JS1

Jegyzet – Változók (bevezető után)

ES6, azaz az ECMAScript 2015-ös szabvány

A következő leckékben a röviden csak *ES6*-nak nevezett új szabvány JavaScript-újdonságairól fogtok tanulni.

Mindenekelőtt hadd szögezzem le, hogy amennyiben bármelyik "újdonságot" (2015 nem tegnap volt, szóval annyira nem is új) szeretnétek használni, győződjetek meg róla, hogy milyen böngésző támogatja, és melyik verziótól. Erre a [kangax](https://kangax.github.io/compat-table/es6/" \t "_blank) vagy a [caniuse](https://caniuse.com/" \t "_blank) oldal tökéletes.

Az [MDN Web Docs](https://developer.mozilla.org/) oldalon is sok érdekességet olvashattok. A hivatalos [ECMAScript-dokumentációban](https://www.ecma-international.org/ecma-262/" \t "_blank) pedig szinte minden kérdésre megtaláljátok a választ.

Polyfillek, transpiler-ök

Ha egy új projektet kezdtek, a specifikációban meg lesz szabva, hogy az alkalmazásnak melyik böngészők melyik verziójában kell működnie. Ekkor már tudjuk, hogy az új szabványból mi is az, ami natív módon támogatott. Szerencsére nem kell kétségbeesni akkor sem, ha olyan böngészőre kell fejlesztenünk, amely nem támogatja az új szabványt. Lehetőségünk van ugyanis polyfillek és transpiler-ök használatára.

Polyfill

*A polyfill egy olyan kód/kódrészlet, amely biztosítja számunkra azt a működést, amelyet natív módon elvárnánk a böngészőtől.*

Abban az esetben, ha valamilyen új szabvány adta új property-t, method-öt szeretnénk használni, de az nem támogatott valamelyik böngészőben, amelyben működnie kell a kódunknak, nagy segítségünkre lehetnek a polyfillek. Ezek segítségével emulálni tudjuk a működést.

Ha használni akarjuk a String.includes() metódust, de az IE11-ben nem támogatott, írhatunk egy saját kódot vagy használhatunk már mások által megírtakat, hogy az új metódus működését szimuláljuk.

if (!String.prototype.includes) {  
  String.prototype.includes = function(search, start) {  
    'use strict';  
    if (typeof start !== 'number') {  
      start = 0;  
    }  
    if (Number.isNaN(search) || start + search.length > this.length) {  
      return false;  
    } else {  
      return this.indexOf(search, start) !== -1;  
    }  
  };  
}

A fenti példa használja a Number objektum isNaN() metódusát, amely ha nem támogatott, szintén egy polyfillre van szükségünk.

Valahogy így:

Number.isNaN = Number.isNaN || function(value) { return value !== value; }

Természetesen könnyen lehet, hogy nekünk a globális isNaN() metódus is megfelelne, a későbbiekben lesz majd szó a különbségekről.

Transplier

*A transpiler-ök olyan eszközök, amelyek egy adott nyelvben írt kódot átalakítanak egy másik nyelv ekvivalens kódjává. A JavaScript esetében ez nem feltétlenül egy másik nyelvet, hanem a szabvány egy másik verzióját jelenti.*

Tehát transpiler-öket akkor érdemes használni, ha nemcsak új tulajdonságokat, metódusokat akarunk használni, hanem újfajta szintaxist is. A traspiler-ök sem támogatnak mindent. Ezt mindig ellenőrizzük! A transpiler-ök használata által generált kód performancia szempontjából sokkal rosszabb lehet, mint a natív kód.

A [Babel](https://babeljs.io/) az egyik legnépszerűbb transpiler. Csak a szintaxist alakítja át, de a babel-polyfill segítségével a Promise-ok és az új metódusok használata is támogatott.

Arra szolgál, hogy az általunk megírt *ES6*-os*JavaScript-*kódot áttranszponálja a régi szabvány szerintire. Más szóval az *ES6*-os (vagy újabb) kódunkból *ES5*-os verziójút készít. (A Babel beállításai és pluginjai segítségével rengeteg egyéb opció is a rendelkezésünkre áll.) Remekül testre szabható, konfigurálható eszközről van szó. Nem méltán népszerű. Nézzünk meg egy példát, mi történik, ha az új szabvány szerinti class kulcsszóval hozunk létre egy új osztályt, és ezt a kódot a régebbi szabvánnyá alakítjuk.

**Az eredeti kód:**

class A {}

**és a transzponált:**

function \_classCallCheck(instance, Constructor) {  
  if (!(instance instanceof Constructor)) {  
    throw new TypeError('Cannot call a class as a function');  
  }  
}  
  
var A = function A() {  
 \_classCallCheck(this, A);  
};

Változók és konstansok

var - globális/lokális változók

A *JavaScript*ben eddig a var kulcsszó segítségével hoztunk létre változókat. Az így létrehozott változók vagy globális hatókörrel (global scope) rendelkeztek, bárhonnan elérhettük őket, vagy ha függvényen belül hoztuk őket létre, akkor csak a függvényen belül "léteztek" (lokális hatókör - local scope). Az *ES6-*szabvány óta ez a helyzet megváltozott.

let - blokkszintű változók

Blokkszintű (block scoped) változókat tudunk létrehozni a let kulcsszó segítségével. Amennyiben let-tel hozunk létre egy változót, ugyanúgy lehet globális, mint var esetében - ha nem függvényen belül hozzuk létre -, de egyébként csak egy adott blokkon belül - leegyszerűsítve a nyitó { és a záró } között - létezik.

