WEB04-Javascript

Változatok:
JavaScript, ActionScript, TypeScript, ...
Feladata az **interaktivitás és viselkedés** leírása.

A Java és a JavaScript mindenben eltér. A JS:

- dinamikusan típusos,
- gyengén típusos
- forráskódból töltődik be, és
- prototípus alapú objektumokkal rendelkezik.

Nyelvi alapok

A <script> elemet mindig explicit záró címkével használjuk. Segítségével HTML fájlbe is kerülhet JS kód, bár nem célszerű.

Kommentek	`// egy soros; /* több soros */`
Aritmetikai operátorok	`+, -, /, *, %, ++, --`
Értékadás operátorok	`=, +=, -=, *=, /=, %=`
Bitenkénti operátorok	`&, |, ^, ~, <<, >>, >>>`
Logikai operátorok	`||, &&`
Összehasonlító operátorok	`==, ===, !=, !==, <, >, <=, >=`
Feltétel vizsgálat	`if..else, switch..case (break, default), instanceof, typeof, .. ? .. : ..`
Ciklusok	`for, for..in, while, do..while, break, continue`
Hibakezelés	`try..catch..finally, throw`
Objektumok kezelése	`new, delete`
Függvények	`function, return`

Definiált típusok:

`number, bigint, string, boolean, undefined, null, symbol, Object, Function`

- Ha a `var` kulcsszót elhagyjuk, akkor is létrejön a változó, de akkor a **globális névtérben** jön létere.

Létezik **strict mód**, melnyek bekapcsolásával a `var` elhagyása hibát eredményez.

Dinamikusan típusos: nem adunk meg típust.
Gyengén típusos: operátor működése változik a változóban tárolt érték típusától.
- Ha összeadunk két változót, előfordulhat, hogy az egyikben string a másikban pedig number szerepel.

Az == csak az értéket hasonlít össze, míg az === egyenlőség a típus infót is ellenőrzi.

❗ Ha egy változót értékadás nélkül hozunk létre, akkor az értéke `undefined` lesz, és a típusa is `undefined` lesz.
ℹ️ Az **undefined** típua **undefined**
❗ A **null** viszont egy **Object**

Konstanok

- `const` kulcsszó.
- Létrehozásakor az **értéket is meg kell adni**

Változók láthatósága

function scoping: A `var`-ral létrehozott változók az egész függvényen belül láthatóak
block scoping: A `let`-tel létrehozott változók viszont csak a blokkon belül láthatóak

Truthy, Falsey

Minden bool-lá alakítható!
turthy: `true, '0', 123 , 'valami', [], {}`
falsey: `false, 0, NaN, '', null, undefined`
Többféleképpen is konvertálhatunk bool-ba:

```
let b_obj = new Boolean(false); //objektum!!
let b = false; // primitív
let b_cast = Boolean(false); // bool kaszt
let b_cast2 = !!false; // bool kaszt
```

⚠ A new Boolean(false) egy objektumot hoz létre!
- A tagadása (bool-lá alakítása a `!!` operátorral) esetén azt nézi, hogy ez egy *nem üres objektum*.

Burkolás

Az egyszerű típusokat **objektumba lehet burkolni**. Nagybetűkkel kezdődnek.
- Ilyenkor ha utána bool-lá alakítjuk, akkor igaz, vagy hamis értéket kapunk, mert azt nézi, hogy üres-e az objektum.

⚠ Mivel minden változó az értékétől függetlenül bool-lá alakítható és mint tudjuk az `undefined` falsy, ezért azt, hogy egy változónak van-e értéke, egy egyszerű `if(valtozo)`-val tudjuk vizsgálni.

Ha egy változónak csak akkor szeretnénk értéket adni, ha még nincs neki, akkor azt az alábbi kódrészlettel tehetjük meg:

```
let afa;
let szokasosAfa = 27;
let afa = afa || szokasosAfa;
alert(afa); // 27
```

📌 Lényege az, hogy ha nincs megadva az `afa` nak semmi, akkor a `szokasosAfa` is kiértékelődik, és azt az értéket kapja így meg a változó.

