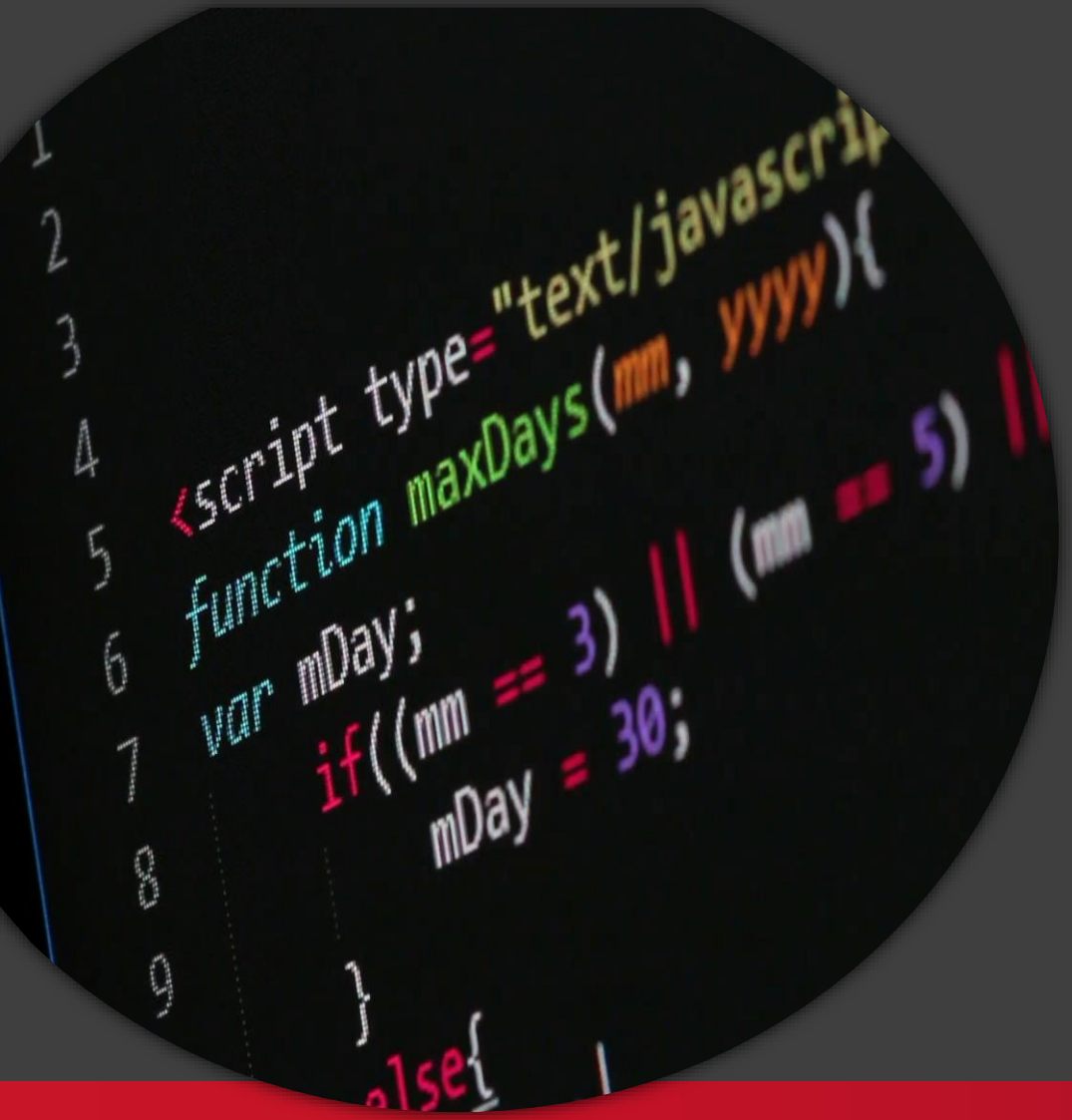


Bevezetés a JavaScriptbe

Kliensalkalmazások

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Gincsai Gábor
gincsai@aut.bme.hu

Brendan Eich

Java és JavaScript

- Brendan Eich, a Netscape mérnöke alkotta meg.
  - Mocha, majd LiveScript néven 1995-ben.
  - 1995 decemberben a Netscape és a Sun licenc megállapodás, azóta JavaScriptnek hívják.
  - A „JavaScript” ma az Oracle bejegyzett védjegye.
- Az eredeti cél egy egyszerű, nem csak hivatásos fejlesztők által használható szkript nyelv megalkotása volt, amely a Java nyelvet kiegészítve lehetővé teszi interaktív weboldalak készítését.