A következő esetben jól látható a különbség:

**var**

{  
  var x = 10;  
  console.log(x); // 10  
}  
console.log(x); // 10

**let**

{  
  let x = 1;  
  console.log(x); // 1  
}  
console.log(x); // Uncaught ReferenceError: x is not defined

Újradeklarálás

**var**

A var-ral deklarált változót újra lehet deklarálni.

var foo; // undefined  
var foo; // undefined

**let**

A let-tel deklarált változót nem lehet újradeklarálni.

let foo; // undefined  
let foo; // Uncaught SyntaxError: Identifier 'foo' has already been declared

Azonos változónevek globális és lokális scope-ban

**var**

var x = 1;  
{  
  var x = 2; // ugyanaz az x változó!  
  console.log(x); // 2  
}  
console.log(x); // 2

**let**

let x = 1;  
{  
  let x = 2; // ez egy másik x változó  
  console.log(x); // 2  
}  
console.log(x); // 1

A let és a switch case

A switch egyetlen blokk.

switch(x) {  
    case 0:  
        let foo;  
        break;  
    case 1:  
        let foo; // Uncaught SyntaxError: Identifier 'foo' has already been declared  
        break;  
    default:  
}

TDZ - Temporal Dead Zone

A let-tel létrehozott változó inicializálás előtt nem elérhető. Van hoisting itt is. A var-ral definiált változóhoz képest a let-tel létrehozottnál a különbség az inicializálásnál van. A var inicializálódik undefined értékkel, azonban a let inicializálatlan marad. Tehát ha megpróbáljuk elérni, ReferenceError-t kapunk. A let\const\class csak akkor lesz inicializálva, amikor az utasítás kiértékelődik. A kiértékelés előtti részt Temporal Dead Zone-nak hívjuk.

console.log(foo); // ReferenceError  
let foo = 2;

for loop és a let

Ha a for loop header-ben deklaráljuk, minden egyes iterációnál új binding lesz. Valamint a scope a ciklusmagra is vonatkozik. (Az IE11 nem támogatja a let-et ebben az egy esetben, sem strict módban, sem anélkül. Mintha var-t használtunk volna.)

for (let i = 0; i <= 5; i++) {  
 setTimeout(function() {  
    console.log(i); // 0 1 2 3 4 5  
  }, i \* 1000);  
}

Ellenkező esetben:

let i = 0;  
for (i = 0; i <= 5; i++) {  
 setTimeout(function() {  
    console.log(i); // 6 6 6 6 6 6  
  }, i \* 1000);  
}

A let-et emiatt nagyon jól használhatjuk több elemhez való eseményfigyelő hozzárendelésénél is:

let elements = document.getElementsByClassName('sample’);  
  
for (let i = 0; i < elements.length; i++) {  
 elements[i].addEventListener('click', function () {  
 console.log('item: ' + i);  
    });  
}

const - konstansok

Lehetőségünk van konstansok létrehozására a const kulcsszóval. A konstans egy immutable binding-ot készít. Tehát nem a konstansban tárolt adat, hanem a binding maga módosíthatatlan. Ebből következik, hogy nem adhatunk új értéket, nem deklarálhatjuk újra. A let-nél tárgyalt scope-szabályok a konstansra is érvényesek.

**Nézzünk egy példát:**

const myConst = 'Constant';  
console.log(myConst); // 'Constant'  
myConst = 'Change value'; // TypeError: Assignment to constant variable

Láthatjuk, hogy ha a konstans értékét meg akarjuk változtatni, hibát kapunk. Azonban van egy érdekes kivétel:

const person = {  
  firstName: 'John',  
  lastName: 'Doe',  
  age: 31  
};  
  
person.age = 40;  
console.log(person); // {firstName: 'John', lastName: 'Doe', age: 40}

Nem kapunk hibát, az objektum tulajdonságát megváltoztathatjuk. A konstansok értékét nem módosíthatjuk, azonban ha a konstans egy objektumot tárol, akkor a tulajdonságai, metódusai módosíthatók.

Azonban:

const person = {  
  firstName: 'John',  
  lastName: 'Doe',  
  age: 31  
};  
  
person = {  
  firstName: 'Jane',  
  lastName: 'Doe',  
  age: 20  
}; // TypeError: Assignment to constant variable

Ebben az esetben hibát kapunk. Azt nem tehetjük meg, hogy egy új objektumot vagy egy primitívet adunk neki új értékül. A reassignment nem megengedett.

Object.freeze() - az objektum "lefagyasztása"

A JavaScriptben lehetőségünk van arra, hogy egy objektumot *"lefagyasszunk"*. Ezzel elérhetjük azt konstansok esetében, hogy ne tudjuk megváltoztatni az objektum tulajdonságait, metódusait. A *"lefagyasztott"* elem immutable. Nemcsak konstansoknál használhatjuk, hanem bármilyen objektumon.

const person = {  
  firstName: 'John',  
  lastName: 'Doe',  
  age: 31  
};  
  
Object.freeze(person);  
person.firstName = 'NewName';  
  
console.log(person); // {firstName: 'John', lastName: 'Doe', age: 31}

Látjuk, hogy az objektum tulajdonsága nem változott meg.

Konstansok névadása

A legtöbb programozási nyelvben a konstansok névadásánál csak nagybetűket használnak.  
JavaScriptnél ugyanúgy camelCase neveket használjunk, mint a var vagy a let esetében.

Mikor melyiket használjam?

Felmerülhet a fenti kérdés. Nincs hivatalos szabály, a bevett gyakorlat a következő:

* a var-t többé ne használjuk
* ha az adott primitív érték biztosan nem változik meg, használjunk const-ot
* objektumok esetében szinte mindig használjunk const-ot
* minden más esetben pedig let-et

Memory life cycle

1. Memóriaterület lefoglalása (allocate)
2. Memóriaterület írása/olvasása (use)
3. Memóriaterület felszabadítása (release – garbage collection)

Jegyzet – Függvények (bevezető után)

Függvények

Definíciók

* callback: olyan függvény, amelyet paraméterként adunk át egy másik függvénynek
* higher order function: olyan függvény, amelyik más függvényekkel (pl. callback-ekkel) dolgozik
* closure: olyan függvény, amely esetében a belső hatókör mindig hozzáfér az őt körülvevő, külső hatókör (parent scope) változóihoz (tehát egy változó mindaddig elérhető, amíg a kódban szerepel rá mutató referencia; ha nincsen ilyen, a garbage collector törli a memóriából)
* IIFE (Immediately Invoked Function Expression): azonnal meghívott függvénykifejezés

**Arrow function**

Elérkeztünk a függvényekkel kapcsolatos újítások legzseniálisabb részéhez, azaz az arrow function-ökhöz.

