A JavaScript programozási nyelv

Webprogramozás – 4. előadás

Sulyok Csaba
csaba.sulyok@gmail.com

1. rész

JavaScript: Bevezető

JavaScript

- a JavaScript a *Web*en használt legnépszerűbb szkriptelési nyelv
- az elterjedtebb böngészők mind ismerik (Internet Explorer, Mozilla, Firefox, Chrome, Opera, Edge)
- **értelmezett** (interpreter) nyelv - a kód értelmezése dinamikusan vagy JIT (Just-In-Time) történik
- a HTML oldalak készítői számára:
  - egy programozási eszközt biztosít (használata egyszerű)
  - módosíthatja a HTML tartalmát, kinézetét (a **HTML DOM** objektumaihoz való hozzáférés által)
  - **események**re tud reagálni (pl. oldal betöltése, kattintás egy elemre, stb.)
  - a bevitt adat helyességének ellenőrzésére ad lehetőséget (mielőtt elküldenénk a szerverre)
  - sütiket (cookies) hozhatunk létre a kliens gépén való információ-tárolás érdekében
  - megvizsgálhatjuk a böngésző típusát, és ennek függvényében más-más, böngésző-specifikus tartalmat tölthetünk be

JavaScript

- a Netscape vezette be (kezdetben *LiveScript* néven)
- minden böngésző külön írja meg a JavaScript interpreter motorját – ezek közül az egyik legismertebb a Google Chrome által használt **V8**
- noha szintaxisa hasonlít a Java programozási nyelvéhez, nincs köze hozzá
- használható mint szekvenciális vagy deklaratív (funkcionális) nyelv
- hasznos linkek:
  - `http://javascript.info/`
  - `https://www.quackit.com/javascript/examples/`
  - `https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/JavaScript_basics`

ECMAScript

- közös hivatalos szabvány: **ECMAScript** (ECMA-262 - standard)
- minden *modern* böngésző JavaScript motra támogatja legalább az ECMAScript 5.1-es szabványt
- 2015: megjelenik az **ECMAScript 6** vagy **ES2015**
  - 2017 augusztusától minden modern böngésző támogatja (az Internet Explorer azóta *nem* tartozik bele a "modern" kategóriába)
  - a **laborfeladatoknál elvárt a betartása**
- az ES2015-tel kezdődően évente frissítik a szabványt; a legfrissebb az *ES2018*
  - verziók leírása és böngészői támogatás
  - hivatalosan már csak az évszámból származó nevet alkalmazzák, de a közösség informálisan folytatja a számozást (ES2018 ⇔ ES9)
  - legtöbb böngésző még nem támogatja az újabb verziókat (az ES2016-ot is csak a Chrome és az Opera támogatja)
  - ha böngészőben futtatnák újabb ECMAScript kódot, **transzpilálás** szükséges (*source-to-source compilation*) - ehhez ajánlott a Babel

JavaScript használata parancssorból

- A JavaScript klasszikusan kliensoldali nyelv ⇒ általában a böngészőben fut
- A node.js jóvoltából a JavaScript futtatható **böngészőn kívül is**.
- A **node.js** a Google V8-as JavaScript értelmező motra elválasztva a böngészőtől. Megengedi a JS kód futtatását parancssorból mind állományokból, mind interaktív konzolon keresztül.
- Telepítés utáni használat: `node állomány.js`
- A `node` parancssori argumentum nélküli meghívása egy interaktív konzolt eredményez - neve **REPL** (Read-Eval-Print-Loop).
- Gyors ECMAScript támogatás: `https://node.green/` (az ECMAScript 2019 szabvány még nem finalizált, de a node 12.0.0-s verziója már mindegyik indítványát támogatja)
- A gyors és gyakori változások miatt ajánlott a legújabb **LTS** (long-term support) verzió használata