Hasonló logikával tudjuk elérni azt, hogy kódrészleteket csak **feltételesen futtas**sunk. Ehhez a `x && alert('lefutott');` -ot használhatjuk.

Tömbök, függvények

Tömbök kétféle képpen definiálhatóak, mindkettő esetben **object** típusúként kapjuk vissza őket.

```
let napok = [ 'hétfő', 'kedd', 'szerda' ];
let evszakok = new Array('tavasz','nyár','ősz','tél');
alert( typeof napok ); // 'object'
alert( typeof evszakok ); // 'object'
```

Elem hozzáadása `array.push()`;
Kivenni az utolsó elemet a `array.pop()`-pal lehet.

`splice(index, howMany [, el1, ... elN])`

`index`: tömb módosítása ettől az indextől kezdve
`howMany`: mennyi elemet szeretnénk törölni

`el1...elN`: (ha van) beszúrandó elem(ek) a tömbbe.
Visszaadja a törölt elemeket.

```
var szinek = [ 'piros', 'sárga', 'kék' ];
var toroltek = szinek.splice( 1, 2, 'fehér', 'zöld' );

// Törölt színek
for( var t in toroltek ) {
        console.log( toroltek[ t ] ); // Törölt elemek: sárga,kék
}

// Ami maradt
for( var sz in szinek ) {
        console.log( szinek[ sz ] ); // Tömb elemei: piros,fehér,zöld
}
```

Ciklusok

`for..in`: az objektumon található tulajdonságok nevén halad végig
`for..of`: **iterátor** segítségével veszi sorra a tömb értékeit (kulcs-érték párokat nem olvas ki (?))

Függvények:

Nem lehet megadni a:

- paraméterek *típusát*
- *visszatérési érték típusát*

Nem okoz gondot, ha eltér a híváskor a paraméterek száma a deklarációhoz képest.

- Kevesebb esetén a nem definiáltak értéke `undefined` lesz.
- Több esetén figyelmen kívül hagyja.
- Függvény híváskor a paraméterekre névvel nem lehet hivatkozni, ezért csak a paraméter lista végéről tudunk elhagyni elemeket.

❗ **Nincs overload!**

- ha két azonos nevű függvény van, akkor a később deklarált fog nyerni.

🔖 alapértelmezett értéket viszont *lehet adni* a függvénynek.

A függvény is egy teljes értékű típus!

- Egy *függvénynek lehet függvény a bemenő paramétere*
- Arra viszont nekünk kell figyelni, hogy tényleg függvény-e a paraméter.

Scope

Böngészőben futtatott JavaScript kód esetén a `window` objektum segítségével hivatkozhatunk az oldalhoz tartozó ablakra vagy keretre (**frame**).

- Rengeteg általános tulajdonság, metódus, és esemény definiálva van rajta keresztül.

📍 A `var`-ral létrehozott változók amik nincsenek függvénybe zárva, a **window** objektumon jönnek létre.

- A függvények is a **window** objektumra kerülnek.

📍 Nincs közvetlen lehetőség a privát és publikus tagok megjelölésére.

- A változók megtalálása a legmélyebb szintről felfelé történik.
  - Így halad felfelé egészen a globális névtérig
  - **shadowing**: a lokális változók elfedik a külső változókat.

Javascript API-k

DOM API a HTML és CSS tagek dinamikus manipulálására.
XmlHttpRequest: Szerver kommunikációt lehetővé tevő API.
Client Storage API: web storage, IndexedDB-hez való hozzáférést segíti

Legfontosabb objektumok:

window: A böngésző tabfüle, amibe a weboldal betöltődik.
navigator: A böngéső állapotát tárolja
- Lekérdezhető vele a nyelv, geolokáció, stb.
document: Maga a **DOM**, ami a window objektumra betöltődik. Ezen keresztül tudjuk módosítani a HTML-t.
- A *DOM* egy olyan modell, mely leírja egy HTML oldal felépítését egy fa struktúrában, melynek gyökér eleme a „*Document*” objektum, ami alatt az oldalon lévő elemek találhatók hierarchikusan.