Nézzük meg, mi is ez az új CSODA. Most a gyakorlatban is meglátjuk, hogy hogyan teszik egyszerűbbé az életünket az arrow function-ök.

Az arrow function segítségével egyszerűbben tudunk létrehozni függvényeket, mint függvénykifejezéssel.

const people = [  
  {  
    firstName: 'John',  
    lastName: 'Doe'  
  },  
  {  
    firstName: 'Jane',  
    lastName: 'Doe'  
  }  
];  
  
  
const template = `<div>  
            ${people  
              .map(function(person) {  
                return `<p> ${person.firstName} ${person.lastName} </p>`;  
              })  
              .join('')}  
        </div>`;

Egyszerűsítsük:

const people = [  
  {  
  firstName: 'John',  
    lastName: 'Doe'  
  },  
  {  
    firstName: 'Jane',  
    lastName: 'Doe'  
  }  
];  
  
const template = `<div>  
            ${people  
              .map(person => `<p> ${person.firstName} ${person.lastName} </p>`)  
              .join('')}  
    </div>`;  
  
/\*  
        <div>  
            <p> John Doe </p>  
            <p> Jane Doe </p>  
        </div>  
        \*/  
console.log(template);

Ugyanazt kapjuk a kimeneten. De mi is történt? A callback függvényünket átírtuk egy újfajta szintaxisra.

* elhagytuk a function kulcsszót
* az arrow function mindig anonim lesz
* a paraméter körüli zárójeleket töröltük (EZT CSAK AKKOR TEHETJÜK MEG, HA 1 PARAMÉTER VAN!)
* elhagytuk a kapcsos zárójeleket (NEM MINDIG!)
* egy => jelet írunk a paramétere(k) után
* elhagytuk a return kulcsszót

Nézzünk egy másik, egyszerűbb példát is:

const languages = ['java', 'php', 'python'];

// Ezt írjuk át:

const scriptsOldStyle = languages.map(function(language) {  
  return `${language} script`;  
});

// Erre:

const scriptsNewStyle = languages.map(language => `${language} script`);

A fenti példán is jól láthatók a szintaxisbeli különbségek. Nézzünk meg egy példát több paraméterrel:

const languages = ['java', 'php', 'python'];  
  
const languageRank = languages.map(  
  (language, i) => `${i + 1}. ${language} script`  
);  
  
console.log(languageRank); //  ['1. java script', '2. php script', '3. python script']

Láthatjuk, hogy több paraméter esetén a zárójeleket ki KELL tennünk. Nézzünk egy példát egy összetettebb függvényre:

const alertMessage = message => {  
  alert(`Alert ${message}`);  
 console.log(`Alert ${message}`);  
};  
  
alertMessage('Message');

Tehát, ahogy említettem, ha nemcsak egy egyszerű utasításunk van a függvényben, a kapcsos zárójeleket kitesszük, illetve ha nincs kapcsos zárójel, az azt jelenti, hogy visszatérési értékünk lesz. Nézzünk még egy érdekes példát arra az esetre, hogy ha a függvénynek egyáltalán nincs paramétere:

const infoMessage = () => console.log('Arrow function');  
infoMessage();

Nézzünk még egy szép összetett real-life példát funkcionális programozási szemléletmóddal így a végére:

Van egy webshopunk, és a HTML-kódból meg kell mondanunk, hogy az általunk árult szerverek összesített ára mennyi:

    <!DOCTYPE html>  
    <html lang='hu'>  
  
  
    <head>  
        <meta charset='UTF-8'>  
        <meta name='viewport' content='width=device-width, initial-scale=1.0'>  
        <meta http-equiv='X-UA-Compatible' content='ie=edge'>  
        <title>Arrow Function Example</title>  
    </head>  
  
  
    <body>  
        <ul id='price-list'>  
            <li>  
                TV  
            </li>  
            <li data-company='Dell'>  
                Server  
            </li>  
            <li data-price='11.012,12'>  
                TV  
            </li>  
            <li data-price='23.243,55'>  
                Server  
            </li>  
            <li data-price='4.543,29'>  
                TV  
            </li>  
            <li data-price='230.323,32'>  
                Server  
            </li>  
            <li data-price='48.344,30'>  
                Server  
            </li>  
        </ul>  
    </body>  
    <script>  
        const serverPrice = Array.from(document.querySelectorAll('#price-list li[data-price]'))  
                                 .filter(item => item.textContent.includes('Server'))  
                                 .map(item => parseFloat(item.dataset.price.replace('.', '').replace(',', '.')))  
                                 .reduce((total, item) => total + item);  
  
        console.log(serverPrice);  
    </script>  
  
  
    </html>

(A kódot - amint használjátok - formázzátok meg; a tanfolyami felületen a fenti kód formázása másolható formában csak így lehetséges.)

Mindössze annyit teszünk, hogy használjuk a tömbmetódusainkat, most már arrow function-ökkel kombinálva. (Az Array.from() szintén hamarosan kifejtésre kerül, csakúgy, mint az includes, ami leginkább az indexOf()-ra hasonlít.)

* Kigyűjtjük egy konstansba a price-list id-jú elemen belül az összes li-t,amelynek a data-price attribútuma meg van adva.
* Leszűrjük az elemeket, hogy a *"Server"* szót tartalmazó elemek maradjanak, tehát csak a szerverek árai érdekelnek.
* Tömböt készítünk ezen elemek áráról, amely érték már floattá lesz alakítva, és figyelünk a *","* karakterekre is, amelyeket *"."*-ra cserélünk.
* A kapott összegeket összeadjuk.