A JS egy szkriptnyelv

- A JS motor sorról sorra olvassa be és hajtja végre az utasításokat
- Hiba esetén a végrehajtás megáll és a további utasítások nem hajtódnak végre
- Nincs előzetes fordítás (szintaktikai és szemantikai ellenőrzés)
- Például:

```
Console.log("Sziaszok!");
asdasdasda
```

  - A fájlt beolvassuk, az első sort végrehajtjuk, ezután lesz csak hiba.

JavaScript változatai

- JScript (1996 augusztus)
  - Microsoft által a jogi problémák elkerülésére más néven kiadott dialektus.
  - Az Internet Explorer 9-ben található JScript 9.0 a JavaScript 1.8.1 és az ECMA-262 5. változatával kompatibilis.
- ECMAScript (1 változat 1996 június)
  - ECMA által az ECMA-262 szám alatt szabványosított változat
  - Mind a JavaScript, mind pedig a JScript a szabványhoz képest további funkciókat biztosít.
- Jelenleg évente készül új verzió. Pl: ECMAScript 2020.

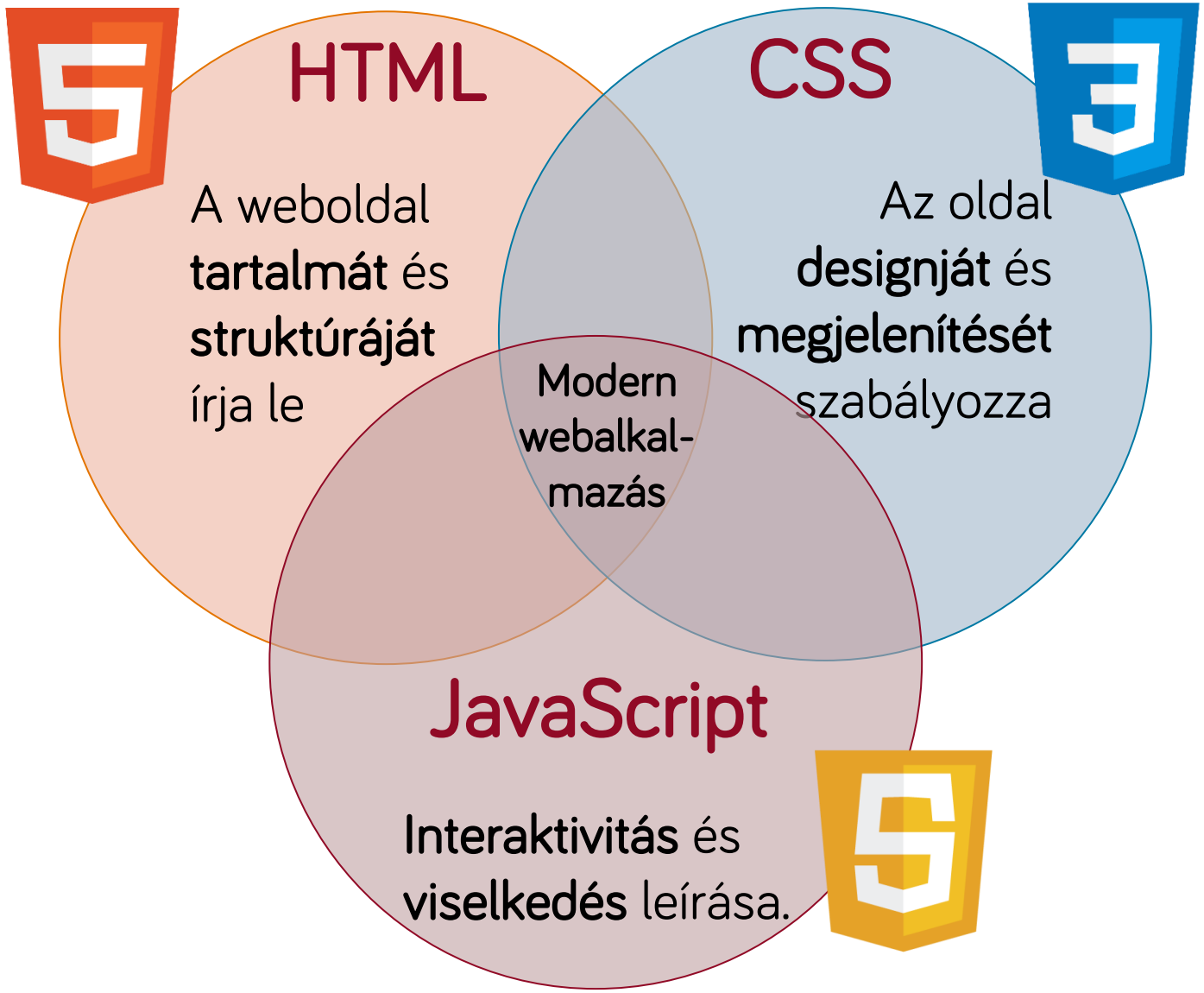
„ECMAScript was always an unwanted trade name that sounds like a skin disease.”