Elemek lekérdezése a DOMból:

```
//Elem lekérdezése tag alapján
document.getElementsByTagName("img")

//Adott CSS osztállyal rendelező elem:
document.getElementsByClassName()

//Adott ID-jú elem:
document.getElementById( ... )

//Adott Name-mel rendelkező elemet
document.getElementsByName( ... )

/*tetszőleges elem lekérdezése selectorral: */

//csak a legelső találatot adja vissza:
document.querySelector('a');

//az összeset visszaadja:
document.querySelectorAll('a');
```

Elemek dinamikus létrehozása

```
//A createElement() segítségével új HTML elemeket hozhatunk létre.

//Új bekezdés elem
var p = document.createElement('p');

//Adjunk neki valami szöveget
p.textConent = 'Új bekezdés';

//Fűzzük hozzá a HTML-be egy már létező elemhez
var section = document.querySelector('section')
section.appendChild(para)

//Módosítani is lehet a lekérdezett adatokat
p.stlye.color = 'white';
p.sytle.textAlign = 'center';
//...

//Új attribútumot is rátehetünk egy-egy elemre futási időben:
p.setAttribute('class', 'highlight')
```

⚠️ A tulajdonságokat másképpen írjuk CSS-ben és JS-ben! JS-ben camelCase-et használunk!

Események kezelése

A felhasználói interakciókat kliens oldalon kezelni kellene. Számos eseményre tudunk feliratkozni, pl *kattintás*, *billentyűlenyomás*, *fókuszbakerülés*, *egér elem fölé vitele*, *beviteli mező tartalma megváltozott*,...

- A HTML-ből inline is feliratkozhatunk, bár a karbantarthatóságot jelentősen csökkenti.

⚠️ Eseménykezelőt csak akkor tudunk egy elemhez regisztrálni, **ha az elem már létezik.**

- Ezt tudjuk elérni a `window.onload` használatával. A lefutása előtt nem biztos, hogy léteznek az elemek a HTML-ben!
- megadhatjuk az esemény neve mell az `on` prefixet is:

```
// Ezek csak akkor működnek, ha egyetlen függvényt akarunk beregisztrálni!
btn.onclick = function() {/*...*/}
btn.onclick = kiir;

/** Több eseménykezelő beregisztrálása
        Erre szolgál az addEventListener()
**/
var btn = document.querySelector('#myBtn');
btn.addEventListener("click", myFunc);

//vagy:
btn.addEventListener("click", function(){/*...*/})
```

⚠️ Az eseménykezelők a `regisztráció sorrendjében egymás után futnak le.`

Ha a fában több elemnél is feliratkozunk például a kattintás eseményre, akkor:

1. Fentről lefelé megkeresi a böngésző, hogy melyik elemre kattintottunk.
2. Meghívja az ott beregisztrált eseménykezelőt.
3. Majd ha az lefutott, akkor az esemény felgyűrűzik egészen a gyökér elemig, még ebben a fázisban is kezelhetjük az eseményt.
   `stopPropagation()`-el le is állíthatjuk ezt a felgyűrűzést.

Állapotkezelési megoldások

- A cookie-kal gondok vannak, lásd WEB01-Webes Architektúra, HTTP, HTML, HTTPS > Állapotmegőrzés

👎 Több böngésző ablakban egy időben nehezen használható (HTTP kérésekhez kötődik)

- Megoldás: **Web Storage**
  - *kulcs-érték* párok tárolására találták ki.
    - más típusú értékek automatikusan stringre konvertálódnak
    - `JSON.parse()` és `JSON.stringify()` segítségével könnyen kovertálhatúak komplex struktúrák JSON-ná.