**Arrow function és a return utasítás**

// return: undefined  
// Üres blokk, implicit return  
(firstName => {})()  
  
// return: 'Hello John'  
// Nincs blokk, implicit return  
(firstName => 'Hello ' + firstName)('John')  
  
  
// return: undefined  
// explicit return kell a blokkon belül, de Hello hiányzik  
(firstName => {'Hello ' + firstName})('John')  
  
  
// return: 'Hello John'  
// explicit return a blokkon belül  
(firstName => {return 'Hello ' + firstName})('John')  
  
  
// return: undefined  
// a block egy labelt tartalmaz, nincs explicit return  
// https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/label  
(firstName => {name: firstName})('John')  
  
  
// return: {name: 'John'}  
// implicit return a ( )-en belül, amely egy objektum  
(firstName => ({name: firstName}))('John')  
  
  
// return: name: 'John'  
// explicit return a blokkon belül, objektum  
(firstName => {return {name: firstName}})('John')

**Arrow function és a local binding**

*A JavaScript-fejlesztők élete nem csak játék és mese.* Az arrow function-nél nincs local binding. Tehát nincs arguments, this, super, new.target. Ha az arrow function-ön belül próbáljuk meg elérni, a parent scope lesz használva.

Nézzünk egy példát this esetére, és hogy mikor jó nekünk az arrow function e tulajdonsága:

/\*

    A régi módszer: a this-t beletettük a that-be, hogy a setInterval-on belüli callback-ben is elérjük

\*/

function People() {  
  var that = this;  
  that.age = 0;  
  
  setInterval(function setAge() {  
    that.age = 30;  
  }, 1000);  
}

// És most:

function PeopleWithAF() {  
  this.age = 0;  
  
  setInterval(() => {  
    this.age = 30;  
  }, 1000);  
}

Íme, látjuk, hogy ez szép, jó és hasznos, szóval jöjjön is egy példa, hogy mikor NEM jó.

Ilyen esetben - objektum metódusában - érdemes kerülni a használatát:

const sampleObject = {  
  a: 10,  
  b: () => console.log(this.a, this),  
  c: function() {  
    console.log(this.a, this);  
  }  
};  
  
sampleObject.b(); // undefined, Window {...} (vagy global object)  
sampleObject.c(); // 10, Object {...}

**Heap és stack**

Hol tárolódnak az objektumok?

Heap

Hol tárolódnak a primitívek?

Stack

[Egy hasznos magyarázó cikk a témában](https://felixgerschau.com/javascript-memory-management/)

Jegyzet – Metódusok (tömbök bevezetője után)

Metódusok

* forEach(): egy függvényt meghív a tömb minden elemére
* map(): egy új tömböt ad vissza, az eredeti tömb tartalmát módosítja a megadott függvény szerint
* filter(): egy új tömböt ad vissza, a feltételnek megfelelő elemeket válogatja, szűri ki
* reduce(): végigmegy a tömb elemein, azokon elvégzi a megadott műveletet, és EGY értékkel tér vissza
* flat(): egy új tömböt ad vissza, a többdimenziós tömböt a megadott mélységig „kilapítja”
* flatMap(): egy új tömböt ad vissza, az eredeti tömb tartalmát módosítja a megadott függvény szerint, és egy dimenziónyit „lapít” rajta

Jegyzet - Új metódusok, típusos tömbök (flatmap után)

Tömbök és egyéb új adatszerkezetek

**Array**

**Új tömbmetódusok**

A következő új metódusaink vannak:

* Array.from(): Tömböt készít egy iterálható objektumból.

const listElements = Array.from(document.querySelectorAll('li')).map(  
  element => element.textContent  
);

* find() - Visszaadja a tömb keresett elemét, ha nem találja, undefined-dal tér vissza.
* findIndex() - Visszaadja a tömb keresett elemének indexét, ha nem találja, -1-gyel tér vissza.
* some() - Megadja, hogy a tömb valamely elemére igaz-e a keresési feltétel.
* every() - Megadja, hogy a tömb összes elemére igaz-e a keresési feltétel.

const data = [  
  {  
    firstName: 'Kiss',  
    lastName: 'József',  
    age: 18  
  },  
  {  
    firstName: 'Horváth',  
    lastName: 'Péter',  
    age: 34  
  },  
  {  
    firstName: 'Kováts',  
    lastName: 'Ilona',  
    age: 24  
  },  
  {  
    firstName: 'Nagy',  
    lastName: 'Antal',  
    age: 66  
  }  
];  
  
console.log(data.find(item => item.age > 18)); // Object  
console.log(data.findIndex(item => item.age > 18)); // 1  
console.log(data.some(item => item.age > 18)); // true  
console.log(data.every(item => item.age > 18)); // false

**Típusos tömbök**

Lehetőségünk van típusos tömbök létrehozására. Tehát csak és kizárólag olyan elemeket tartalmazhatnak ezek a tömbök, amelyek adott típusba tartoznak. Ezek a következők:

* Int8Array() - 8 bites előjeles egész
* Uint8Array() - 8 bites előjel nélküli egész
* Uint8ClampedArray() - 8 bites előjel nélküli egész (a nem megfelelő értékeket módosítja a legkisebb/legnagyobb értékre)
* Int16Array() - 16 bites előjeles egész
* Uint16Array() - 16 bites előjel nélküli egész
* Int32Array() - 32 bites előjeles egész
* Uint32Array() - 32 bites előjel nélküli egész
* Float32Array() - 32 bites lebegőpontos
* Float64Array() - 64 bites lebegőpontos

**Példa:**

const intArr = new Int8Array(2);  
intArr[0] = 0;  
intArr[1] = 127;  
intArr[2] = 120;

JS2 - Jegyzet – Szövegek (bevezető után)

String-ek

Template literal

Ezidáig ha string literált és változókat akartunk összefűzni, a '+' jelet használtuk. Nincs is ezzel baj, hamar megszoktuk, még ha eleinte kicsit körülményesnek is találtuk. A jó hír, hogy már van egy szebb, jobb módszer. Ez pedig a template literal.

const firstName = 'John';

const lastName = 'Doe';

const age = 31;

// Régi módszer:

console.log(

'My name is ' +

   firstName +

   ' ' +

   lastName +

   ', and I'm ' +

   age +

   ' years old.'