2. rész

JS szintaxis

JavaScript szintaxis

- Java-hoz hasonló részek:
  - aritmetikai és logikai operátorok
  - feltételes és ismétlő struktúrák: `if`, `if else`, `while`, `switch`, `for`, `while`, `do while`
  - vezérlési utasítások: `return`, `break`, `continue`
  - egysoros és többsoros kommentek: `// ...`, `/* ... */`
- konzolra való kiírás: `console.log(...);`, `console.error(...);`
- a pontosvesszők használata sorok végén nem kötelező, de ajánlott
- **változók**
  - a JavaScript **nem** erősen típusos: nem adunk típust a változóinknak, s típusaik dinamikusan változhatnak a kód futása közben
  - a változók deklarálása nem kötelező, de ajánlott; kulcsszavak:
    - `var` - függvény hatáskörű
    - `let` (`ES6`-tól) - blokk hatáskörű
    - `const` (`ES6`-tól) - mint `let`, de nem változtathatja értékét első megadás után
  - a JavaScript különbséget tesz kis- és nagybetűk közt
  - elterjedt elnevezési konvenció: **camel case** (pl. `newExchgRate`)
  - a változók neve betűvel vagy az _ karakterrel kell kezdődjön

Adattípusok

- Minden változónak van egy hozzárendelt típusa, amely változhat az életciklusa során. A három legalapvetőbb típus:
  - **String** - karaktersorozatok
  - **Number** - számok, *lebegőpontos számokat* is beleértve. speciális érték: *NaN*
  - **Boolean** - *true* vagy *false*
- Két (primitív) adattípus, amely nem tárol információt, csupán informatív jellegű:
  - **Null** - objektum, értéke semmi
  - **Undefined** - nem értelmezett
- Összetett típusok: **Object**, **Array**, **Date**
- Függvény típus: **function**

Adattípusok és kiírás

- kiírás string-konkatenálással:

```
console.log("value: " + s + "; type: " + typeof(s));
```

- kiírás **sablonliterállal** (template literal; ES6-tól):

```
console.log(`value: ${s}; type: ${typeof(s)}`);
```

- példa változó típusokra:

```
let s = 42;
console.log(`value: ${s}; type: ${typeof(s)}`); // value: 42; type: number
s = "abc";
console.log(`value: ${s}; type: ${typeof(s)}`); // value: abc; type: string
s = false;
console.log(`value: ${s}; type: ${typeof(s)}`); // value: false; type: boolean
```

Típuskonverzió

- Típusok közötti konverzió a legtöbb esetben automatikusan történik. Nem biztos, hogy az eredmény az elvárt lesz.
- Explicit konverzió, pl: `parseInt(); parseFloat(); Number(); String(); Boolean()` függvényekkel
- *Változók típusának ellenőrzése:*
  - `instanceof` operátorral:

    ```
    if (x instanceof Number) {...}
    ```

  - `typeof` operátorral, pl. `if(typeof x === "number") {...}`

    ```
    if (typeof x === "number") {...}
    ```

  - egy változó `.constructor` mezőjének segítségével, pl.

    ```
    if (x.constructor === Number) {...}
    ```

- *Egyenlőség operátorok:*
  - `==` operátor: a két operandus értéke megegyezik
  - `===` operátor (*identical to*): a két operandus értéke *és típusa* megegyezik (használata ajánlott)
  - `.is(...)` metódus (ES6-tól)

JavaScript beépített objektumok

- `String` - minden `string` típusú változó tulajdonképpen `String` típusú objektum (az alábbi mezők és metódusok a `String` prototípus részei)
  - mező: `length`
  - fontosabb metódusok: `toUpperCase`, `toLowerCase`, `replace`, `substring`, `slice`, `charAt`, `indexOf`, stb.
- `Date` - dátum/idő beállítás/lekérdezés
  - létrehozás:

```
birthday = new Date();
birthday = new Date("June 20, 1996 08:00:00");
```

  - metódusok: `set...()`/`get...()`: `Date`, `Year`, `Month Day`, `Time`, `Hours`, `Minutes`, `Seconds`
- `Math` - automatikusan létre van hozva (explicit `Math` objektum létrehozása nélkül használható)
  - metódusok:
    - kerekítés: `ceil()`, `floor()`, `round()`;
    - véletlenszám generálás (0 és 1 között): `random()`

JavaScript függvények

```
function fuggvenynev(valtozo1, valtozo2,...) {
   // ...
   return ertek;  // nem kotelezo
}
```