  👍 Egyszerre több ablakból is könnyedén használhatjuk az oldalt.

  - méretkorlát van rajtuk (~5MB/origin)

  1. **Session Storage**
     - Az információk csak a tab **bezárásáig** maradnak meg
     - Csak az adott ablak érheti el az adatot.
  2. **Local Storage**
     - Az adott domainhez tartozó összes oldal elérheti a tárolt adatokat
     - Bármikor törölheti a felhasználó

```
/* LocalStorage használata */

//Elem hozzáadása
```

```
localStorage.setItem('myCat','Tom');

//Elem kiolvasás
const cat = localStorage.getItem('myCat');

//Elem törlése
localStorage.getItem('myCat');

//Összes elem törlése
localStorage.clear();


/* Session Storage használata */
sessionStorage.setItem('myCat','Tom');
sessionStorage.getItem('myCat');
sessionStorage.getItem('myCat');
sessionStorage.clear()
```

- 👎 Kis mennyiségű adat tárolására alkalmas
- 👎 Csak string kulcs-érték párokat tud tárolni
- 👎 Csak szinkron API van.
- 👎 Nem lehet keresni az adatok közt.
- 👍 optimalizált.

Szempont	Cookie	Storage
Méret korlát	4KB	5MB (2.5MB)
Élettartam	Session és persistent	Session és local
Tartalom típus	String	String
Hálózati forgalom	Utazik	Nem utazik
API	Kliens és szerver oldali	Csak kliens oldali, van eseménykezelés
Böngésző támogatás	Mindegyik	Szinte mindegyik
Biztonság	Hálózaton és kliensen is támadható, de lehet HttpOnly	Kliensen támadható

Indexed Database

Cél: **nagy mennyiségű adat tárolása kliens oldalon** + gyors keresés indexekkel.
Kliensoldali gyorsítótárazásra, teljes offline működésre nagyon hasznos.

- 👍 Tartozik hozzá aszinkron API
  - Kéréseket lehet definiálni, amik callbackeket hívnak meg
  - Nem SQL alapú
- 👍 Kulcs-érték, de az érték lehet összetett objektum
- 👍 Tranzakcionális modell
- 🔖 10MB-2GB méretek, **same origin policy!**
- 👎 Nem támogatja a nyelvfüggő rendezéseket

👎 Nem tud szerveroldali adatbázissal szinkronizálni
👎 Nem támogatott a szabadszöveges keresés, nincs `LIKE` operátor

History API

📌 A teljes oldalt nem akarjuk frissíteni, de működnie kellene a böngésző Back/Forward gombjainak.

Hol tároljuk az állapotot?:

- Bookmarkolható az állapot? - *URL*
- A szerveroldalon is szükség van rá? - *URL, hidden field, cookie*
- Kis mennyiségű, egyszerű adat?: - *DOM Storage*
- Nagyobb mennyiségű, vagy bonyolultabb lekérdezések? - *IndexedDB*
- Navigációval összefüggő állapotot kell tárolni? - *History API*

Biztonság szempontjából:

❗ A tárolt adatokat bárki láthatja - **titkosítani kell**
❗ Bárki módosíthatja - **integritásvédelem!**

Megbízhatóság:

❗ Az adatot bárki törölheti teljes egészében, vagy részében - **fallback**
❗ A kvóta limitet elérhetjük
❗ A felhasználó bármikor megnyithat több böngésző ablakot.

Feketemágia a függvényekkel - Closure

Függvények vannak egymásba ágyazva, de a külső függvény elérhetővé teszi a külvilág számára a belső függvényt.

❗ A belső függvény megőrzi azt az állapotot, ami a létrehozása pillanatában volt.