JavaScript egyéb változatai

- ActionScript
  - Macromedia (azóta Adobe) által a Flash platform programozására kialakított dialektus (kihalt).
- TypeScript
  - Microsoft által készített bővítmény
  - Típusossággal és valódi osztály-alapú objektum orientáltsággal egészíti ki a JavaScriptet.
  - Funkcionalitásában és szintaktikájában az ECMAScript-re épít.
  - A JS és TS fejlődése nagy hatással vagy egymásra mindkét irányban.

Transpiler

- Transpiler: source-to-source compiler
  - Babel: újabb verziójú JS fordítása régebbi verziókra (böngésző kompatibilitás)
  - CoffeeScript (önálló programozási nyelv)
  - TypeScript (típusos JavaScript)
- A transpiler olyan fordító, amely egy A nyelvű forráskódot lefordít B nyelvnek megfelelő forráskódra


HTML
A weboldal **tartalmát** és **struktúráját** írja le
CSS
Az oldal **designját** és **megjelenítését** szabályozza
Modern webalkal-mazás
JavaScript
Interaktivitás és **viselkedés** leírása.

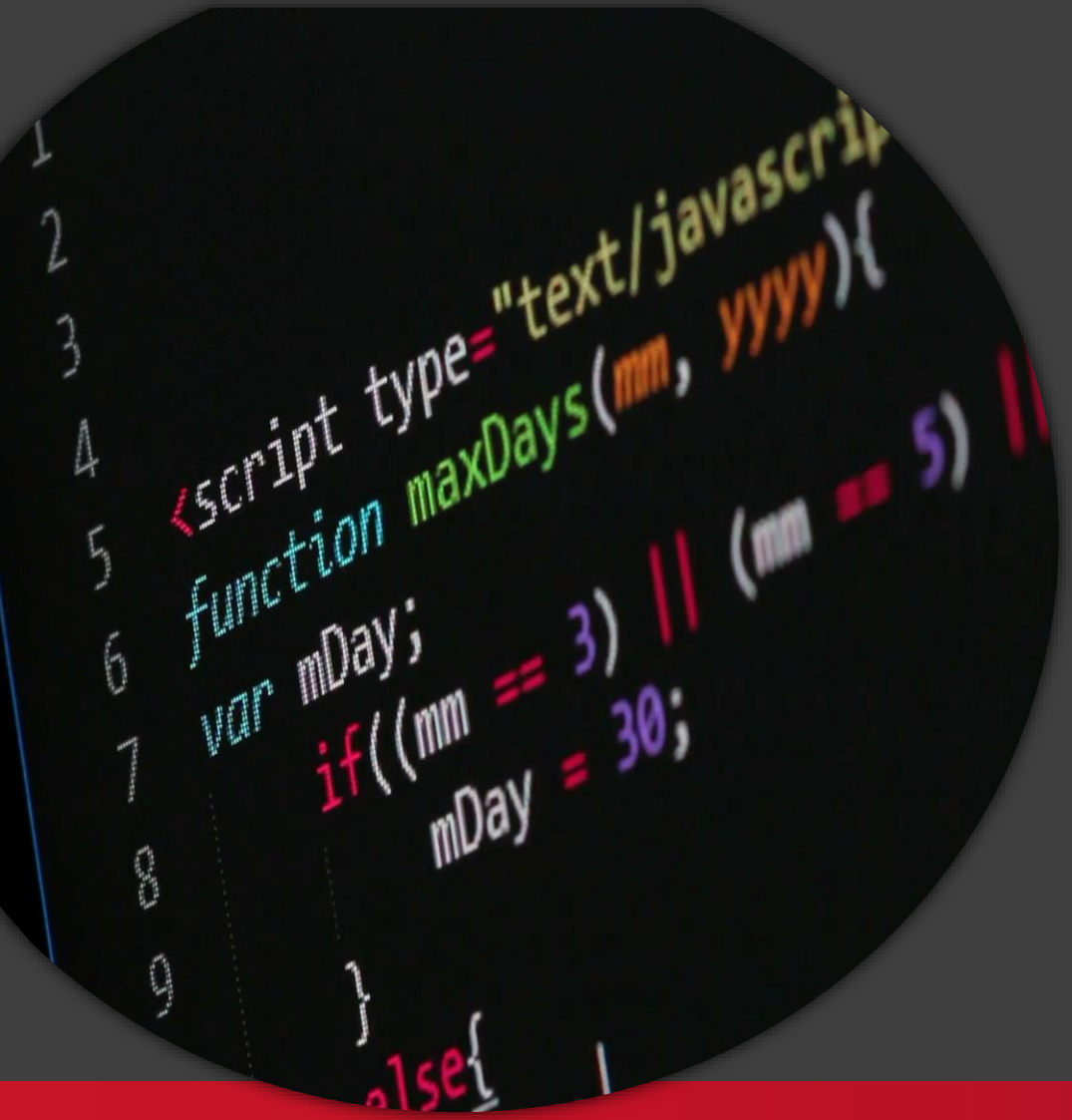
JavaScript, CSS és HTML összekötése

```
<!DOCTYPE html>
<html>
 <head>
   <link type="text/css" rel="stylesheet" href="mystyle.css"/>
   <script type="text/javascript" src="myscript.js"></script>
 </head>
 <body>
   ...
   <script type="text/javascript" src="myscript.js"></script>
 </body>
</html>
```

Java és a JavaScript mindenben eltér

Java	JavaScript
statikusan típusos	dinamikusan típusos
erősen típusos	gyengén típusos
bájtkódból töltődik be	forráskódból töltődik be
osztály-alapú objektumok	prototípus-alapú objektumok