- meghívhatóak a HTML oldal bármely pontjáról, vagy egy esemény bekövetkeztekor
- a JavaScript úgy kezeli a függvényeket mint `function` típusú változókat:

```
function peldaFuggveny(param1, param2) {
    console.log(`param1=${param1}, param2=${param2}`);
}
console.log(typeof peldaFuggveny); // kimenet: "function"
```

- deklarálhatunk függvényeket változóként is, pl. a következő egyenértékű:

```
let peldaFuggveny = function(param1, param2) {
    console.log(`param1=${param1}, param2=${param2}`);
};
```

- a függvényen belül (`var`/`let` kulcsszóval) deklarált változók lokálisak (a *nem deklarált* változók **globálisak**)

JavaScript függvények

- ES6-tól kezdve paramétereknek lehet *alapértelmezett értéket* megadni, valamint kulcs-érték párosokkal meghívni
- régi variáns:

```
function peldaFuggveny(param1, param2) {
    if (arguments.length == 0) return false;
    // old
    if (param1 === undefined) {
        param1 = "alapérték";
    }
    // new
    param2 = param2 || 42;
    // logika...
}
```

- új (ajánlott):

```
function peldaFuggveny(param1="alapérték", param2=42) {
    console.log(`param1=${param1}, param2=${param2}`);
}

peldaFuggveny();                        // param1=alapérték, param2=42
peldaFuggveny(param1="fölülírtérték"); // param1=fölülírtérték, param2=42
peldaFuggveny("fölülírtérték2");       // param1=fölülírtérték2, param2=42
```

JavaScript arrow függvények

- ES6-tól
- rövidített függvénydeklarációs séma
- `(param1, param2, ...) => expression`
- példa:

```
// ES5
var multiplyES5 = function(x, y) {
    return x * y;
};

// ES6
const multiplyES6 = (x, y) => { return x * y };
```

- 1 paraméter esetén a `()` zárójelek nem szükségesek

```
const sqr = x => { return x * x; };
```

- 0 paraméter esetén `() => { ... }`:

```
const logHello = () => { console.log("Hello"); };
```

- 1 kifejezés esetén a kapcsoszárójel és `return` kulcsszó elhagyható

```
const multiplyES6 = (x, y) => x * y;
```

JavaScript szintaxis átfogó példa

```
// single line comment

/*
multi-line comment
*/

const a = 21 + 21; // semicolon not mandatory but recommended
console.log(`a = ${a}`);

if (a === 42) { // strict equals operator (also checks type)
    console.log(`a = ${a}`);
} else {
    console.error("Error: a is not set correctly");
}

function fact(x) {
    let fx = 1;
    for (let i = 2; i < x; ++i) {
        fx *= i;
    }
    return fx;
}
console.log(`10! = ${fact(10)}`); // calling function
```

JavaScript tömbök

- literálok (egyszerűbb, preferált):

```
let tomb1 = ["hetfo", "kedd", "szerda"];
let tomb2 = []; tomb2[0] = "egy";
```

- az `Array` beépített objektum segítségével:

```
let tomb3 = new Array(); tomb3[0] = "paros";
let tomb4 = new Array("paros", "paratlan");
```

- új elemek hozzáadása:

```
tomb1[3] = "csutortok";
tomb1[tomb1.length] = "pentek";
tomb1.push("szombat");
```

- a JavaScipt tömbök tulajdonképpen indexekkel ellátott, mezőkkel rendelkező *objektumok*ként vannak tárolva
- az `Array` objektum mezői, metódusai:
  - `length` mező (pl. `tomb1.length`)
  - fontosabb metódusok: `push`, `pop`, `splice`, `concat`, `join`
  - fontosabb függvény paraméterű metódusok: `forEach`, `filter`, `every`

JavaScript tömbök

```
// create empty array
const array1 = [];
array1.push("stringValue");
array1.push(42);

// create populated array
const array2 = ["stringValue", 42];

// access elements
console.log(array1);          // [ "stringValue", 42 ]
console.log(array1[0]);       // stringValue
console.log(array1[4]);       // undefined
console.log(array1.length);   // 2

// nest lists
array2.push(array1);
console.log(array2);          // [ "stringValue", 42, [ "stringValue", 42 ] ]

// removing elements
const oldElement = array2.pop();
console.log(array2);          // [ "stringValue", 42 ]
```