✎ Closure létrejötte

Amikor egy függvény egy belső függvényét láthatóvá teszi a külvilág számára, akkor egy ún. **closure** jön létre, ami nem más, mint az adott belső függvény és a hozzá tartozó állapot együttvéve.

- Minden függvény egyben egy closure is (kivétel a `new Function()`, ami szövegből készít függvényt)

```
/** Closure névtelen függvénnyel
- A hozzátartozó állapot is vele van **/
let kulso = function () {
        let x = 8;
        return function () {
                alert(++x);
        }
};

let b = kulso();
b(); // 9
b(); // 10
```

Automatikusan futtatható függvények

```
/* Self executing functions*/
(function (nev) {
        alert('Szia ' + nev);
})('világ');
```

Szintaktikailag a neve helyén van maga a function.

Modul tervezési minta

A kódunkat egy minden mástól független névtérbe csomagolhatjuk, elkerülve a globális névtér szennyezését.

- Kisebb logikai egységekre bonthatjuk a kódunkat.
- Megvalósítható vele az egységbezárás.

Becsomagolás egy névtérbe:

```
// Modul létrehozása
var myModule = (function()
        'use strict'

        // Ide jön a kód
        // Minden függvény és változó ehhez a függvényhez tartozik

}());
```

Csak a `return`-ön belüli kód lesz "publikus"

```
var myModule = (function () {
        var priValt = 3;
        var priFv = function () { alert('Privát!'); };
        return {
                pubValt: 5,
                pubFv: function () { alert('Publikus'); }
        };
})();

myModule.pubFv();
alert(myModule.pubValt);
```

A fenti példa több fájlra is darabolható:

```
var myModule = (function (my) {
        my.pubFv = function () { alert('Publikus'); };
        return my;
}(myModule || {}));

// Másik fájl
var myModule = (function (my) {
        my.pubValt = 5;
        return my;
}(myModule || {}));

// Felhasználás
myModule.pubFv();
alert(myModule.pubValt);
```

Konstruktor:

Segítségével a modulnak adhatunk bemeneti paramétereket (**import**), a `return` kulcsszó után pedig azt határozhatjuk meg, hogy kívülről mi látsszon a modulból (**export**).

```
//sayHi.js
export function sayHi(user) {
        alert(`Hello, ${user}!`);
```

```
}

//main.js
import {sayHi} from './sayHi.js';
alert(sayHi); // function...
sayHi('Gábor'); // Hello, Gábor!
```

- Az import direktíva a `./sayHi.js` útvonalról betölti a modult, és hozzárendeli az exportált `sayHi` függvényt a változóhoz.

new, this

A konstruktor függvény egy normál függvény.

- Nagy betűvel szoktuk kezdeni.
- Csak a new operatorral használhatjuk. `new` nélkül `undefined`-ot kapunk.

```
function User(name) {
        if (!new.target){ // ha new nélkül hívtuk
                return new User(name)
        }
        this = {}; // implicit belekerül
        this.name = name;
        this.isAdmin = false;
        return this; // implicit belekerül
}

let user = new User("Gábor");
```

A new egy **objektumot** hoz létre.

Ha kézzel beleírjuk a `return`-t, akkor a megadott objektummal tér vissza.

⚠️ Ha nem objektum szerepel a return mögött (üres vagy primitív típus), akkor a `this`-t adja vissza.

Metódust is lehet a konstruktor függvényben definiálni:

```
function User(name) {
        this.name = name;
        this.sayHi = function() {
                alert( "My name is: " + this.name )
        };
}

let gabor = new User("Gábor");
gabor.sayHi(); // My name is: Gábor
```

Gondok a this-sel:

Előfordul, hogy a `this` kulcsszó nem arra az objektumra mutat, amin belül használjuk.
Például egy eseménykezelőben a `this` a DOM elemre mutat. A hiba elkerülése érdekében ezért a `this` használatát kerülni szokás, és az osztály tagjait egy másik változón keresztül élrjük el (pl `that, self`)

```
function User(name) {
        let self = this;
        self.name = name;
        self.sayHi = function() {
                setTimeout( function() {
                        alert ( "My name is: " + self.name )
                }, 1000)
```

```
        };
}

let gabor = new User("Gábor");
gabor.sayHi(); // My name is: Gábor
```