);

// Template literal-lal

console.log(`My name is ${firstName} ${lastName}, and I'm ${age} years old.`);

A fenti példában láthatjuk a különbséget a régi és az új szintaxis között. Ugye, hogy szebb?

A template literal használatával az aposztróf/idézőjel helyett a tompa ékezetet (Alt+96) használjuk, és nem kell a '+' jellel összefűznünk, hanem helyette a ${} jelek közé illesztjük a változóinkat.

A következő példa azt reprezentálja, hogy hogyan tudunk egy tömbben lévő objektumokon végigmenni, és HTML template-et generálni (és lesz ez még szebb is) :

const people = [

{

   firstName: 'John',

   lastName: 'Doe'

},

{

   firstName: 'Jane',

   lastName: 'Doe'

}

];

const template = `<div>

           ${people

             .map(function(person) {

               return `<p> ${person.firstName} ${person.lastName} </p>`;

             })

             .join('')}

</div>`;

/\*

        <div>

            <p> John Doe </p>

            <p> Jane Doe </p>

        </div>

        \*/

console.log(template);

Tagged template literal

Nem csak template literal, hanem tagged template literal-ok is vannak már a JavaScriptben. Hogy ez mit is jelent? Azt, hogy a template-ünket át tudjuk adni egy feldolgozó függvénynek, amely segítségével tetszőleges módosításokat hajthatunk végre. Az újdonság az egészben, hogy nem hagyományos módon hívjuk meg a függvényt.

Nézzünk egy egyszerű példát:

const name = 'John Doe';

function tag(text, value) {

return `Szöveg: ${text[0]} Változó: ${value}`;

}

console.log(tag`Név ${name}`); // Szöveg: Név  Változó: John Doe

Jól látható, hogy a függvény után nincs zárójel. A függvényünknek két paramétere van. Az első egy tömb lesz, amelyben a template szövegrészei lesznek, a második pedig a template-ben lévő változó(ka)t tartalmazza. (Jelenleg csak 1-et.)

const person = 'John Doe';

function tag(text, value) {

return `Szöveg: ${text[0]} Változó: ${value}`;

}

// Szöveg: My name is  Változó: John Doe

console.log(tag`My name is ${person}`);

Nézzünk egy kicsit összetettebb példát:

const person = {

firstName: 'John',

lastName: 'Doe',

age: '31'

};

function tag(texts, ...values) {

let str = '';

texts.forEach(function(text, i) {

   str += `${text}${

     values[i] != undefined

        ? `<strong>${values[i].toUpperCase()}</strong>`

        : ''

   }`;

});

return str;

}

// My name is  JOHN DOE, and I'm 31 years old.

document.write(

tag`My name is ${person.firstName} ${person.lastName}, and I'm ${

   person.age

} years old.`

);

Mi is történik a fenti kódban? Az előző példához képest annyiban módosult, hogy egy objektum tulajdonságait íratjuk ki, és ezáltal több 'változónk' lesz, illetve több szövegrészletet is használunk.

A ...values egy úgynevezett spread operator. Ennek a segítségével - hasonlóan mint egy tömb vagy objektum esetében - végig tudunk iterálni a kapott értékeken (jelenleg az objektum három tulajdonságértékén).

A szövegrészeken végigmegyünk egy forEach-csel, ez egy tömb, könnyedén megtehetjük. Az str változóhoz folyamatosan hozzáfűzzük a tömb adott elemét és az adott indexű (ez az index az i paraméterben van) átadott értékét. A texts és a values elemeinek száma eltérő lehet, ezért használunk  feltételes értékadást. Azaz ha a values[i] nem undefined, akkor nagybetűssé alakítjuk és félkövérré tesszük.

Fontos megjegyezni, hogy adott esetben, ha például ilyen a template: ${a} ${b}, akkor a kettő között lévő whitespace is feldolgozásra kerül.

Multiline string

Végre nem kell többé szenvednünk többsoros szövegek létrehozásával. Vagyis nem annyit... Többsoros szöveget a következőképp tudunk létrehozni:

const multiline = `Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt  ut labore et

                    dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut

aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore

                    eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt

mollit anim id est laborum.`;

Arra kell figyelnünk, hogy ha konzolra íratunk ki, akkor olyan, mintha előreformázott szövegünk lenne. Tehát a sor elején lévő szóközök, tabok szintén megjelennek. Amennyiben ezt szeretnénk elkerülni, szedjük ki a sorok elejéről a felesleges space-eket, tabokat.

function translate(data) {

return data

   .split('\n')

   .map(function(line) {

     return line.replace(/^ +/, '');

   })

   .join('\n');

}

const multiline = `Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt  ut labore et

                    dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut

aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore

                    eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt

mollit anim id est laborum.`;

console.log(translate(multiline));

Ez egy egész szép megoldás, és az arrow function-ökkel lesz még szebb is.

Új string-metódusok

Az *ES6*-tal kaptunk néhány új string-metódust is. Ezek a következők:

* startsWith(searchString[, position]) : Megadja, hogy a string a keresett szövegrészlettel kezdődik-e. Megadhatjuk azt is, hogy honnan kezdje a keresést.
* endsWith(searchString[, length]): Megadja, hogy a string a keresett szövegrészlettel végződik-e. Megadhatjuk azt is, hogy meddig keressen.
* includes(searchString[, position]): Megadja, hogy a string a keresett szövegrészt tartalmazza-e. Megadhatjuk azt is, hogy honnan kezdje a keresést.
* repeat(count) : Egy adott szöveget count-szor ismételtet.