- Az eredeti JavaScript specifikációban az összes Java kulcsszót lefoglalták.



A JavaScript nyelv alapjai

Kliensalkalmazások

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Gincsai Gábor
gincsai@aut.bme.hu

HTML és a JavaScript

- A `<script>` elemet mindig explicit záró címkével használjuk.
- A `<body>`-ba is tehető JS hivatkozás
  - Teljesítmény okokból az oldal végére a `</body>` elé tesszük.
- A `<script>` tag segítségével a HTML fájlba is kerülhet JS kód.
  - kód karbantarthatósága,
  - a funkciók szétválasztása ,
  - teljesítmény okok miatt ez **nem célszerű**.
- Az elemekhez a HTML-ben is rendelhetünk eseménykezelőket
  - pl: onclick="..."
  - kód átláthatósága miatt ez nem célszerű.

A nyelv elemei

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`

Változók láthatósága

- function scoping
  - A `var`-ral létrehozott változók az egész függvényen belül látható.
- block scoping
  - ES6-tól változót `let`-tel is létre lehet hozni.
  - A `let`-tel létrehozott változók csak a blokkon belül láthatók.

```
for( var i = 0; i < 2; i++ ){
      let loc = i;
}
console.log(loc); // ReferenceError: loc is not defined
```

Nyelv által definiált típusok

Egyszerű adattípusok

- number
  - `> typeof 0 // "number"`
- bigint
  - `> typeof 1n // "bigint"`
- string
  - `> typeof "foo" // "string"`
- boolean
  - `> typeof true // "boolean"`
- undefined
  - `> typeof undefined // "undefined"`
- null
  - `> typeof null // "object"`
- symbol – (ES6 óta)
  - `> typeof Symbol("id") // "symbol"`