Arrow function

Tömörebben leírhatunk vele függvényeket, olvashatóbb lesz a kód.

```
//1.
nev => console.log(nev);

//2.
() => console.log("Alma");

//3.
(varos, utca) => {
        let cim = varos + utca;
        console.log(cim)
}

//4. - Egyszerű esetekben nem kell kiírni a return-t:
var func = x => x * x;

//5. Ha van {}, akkor kell return
var func = (x, y) => { return x + y; };

//6. Objektum nem adható vissza simán
var func = () => { foo: 1 }; //!

//Helyette zárójelezni kell:
var func = () => ({foo: 1});
```

⚠ Az Arrow functionnek nincs saját this-e!

JavaScriptben a legtöbb függvénynek van saját this-e, ami időnként megnehezíti a kódolást (amint feljebb láttuk a gondoknál)

- Arrow function esetén viszont **nincs saját this**, tehát nem lehet átállítani, hogy magára mutasson.

```
//Arrow functionben használható a this:
function Person(){
        this.age = 0;
                setInterval(() => {
                        this.age++;
                        // a this helyesen a Person objektumra mutat
                }, 1000);
}
```

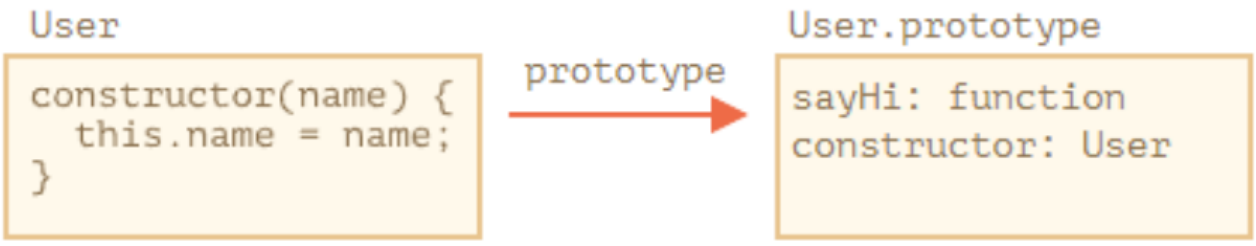
Classok

- Nem egy teljesen új nyelvi elem

```
class User {
        constructor(name) { this.name = name }
        sayHi() { return 'My name:' + this.name; }
}
```

```
alert(typeof User); // function
```

Létrehoz egy `User` nevű függvényt, ami az osztály deklaráció eredménye lesz. A kódja pedig a `constructor` metódusból származik. A **prototípuson** eltárolja a metódusokat



- class-t csak `new`-val lehet hívni.
- Ha kiíratjuk szövegesen az osztályt, az class-al kezdődik.
- Mindig strict módban futnak
- Nem lehet végigiterálni a metódusain

```
//Getter/setter használata
class User {
        age = 40; // Mezőket is adhatunk hozzá
        constructor(name) { this.name = name; /*A settert hívja*/ }
        get name() { return this._name; }
        set name(value) {
                if (value.length < 4) { alert("A név rövid."); return; }
                this._name = value;
        }
}
```

Backtick: *nincs szükség a sztring összefűzésekre*: `return `My name: ${this.name} `;`

- Bár függvényeknél megtehetjük, hogy korábban hívjuk meg, mint deklaráljuk, mert a deklarálást kiemeli a kódból. Viszont ugyanez az osztályokra már nem igaz.