**Példák:**

console.log('hello'.startsWith('hell')); // true

console.log('hello'.endsWith('lo')); // true

console.log('hello'.includes('ll')); // true

console.log('hello'.repeat(5)); // hellohellohellohellohello

Unicode-karakterek

Szépen támogatottak:

console.log('𠮷 length: ', '𠮷'.length); // 𠮷 length:  2

console.log('\uD842\uDFB7'); // 𠮷 // hex

console.log('\u{20BB7}'); // 𠮷

console.log('𠮷'.codePointAt(0) == 0x20bb7); // true

ES8

String-ek

Két lényeges string-metódust kaptunk az **ES8**-cal. Ez pedig a padEnd() és a padStart(). Ennek a két metódusnak a segítségével egy szöveget tudunk fix hosszúságúra kiterjeszteni. Két paramétert vár, egy számot, amely kötelező, és a szöveg új hossza lesz, és egy másikat, amely opcionális: hogy milyen karakterrel/karakterekkel legyen feltöltve a szöveg. Amennyiben szükséges, a kitöltő karaktert ismétli is:

console.log('abc'.padEnd(10)); // 'abc       '

console.log('abc'.padEnd(10, 'foo')); // 'abcfoofoof'

console.log('abc'.padEnd(6, '123456')); // 'abc123'

console.log('abc'.padEnd(1)); // 'abc'

console.log('abc'.padStart(10)); // '       abc'

console.log('abc'.padStart(10, 'foo')); // 'foofoofabc'

console.log('abc'.padStart(6, '123456')); // '123abc'

console.log('abc'.padStart(1)); // 'abc'

Default parameter

Végre valahára a függvényparamétereknek adhatunk alapértelmezett értéket. Tehát nem kötelező az adott paraméter(ek) megadása. Régen a következőt csináltuk (csak azért van a példában var, hogy tényleg autentikus legyen a kód):

function defaultParameter(x, y, z) {  
  var y = y || 0;  
  var z = z || 20;  
  return x + y + z;  
}

Ez ugyan működött, de olyan esetekben, ha egy 0 értéket vagy akár üres string-et (vagy bármit, ami egy if esetében false értéket adna) kellett volna használnunk, akkor az alapértelmezett érték érvényesült. A fenti példa esetében:

defaultParameter(0,0,0) // 20

**NA DE MOST:**

function defaultParameter(x, y = 5, z = 42) {  
  return x + y + z;  
}

// És ez mind jó:

console.log(defaultParameter(1)); // 48  
console.log(defaultParameter(1, 2)); // 45  
console.log(defaultParameter(1, 2, 3)); // 6

Érdekességképp nézzük meg, mi lenne, ha Babellel transzponálnánk ezt a kódot. A következőt kapnánk:

'use strict';  
function defaultParameter(x) {  
    var y = arguments.length > 1 && arguments[1] !== undefined ? arguments[1] : 5;  
    var z = arguments.length > 2 && arguments[2] !== undefined ? arguments[2] : 42;  
    return x + y + z;  
}

**Spread operator**

*"Ennek a segítségével - hasonlóan, mint egy tömb vagy objektum esetében - végig tudunk iterálni a kapott értékeken."*

Mikor érdemes használni:

**Az apply nélküli függvények hívására**

var args = [0, 1, 2];  
function spreadOperator(x, y, z) {  
  return x + y + z;  
}

// ES6 előtt...

spreadOperator.apply(null, args);

// ...és most

spreadOperator(...args);

**Tömbök összefűzésére, másolására**

const firstArray = [1, 2, 3];  
const secondArray = [4, 5, 6];  
console.log([...firstArray, ...secondArray]); // [1, 2, 3, 4, 5, 6]

**Elemek slice-olására**

const str = 'hello';  
console.log([...str]); // ['h', 'e', 'l', 'l', 'o'], mint az str.slice()

**Rest parameter**

A rest paraméter hasonló, mint a spread operator, csak egy kicsit más szemszögből kell néznünk a dolgokat. A következő példa után egyértelmű lesz:

function restParameters(x, y, ...args) {  
  return (x + y) \* args.length;  
}  
  
console.log(restParameters(1, 2, 'hello', true, 5)); // 9

Tehát az első két paraméter után a többi már iterálható lesz, azaz tömbösítettük az argumentumokat.

Jegyzet – Számok (számok bevezetője után)

## Numbers

Az **ES6**-tal jött néhány újdonság a számok tekintetében is.

## Bináris és oktális literál

Bináris és oktális számok létrehozása egyszerűen:

const binary = 0b1111;

const octal = 0o77;

console.log(binary === 15); // true

console.log(octal === 63); // true

### Number.EPSILON

A Number.EPSILON az 1 és az egynél nagyobb legkisebb lebegőpontos szám közötti különbség. Az értéke: 2-52. A segítségével vizsgálhatjuk példaképp azt, hogy egy adott érték nagyobb-e, mint 0:

console.log(Math.abs(0.1 + 0.2 - 0.4) < Number.EPSILON); // false

console.log(Math.abs(0.1 + 0.2 - 0.3) < Number.EPSILON); // true

### Number.isFinite()

A Number.isFinite() metódus segítségével megvizsgálhatjuk, hogy egy adott szám véges-e:

console.log(Number.isFinite(Infinity)); // false

console.log(Number.isFinite(-Infinity)); // false

console.log(Number.isFinite(NaN)); // false

console.log(Number.isFinite(10)); // true

### Number.isNaN()

Az Number.isNaN() metódus segítségével megvizsgálhatjuk, hogy egy adott érték nem szám-e. A Number.isNaN() nem összekeverendő a globális isNaN() metódussal. Hasonló, de mégis más. Nézzük meg, mit is ír a szabvány a [globális isNaN()](https://www.ecma-international.org/ecma-262/#sec-global-object) metódusról és a [Number.isNaN()](https://www.ecma-international.org/ecma-262/" \l "sec-isnan-number)()-ról. Tehát a fő különbség, hogy a Number.isNaN() nem végez típuskonverziót az összehasonlítás előtt.

console.log(Number.isNaN(true)); // false

console.log(Number.isNaN('hello')); // false

console.log(Number.isNaN(12)); // false

console.log(Number.isNaN(NaN)); // true

console.log(Number.isNaN({})); // false

console.log(Number.isNaN('NaN')); // false

## ES7

Néhány dolog a 2016-os szabványból.