Összetett adattípus

- Object
  - `> typeof Math // "object"`

És a függvény

- Function
  - `> typeof alert // "function"`

Egyszerű értékadás

```
var szoveg = 'Gipsz Jakab’;
szoveg2 = "Gipsz Jakab";
let szam = 10;
let logikai = true;
```

Ha a *var / let* kulcsszót elhagyjuk, akkor is létrejön a változó, azonban ebben az esetben a globális névtérben a window object-en jön létre.

strict mód

- A var kulcsszó véletlen elhagyásából eredő problémákat a **strict mód** bekapcsolásával lehet elkerülni, mert ilyenkor a var / let elhagyása hibát eredményez.

```
"use strict";
var ev = 2016;
honap = "január"; // Hiba
```

Dinamikus és gyenge típusosság

- A JavaScript nyelv **dinamikusan típusos**
  - `var/let` kulcsszóval hozzuk létre, azaz *nem adunk meg a típust.*
  - Egy változó amiben sztring érték van tartalmazhat később számot.
- A JavaScript nyelv **gyengén típusos**
  - Összeadás operátor működése változik a *változóban tárolt érték típusától*
    - Stringnél konkatenáció
    - Numbernél összeadás
    - És mi lesz ha stringet és numbert adunk össze?

Dinamikus és gyenge típusosság

```
let a = 3;
let b = 2;
alert(a + b);   // 5 (number)


a = 'Gipsz ';
b = 'Jakab';
alert(a + b);   // Gipsz Jakab (string)
```

Implicit típus konverzió

- A JavaScript gyengén típusos
  - Ha összeadunk két változót, előfordulhat, hogy az egyikben string a másikban pedig number szerepel.
  - Implicit típus konverzió van.

```
let ev = 2020;
let honap = 'január';
alert( ev + honap ); // '2020január'

alert( ev + honap + 1 ); // '2020január1'
alert( ev + 1 + honap ); // '2021január'
```

== és === közötti különbség

- Az == csak az értéket hasonlítja össze tehát a 1 mint szám és az "1" mint sztring egyenlő lesz.
- Az === egyenlőség viszont a típust infót is ellenőrzi tehát egy szám nem lehet egyelő egy sztringgel.

```
let ev = 2020;
if( ev == '2020' ) {    // True
        alert( "2020 == '2020' igaz" )
}
if( ev === '2020' ) {   // False
        alert( "2020 === '2020' igaz" )
} else {
        alert( "2020 === '2020' hamis" )
}
```

Ha nem adunk meg értéket undefined

- Ha egy változók értékadás nélkül hozunk létre, akkor
  - az értéke undefined lesz
  - a típusa is undefined lesz

```
let nap;
alert(nap); // undefined
if( nap === undefined ) { // true
      alert( "nap === undefined igaz.");
}
if( typeof nap === 'undefined' ) { // true
      alert( "typeof nap === 'undefined' igaz")
}
```

Mi az undefined és a null típusa?

- Az `undefined` típusa `undefined`
- A `null` esetében más a helyzet, mert a null egy `object`.
  - „The result of typeof null is "object". That’s an officially recognized error in typeof behavior, coming from the early days of JavaScript and kept for compatibility. Definitely, null is not an object. It is a special value with a separate type of its own.”

```
let n = null;
alert( typeof n );  // 'object’

let n = undefined;
alert( typeof n );  // 'undefined'
```


Forrás: https://javascript.info/types

Konstansok

- EcmaScript 6-tól van konstans is a JavaSciprtben.
- Konstans létrehozásakor az értéket is meg kell adni.
- Később nem módosítható az értéke.

```
const PI;       // SyntaxError: missing = in const declaration
const PI = 3.14;
alert( PI );    // 3.14
PI = 500;       // SyntaxError: invalid assignment to const PI
```

Logikai típusok

- JavaScriptben minden bool-lá alakítható!

truthy	falsey
true	false
'0'	0
123 vagy -123	NaN
'valami'	'', (üres string)
[]	null
{}	undefined

```
let b = !!'valami';
alert(typeof b); // boolean
alert(b);        // true


alert(!!'0');    // true
alert(!!0);      // false
```

Primitív típus burkolása objektumba

- Egy bool változót többféleképpen is létrehozhatunk

```
let b_obj = new Boolean(false); // objektum!!
let b = false;                  // primitív típus
let b_cast = Boolean(false);    // bool-lá kasztolás
let b_cast2 = !!false;          // bool-lá kasztolás
```

- A **new** Boolean(false) egy objektumot hoz létre! Emiatt a tagadása (bool-lá alakítása) esetén azt nézi, hogy ez egy nem üres objektum.
  - **!b_obj** értéke **false** lesz! → Nem üres objektum tagadása.
  - **!b_cast** pedig **true**! → Ez primitív típus az értékét negálja.