JavaScript tömbök

- *iterálás* klasszikus `for`-ral:

```
for (let i = 0; i < array1.length; ++i) {
    console.log(`array[${i}] = ${array1[i]}`);
}
```

- *iterálás* a `for in` operátorral:

```
for (const i in array1) {
    console.log(`array[${i}] = ${array1[i]}`);
}
```

- *iterálás* `forEach` függvény paraméterű *prototype* metódussal; ekvivalens, de csak ES6-tól érvényes:

```
// ES6 iteration with function parameter
array1.forEach((item, index) => {
    console.log(`array[${index}] = ${item}`);
});
```

- számos hasznos függvény paraméterű metódus, pl. *szűrés* a `filter`-rel:

```
const combinedArray = ["str1", 42, "str2", false];
// filter for string types
const stringArray = combinedArray.filter(item => typeof item === "string");
console.log(stringArray); // [ "str1", "str2" ]
```

JavaScript objektumok

- kulcs-érték párosok tárolója, ahol az értékek lehetnek további objektumok (komplex struktúra)
- általános objektum létrehozása:
  - `let obj = new Object();` - létrejön egy üres objektum
  - egyszerűbb és ajánlott a literál jelölés: `let obj = {};`
- a mezőket hozzáadhatjuk menet közben; ha eddig nem volt ilyen mező, érték-hozzárendeléskor automatikusan létre lesz hozva:

```
obj.nev="One Eyed Jack";
obj.eletkor=20;
```

- a mezőkre az alábbi szintaxissal is hivakozhatunk (*tömb jelleg*):

```
obj["nev"]="One Eyed Jack";
```

```
- *előny:* a szögletes zárójelen belül használhatunk kifejezést is
```

JavaScript objektumok

- az objektumok tartalmazhatnak függvény típusú mezőket – ezek a **metódusok**
- hozzárendelhető direkt módon (deklarációkor):

```
obj.hanyEves = function() {
    alert(`${this.nev} ${this.eletkor} éves`);
}
```

- előzőleg definiált függvény hozzárendelése:

```
function koszon() {
    alert(`Szia ${this.eletkor}`);
}
// ...
obj.koszon = koszon;
```

- **Vigyázat**: nem `obj.koszon = koszon()`, mivel a `(...)` operátor meghívja a függvényt, s a visszatérítési érték kerül az objektumba.

JavaScript objektumok

```
// create empty object
const obj1 = {};
// set some key/value pairs
obj1.key1 = "value1";
obj1["key2"] = 41;
obj1["key2"] = 42;

// create populated object
const obj2 = {
    key1: "value1",
    key2: 42
};

// access object elements
console.log(obj1);
    // { key1: "value1", key2: 42 }
console.log(obj1.key1);      // value1
console.log(obj1["key2"]);   // 42
```

```
// iteration
for (const key in obj2) {
    console.log(`obj2[${key}]=${obj2[key]}`);
}

// iteration using forEach
Object.keys(obj2).forEach((key) => {
    console.log(`obj2[${key}]=${obj2[key]}`);
});

// remove element
delete obj1.key1;
console.log(obj1); // { key2: 42 }
console.log("key1" in obj1); // false
```

JavaScript literálok

- komplex objektum kézzel való felépítése:

```
let verneGyula = new Object();
verneGyula.name = "Jules Verne";
verneGyula.foglalkozas = "ifjúsági regényíró";

let myLibrary = new Object();
myLibrary.books = new Array();
myLibrary.books[0] = new Object();
myLibrary.books[0].title = "Kétévi vakáció";
myLibrary.books[0].authors = new Array();
myLibrary.books[0].authors[0] = verneGyula;
```

- bonyolultabb struktúra felépítése kissé nehézkes
- gyorsabb megoldás: JavaScript tömb- és objektumliterálok

JavaScript literálok

- tömb esetén:

```
myList = [elem1, elem2, elem3];
```

- objektum esetén:

```
myObject = { kulcs1: ertek1, kulcs2: ertek2 };
```

- korábbi példa:

```
const myLibrary = {
    books: [{
        title: "Kétévi vakáció",
        authors: [{
            name: "Jules Verne",
            foglalkozas: "ifjúsági regényíró"
        }]
    }]
};
```