Az **ES7**-ben behoztak egy új tömbmetódust, ez pedig az Array.prototype.includes. Hasonlóan működik, mint az indexOf(), a különbség a NaN értéknél figyelhető meg. Az indexOf() esetében -1 lesz a visszatérési értékünk, ha NaN értéket keresünk. Ez azért történik, mert egy strict equality comparison, egy szigorú egyenlőségi összehasonlítás történik a következőképp: [NaN].indexOf(NaN) === -1. Az isNaN() az értéket számmá konvertálja, ebből kifolyólag lesz a -1 a visszatérési érték.

**Figyelem**: Ne keverjük a globális isNaN()-t a Number.isNaN()-nal! Ez sok esetben igencsak kellemetlen, és egy plusz feltételvizsgálatot is használnunk kell, ha NaN értékre akartunk szűrni.

Az includes() tökéletes megoldást nyújt erre a problémára, mert a SameValueZero összehasonlítást használja a strict equality comparison helyett. [Dokumentáció](https://developer.mozilla.org/hu/docs/Web/JavaScript/Equality_comparisons_and_sameness)

// Array.prototype.includes

let arr = [1, 2, 3, 4, 5];

console.log(arr.includes(1)); // true

console.log(arr.includes(10)); // false

console.log([NaN].includes(NaN)); // true

console.log([NaN].indexOf(NaN)); // -1

### Hatványozás egyszerűbben

Bevezették a \*\* operátort. Így már sokkal egyszerűbb a hatványozás művelete. Többé nem kell a Math.pow()-t használnunk.

console.log(2 \*\* 10); // 1024

### Number.isSafeInteger()

A Number.MIN\_SAFE\_INTEGER a legkisebb, a Number.MAX\_SAFE\_INTEGER konstans a legnagyobb biztonságosan használható egész számot adja meg. Ez alatt/felett már nem biztos, hogy a számításaink eredményei vagy az összehasonlítások pontosak/megbízhatóak lesznek.

A „biztonságos” szó arra utal, hogy a matematikai egészek hogyan jelennek meg JavaScriptben. A -253és a 253 értékek közé eső értékek biztonságosak, mert a Javascript 64 biten, előjelesen ábrázolja az egész számokat (1 bit – előjelbit, 11 bit – karakterisztika, 52(+1) bit - mantissza). Ezeken a határokon kívül az értékek „nem biztonságosak”, mert egynél több matematikai szám is jelentheti ugyanazt az értéket JavaScriptben. Például 253 felett csak minden második egész értéket tudja ábrázolni. Tehát biztonságos az a JavaScript egész, amely egyetlen matematikai egész számot jelöl.

Az isSafeInteger() megmondja egy számról, hogy a biztonságos zónába tartozik-e.

console.log('MinSafeInteger:', Number.MIN\_SAFE\_INTEGER); // -9007199254740991

console.log('MaxSafeInteger:', Number.MAX\_SAFE\_INTEGER); // 9007199254740991

console.log(Number.MIN\_SAFE\_INTEGER + 1 == Number.MIN\_SAFE\_INTEGER + 2); // false

console.log(Number.isSafeInteger(42)); // true

console.log(Number.isSafeInteger(9007199254740992)); // false

### Math.sign()

Egy szám előjelének vizsgálatára használjuk:

console.log(Math.sign(10)); // 1

console.log(Math.sign(0)); // 0

console.log(Math.sign(-0)); // -0

console.log(Math.sign(-10)); // -1

console.log(Math.sign(NaN)); // NaN

### Math.trunc()

A Math.trunc() segítségével egy lebegőpontos szám egész részét tudjuk lekérdezni:

console.log(Math.trunc(30.17)); // 30

console.log(Math.trunc('-1.123')); // -1

console.log(Math.trunc(0.11)); // 0

console.log(Math.trunc(-0.11)); // -0

### BigInt

A 2^53 - 1-nél (Number.MAX\_SAFE\_INTEGER) nagyobb egész számokat jelöli. Jele "n" a szám után.

const max = Number.MAX\_VALUE;  
console.log(max \*\* 2);  
const bigIntNumber = BigInt(max);  
console.log(bigIntNumber \*\* 2n);

Jegyzet - Adatok átstrukturálása (átstrukturálás bevezetője után)

# Destructuring data

A legtöbb újdonságon túl vagyunk, de még mindig messze a vége. Most megnézzük, hogyan tudunk adatokat átstrukturálni.

## Assignment separate from declaration

Deklaráció és inicializáció szétválasztása több változó esetén:

let a, b;

[a, b] = [1, 2];

console.log(a); // 1

console.log(b); // 2

## Default values

Változók alapértelmezett értékei:

let a, b;

[a = 5, b = 7] = [1];

console.log(a); // 1

console.log(b); // 7

## Swap data

A különböző cserés rendezésekhez már nem kell egy plusz változót felvennünk. Lehetőségünk van az értékek megcserélésére:

let a = 'A értéke';

let b = 'B értéke';

[a, b] = [b, a];

console.log(a); // B értéke

console.log(b); // A értéke

## Destructuring arrays

Legyen egy tömbünk, amely X elemet tartalmaz. Ezután szeretnénk a tömbben tárolt értékeket egy-egy változóban külön tárolni. Egy ilyen zseniálisan egyszerű dolgot tehetünk:

const foo = ['one', 'two', 'three'];

const [one, two, three] = foo;

console.log(one); // 'one'

console.log(two); // 'two'

console.log(three); // 'three'

## Destructuring objects

Egy kicsit dolgozzunk objektumokkal is. Első körben azt nézzük meg, hogy a tömbökhöz hasonlóan hogyan tudjuk az előbb tanultakat egy objektum esetében is használni. Első körben az objektumunk két tulajdonságát tároljuk egy-egy konstansban.

const person = {

firstName: 'John',

lastName: 'Doe',

age: 31

};

/\*  FIGYELEM!: itt a person objektumunk firstName és lastName tulajdonságát kérjük le.

    Nem írhatjuk azt, hogy:

const {

        a,

        b

    } = person;

    mert a personnek nincs a és b tulajdonsága.