Primitív típus burkolása objektumba

- Az egyszerű típusokat objektumba lehet burkolni
  - `new Number(12)`
  - `new Boolean(false)`
- Ilyen esetben ha bool-lá alakítjuk dupla tagadással
  - Igaz vagy hamis értéket kapunk? Miért?

```
let n = new Number(0);
alert(!!n);  // true

let b = new Boolean(false);
alet(!!b);  // true ?!
```

Változók létezésének vizsgálata

- Mivel minden változó az értékétől függetlenül bool-lá alakítható és mint tudjuk az **undefined falsy**, ezért azt, hogy egy változónak van-e értéke az alábbi kóddal tudjuk vizsgálni.

```
let valtozo;
if( valtozo ) {      // bool-lá alakítja az if miatt
      alert("A valtozó létezik");
} else {
      alert("A változó nem létezik")
}
```

- Mi történik akkor ha a változó értéke false (Boolean)?

Alapértelmezett érték megadása

- Mivel az **undefined falsy**, ezért ha egy változónak csak akkor szeretnénk értéket adni, ha még nincs neki, azt az alábbi kódrészlettel megtehetjük.
- A megoldás lényege, hogy az `afa` false lesz ezért a vagy után megadott rész is kiértékelődik és azt az értéket kapja meg a változó.

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

Feltételes kódfuttatás

- Az alábbi kódrészlettel megoldható, hogy az `alert` csak akkor hívódjon meg, ha az `x` változó értéke **truthy**.

```
let x = ''; // próbáljuk ki 'a' -val is.
x && alert('lefutott');
```

Objektumok

- Kulcs-érték párok
  - kulcs: property neve (~ string)
  - Érték: property értéke (tetszőleges adattípus)
- Objetum adatainak elérése
  - objektum.property
  - objektum['property neve'].
- Két érdekes következménye van annak, hogy egy objektum valójában kulcs-érték párok tárolója:
  - Ha le akarunk kérni egy property-t, ami még nem létezik, akkor nem kapunk hibát, hanem egyszerűen undefined lesz a visszatérési érték.
  - Ha írni szeretnénk egy property-t, ami még nincs benne az objektumba, akkor nem kapunk hibát, hanem az egyszerűen belekerül.

Objektumok

```
let objA = new Object();
let objB = {};

objB.alma = "alma";
console.log(objB.alma);    // "alma"
console.log(objB["alma"]); // "alma"
console.log(objB["körte"]); //undefined

objB.name = "dió";
console.log(objB.name); // "dió"
```

```
let obj = {
    name: "alma",
    value: "körte",
    "dió": 5
};

for (let prop in a) {
    // "name", "value", "dió"
}
if ("name" in a) {
    //igaz
}
```

