskodik a memóriával, mivel nem foglal új memóriaterületet mint a másolásnál.

**Hátrányai:**

* Az átadott objektumokat akaratodon kívül módosíthatod.

Összefoglalva

Ez a különbség a referencia és az érték szerinti átadás között :)