\*/

const { firstName, lastName } = person;

console.log(`${firstName} ${lastName}`); // John Doe

Ezzel természetesen nincs vége. A fenti kódnál azt mondtam, hogy nem írhatunk akármilyen nevet az új struktúrába, arra azonban van mód, hogy 'alias'-t készítsünk, azaz megadhatjuk, hogy az adott attribútumokra milyen névvel hivatkozunk:

const person = {

firstName: 'John',

lastName: 'Doe',

age: 31

};

const { firstName: f, lastName: l } = person;

console.log(`${f} ${l}`); // John Doe

Még egy utolsó dolog. Pontosabban utolsó kettő. Az egyik, hogy használhatunk default value-kat:

const styles = {

align: 'left',

background: 'white',

color: 'red',

display: 'none'

};

const {

  align = 'right',

  background,

  color = 'blue',

  display,

  visibility

} = styles;

console.log(align, background, color, display, visibility); // left white red none undefined

Az utolsó pedig, hogy a default value-kat és az alias-okat mixelhetjük is:

const styles = {

align: 'left',

background: 'white',

color: 'red',

display: 'none'

};

const {

align: a = 'right',

background: b,

color: c = 'blue',

display: d,

visibility: v

} = styles;

console.log(a, b, c, d, v); // left white red none undefined

## Function destructuring

Függvények esetében, ha például a visszatérési érték egy objektum, akkor ugyanazt a módszert tudjuk használni, mint az objektumoknál. A következő példában egy egyszerű mértékegység-átváltó függvényt láthatunk, amely a métert angolszász hosszmértékegységekre konvertálja:

function convertLength(meter) {

const british = {

   inch: (meter \* 100 / 2.54).toFixed(2),

   feet: (meter \* 100 / 2.54 / 12).toFixed(2),

   yard: (meter \* 100 / 2.54 / 36).toFixed(2),

   mile: (meter / 1609.344).toFixed(2)

};

return british;

}

const { feet, yard } = convertLength(1);

console.log(`Feet: ${feet}, Yard: ${yard}`); // Feet: 3.28, Yard: 1.09

A következő példa első ránézésre egy kicsit furcsának tűnhet, pedig nem az.

function priceCalc({ price = 100, tax = 27, inward = 4 } = {}) {

return price \* (1 + tax / 100 + inward / 100);

}

const money = priceCalc();

console.log(money);

Nézzük meg lépésről lépésre!

Elsőre 3 paraméterünk lesz, mindegyik kap egy default value-t:

function priceCalc(price = 100, tax = 27, inward = 4) {

return price \* (1 + tax / 100 + inward / 100);

}

const money = priceCalc();

console.log(money); // 131

Ha objektumként szeretnénk megadni; akkor kezdődnek a problémák, ha meghívjuk a függvényt paraméter nélkül:

function priceCalc({ price = 100, tax = 27, inward = 4 }) {

return price \* (1 + tax / 100 + inward / 100);

}

priceCalc();

Rögtön megkapjuk, hogy Cannot destructure property 'price' of 'undefined' or 'null'. Hogy kiküszöböljük ezt a problémát, tehát hogy megmaradjon az object és használjuk a default value-kat is, ezért néz ki ez a rész így: {price = 100, tax = 27, inward = 4} = {}, mert így szépen egyedileg elérjük mindegyik értéket, és meglesz az alapértelmezett érték is.

A {} egy üres objektum literál. Ha ezt elhagyjuk, a függvény legalább egy argumentumot fog keresni. Tehát abban az esetben, ha elképzelhető, hogy nem adunk meg paramétert, akkor használnunk kell a jobb oldali {}-t.

Jegyzet – Dátumok (dátumok bevezetője után)

## Dátumok

Date objektum létrehozása, kötelező a new-t használni:

const dateNow = new Date();

console.log(dateNow); // Mon Oct 04 2021 14:17:06 GMT+0200 (közép-európai nyári idő)

A fenti módon egyedi, saját magunk által megadott dátumokat is tárolhatunk változókban.

### Date metódusai

Az új Date object-et tároló változónak az alábbi metódusait tudjuk meghívni:

console.log(dateNow.toJSON()); // 2021-10-04T12:17:06.144Z

console.log(dateNow.toDateString()); // Mon Oct 04 2021

console.log(dateNow.toLocaleDateString('hu')); // 2021. 10. 04.

console.log(dateNow.toLocaleDateString('it')); // 4/10/2021

console.log(dateNow.toTimeString()); // 14:17:06 GMT+0200 (közép-európai nyári idő)

console.log(dateNow.toLocaleTimeString('hu')); // 14:17:06

console.log(dateNow.toLocaleTimeString('en')); // 2:17:06 PM

console.log(Date.now()); // 1633350223465 – az 1970. 01. 01. óta eltelt idő milliszekundumban

### A get… metódusok

const dateNow = new Date();

console.log('year: ', dateNow.getFullYear());

console.log('month: ', dateNow.getMonth());

console.log('day of the week: ', dateNow.getDay());

console.log('day of the month: ', dateNow.getDate());

console.log('hours: ', dateNow.getHours());

console.log('minutes: ', dateNow.getMinutes());

console.log('seconds: ', dateNow.getSeconds());

console.log('milliseconds: ', dateNow.getMilliseconds());

### A getUTC… metódusok

const dateNow = new Date();

console.log('UTC year: ', dateNow.getUTCFullYear());

console.log('UTC month: ', dateNow.getUTCMonth());

console.log('UTC day of the week: ', dateNow.getUTCDay());

console.log('UTC day of the month: ', dateNow.getUTCDate());

console.log('UTC hours: ', dateNow.getUTCHours());

console.log('UTC minutes: ', dateNow.getUTCMinutes());

console.log('UTC seconds: ', dateNow.getUTCSeconds());

console.log('UTC milliseconds: ', dateNow.getUTCMilliseconds());

### A set… metódusok

const myDate = new Date();

myDate.setFullYear(2022);

myDate.setMonth(10);

myDate.setDate(10);

myDate.setHours(10);

myDate.setMinutes(10);

myDate.setSeconds(10);

myDate.setMilliseconds(10);

console.log(myDate);