Származtatás

Arra is lehetőségünk van, hogy **az osztályok származzanak egymásból**. Ehhez az `extends` kulcsszót kell használnunk. Az ősosztály függvényét például a `super.toString()` a konstruktorát pedig a `super()` segítségével tudjuk meghívni.

```
class ColorPoint extends Point {
        constructor(x, y, color) {
                super(x, y);
                this.color = color;
        }

toString() { return super.toString() + ' in ' + this.color; }
}

let cp = new ColorPoint(25, 8, 'green');
console.log( cp.toString() ); // '(25, 8) in green'
console.log(cp instanceof ColorPoint); // true
console.log(cp instanceof Point); // true
```

Promise

A JavaScript alapvetően aszinkron.

A Promise egy olyan objektum, ami majd a jövőben visszaad egy értéket, de nem most. Emiatt tökéletes *aszinkron* kérések kezelésére.
Három állapota van:

1. *Pending* - függőben van
2. *Fulfilled* - sikeres
3. *Rejected* - hibára futott
   Mindig Pendig állapotból indul és Fulfilled vagy Rejected állapotban ér véget.

Eredmény feldolgozása:

`.then(success, error)`
- Akkor hívódik meg, ha lefutott a Promise.
- Ha sikeresen futott le, akkor a `success` handler hívódik meg.
- Ha sikerteleül, akkor az `error`.
`.catch(f)`
- Csak akkor fut le, ha a Promise hibával tért vissza.
`.finally(f)`
- Minden esetben lefut, ha sikeres, ha nem, de nem tudjuk megmondani, hogyan futott le.

```
learnMobWeb.then(
        result => alert(result);
        error => alert(error);
);

learnMobWeb.catch(alert);
learnMobWeb.finally( () => /* Stop loading */ )
```

`Promise.all(promises)`: megvárja, hogy az összes befejeződjön
`Promise.allSettled(promises)`: Megvárja, hogy az összes Promise befejeződjön és visszaadja, hogy melyik volt sikeres és melyik hibás.
`Promise.race(promises)`: Csak az első Promise-t várja meg és annak eredményét adja vissza.
`Promise.any(promises)`: Az első sikeresen befejeződött Promise-ra vár.

Async/ await

- Kényelmesen kezelhetünk vele `Promise`okat.
- A függvény előtt lévő async azt jelenti, hogy a függvény egy `Promise`-sal tér vissza.
- így használható lesz a függvényre a `.then()`

```
async function f() {
        let promise = new Promise((resolve, reject) => {
        setTimeout(() => resolve("Kész!"), 1000)
});

        let result = await promise; // Vár a promise-ra
        alert(result); // "Kész!"
}

f();
```

> .then és await különbségek

"While `.then` is used for handling promises in a chained manner, `await` is used within `async` functions to pause execution until a promise is settled, providing a more concise and readable syntax for handling asynchronous operations."

Fetch API

`fetch()` segítségével hálózati kéréseket küldhetünk a szerver fele. Korábban ezt csak az `XmlHttpRequest`-el tudtuk elérni.
Támogatja a **WEB05-jQuery-ajax-websocket > CORS**-t, tehát tetszőleges szerver felé indíthatunk kéréseket, és `Promise`-sel tér vissza.

⚠️ A Fetch API-ban a Promise csak akkor reject-elődik, ha hálózati hiba van. Egyébként pedig státuszkódtól függetlenül sikeresen tér vissza, ha a szerver válaszolt.
`let promise = fetch(url, [options])`

- A fetch `Promise` a beépített `Response` osztályt adja vissza, amiben a szervertől visszakapott Header-ek találhatók. A válaszból a body-t egy újabb `Promise`-sal kapjuk meg, amit utána `.json()` függvénynel parsolunk.
- Fetch példa:

```
let response = await fetch(url);

if (response.ok) { // HTTP-status 200-299
        // Response body kinyerése
        let json = await response.json();
} else {
        alert("HTTP-Error: " + response.status);
}
```

Rengeteg mindent meg lehet adni az options mezőhöz, köztük a headereket, methodokat, stb.