- komplex objektumhierarchia építhető fel ilyen módon
- a mezőknek való értékadáskor kifejezést is haszálhatunk (dinamikus tartalom)

Függvénymező megadása egy objektumnak

```
const barkacskonyv = {
    cim: `Csinald magad`,
    szerzok: [
        { nev: "Valaki", ev: 25 },
        { nev: "Barki", ev: 50 }
    ],
    megj: function(hossz=10) {
        console.log(`${this.cim}, ${this.szerzok[0].nev} konyve n${"a".repeat(hossz)}gyon unalmas`);
    }
};

// ...
barkacskonyv.megj(5); // kimenet: Csinald magad, Valaki konyve naaaaagyon unalmas
```

- a literál alapú objektumjelölés, illetve egyszerű JavaScipt használata kiegészíthetik egymást

3. rész

JS/ES modulok

ES modulok

- Az ECMAScript 2015-ös verziója előtt nem lehetett egy JS állományba **beimportálni** egy másikat
- Az akkor javasolt megoldás az ECMAScript **modulok**, melyekkel belső vagy külső JS állományokat lehet importálni, ahonnan látszanak az *exportált* függvények-változók
- Egy modulban az *export* kulcsszóval lehet megadni, hogy mely értékek legyenek láthatóak kívülről
- A használó modulban az *import* kulcsszóval lehet őket elérni
- Lazy loadingot használ, s egy modult csak egyszer tölt be, még ha több helyről is van importálva

ES modulok

Saját függvénykönyvtárak

es2015module_default.js:

```
// alapértelmezett export
export default function libFunction() {
  console.log('I am libfunction');
}
```

es2015module_multi.js:

```
export function libFunction1() {
  console.log('I am libfunction1');
}

export const libFunction2 = () => {
  console.log('I am libfunction2');
}
```

Főállomány index.js:

```
// belső (relatív útvonalon elhelyezkedő)
// modul betöltése
import libFunction from './es2015module_default.js';
import { libFunction1 } from './es2015module_multi.js';

// az összes exportált függvény
// közös néven való betöltése
import * as libMulti from './es2015module_multi.js';

// külső modul betöltése URL-lel
import mustache from 'https://.../mustache.min.js';

// függvények használata
libFunction();
libFunction1();
libMulti.libFunction2();
console.log(mustache.escape('<b>hello</b>'));
```

4. rész

JSON

A JSON leírónyelv

- **JSON** – *JavaScript Object Notation*
- adatleíró, szövegalapú nyelvezet, amely hasonló célokat szolgál, mint az XML
- a JavaScript tömb- és objektumliterálokon *alapszik*, de szintaxisa **kissé különböző**, ne keverjük a kettőt
- **különbségek**:
  - a JSON nem tartalmaz kódot vagy dinamikusan kiértékelendő kifejezéseket
  - a JSON kódban csak a `"` idézőjel használató
  - a JSON kódban minden kulcsot idézőjel közé kell zárni, nem csak a string értékeket
- JavaScriptben objektumok és JSON stringek közötti átalakítás: `JSON.parse(jsonStr)`, `JSON.stringify(obj)`
- MIME type-ja `application/json`, noha a JSON stringek nem tartalmaznak futtatható kódot
- komplex objektumokat gyakran küldünk ezen formátumban hálózaton keresztül, mivel kompaktabb az XML-nél

A JSON leírónyelv

```
// complex object, NOT JSON
const origLibrary = {
    books: [{
        title: "Kétévi vakáció",
        authors: [{
            name: "Jules Verne",
            foglalkozas: "ifjúsági regényíró"
        }]
    }]
};

// convert to JSON string
const jsonLibrary = JSON.stringify(origLibrary);
// re-parse to JavaScript object
const parsedLibrary = JSON.parse(jsonLibrary);

console.log(typeof origLibrary);   // object
console.log(typeof jsonLibrary);   // string
console.log(typeof parsedLibrary); // object
```

- a `jsonLibrary` string értéke:

```
{"books":[{"title":"Kétévi vakáció","authors":[{"name":"Jules Verne","foglalkozas":"ifjúsági regényíró"}]}]}
```