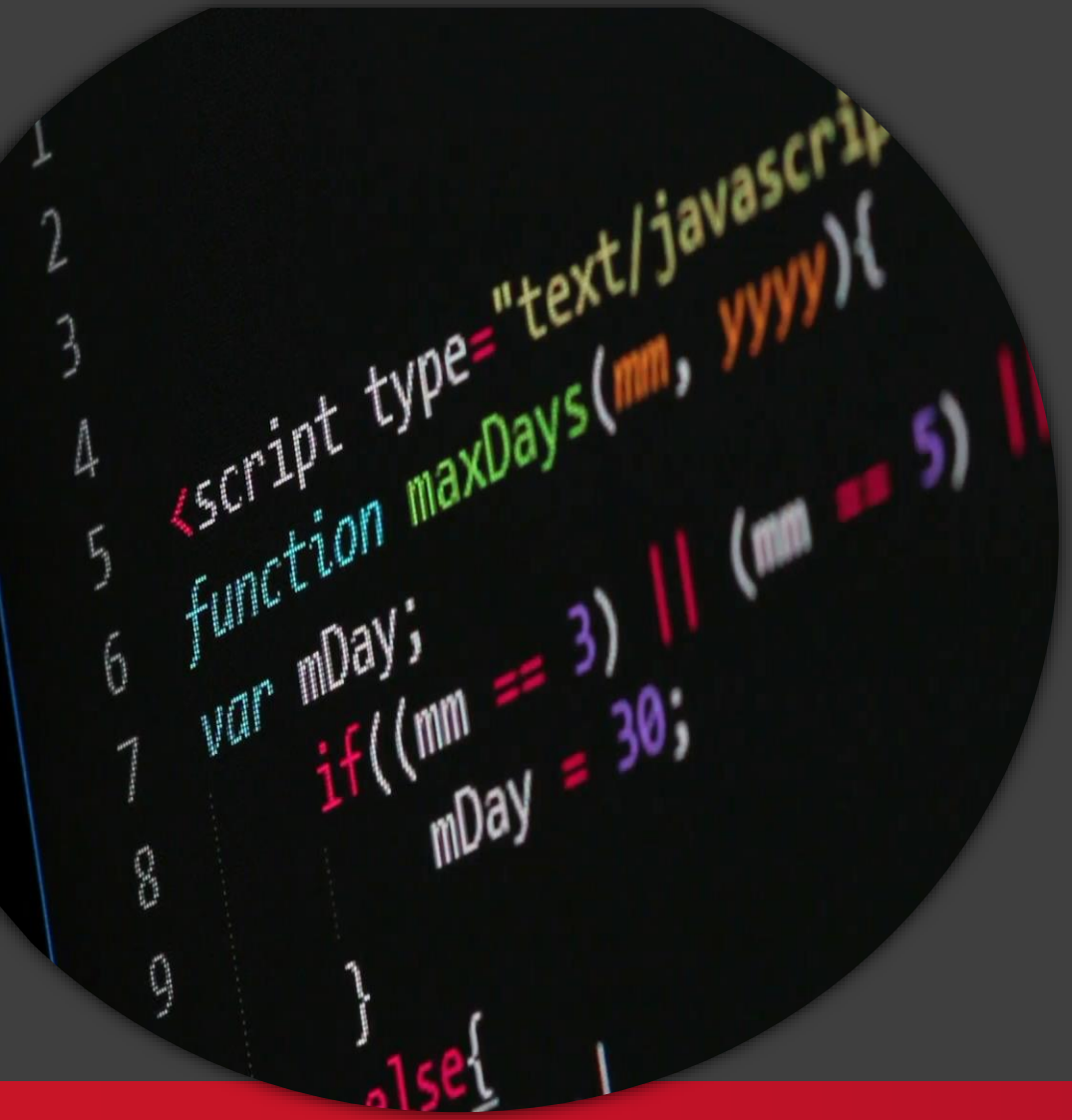
JavaScript Object Notation (JSON)

- Adatleíró szöveges formátum
- JS objektumokhoz inicializálásához hasonlít
  - A property-k neveit idézőjelekbe tesszük
  - Értelemszerűen nincsenek metódusok, referenciák benne (nem programozási nyelv)
  - Egy gyökér objektum (lehet lista is)
  - Nincs komment
- Gyakran használjuk AJAX hívásoknál XML helyett
- JS beépítve támogatja az objektumok sorosítását
  - JSON.stringify()
  - JSON.parse()

JSON

```
{
  "name": "Felhasználó",
  "id": 1121,
  "address": {
    "street": "József Attila u.",
    "city": "Budapest"
    "nbr": "5"
  },
  "phones": [
    "06-30-12345678",
    "06-30-98765432"
  ]
}
```

```
// list <-- [1,2,3]
let list = JSON.parse("[1,2,3]");
console.log(typeof list);  // object

// list_s <-- "[5,6,7]"
let list_s = JSON.stringify([5,6,7]);


console.log(typeof list_s); // string
```



Tömbök és függvények

Kliensalkalmazások

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Gincsai Gábor
gincsai@aut.bme.hu

Tömbök kezelése JavaScriptben

- Tömböket az alábbi két módon lehet létrehozni.
  - A két megoldás ekvivalens egymással.

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

alert( typeof napok );        // 'object'
alert( typeof evszakok );     // 'object'

for( let i = 0; i < napok.length; i++ ) {
      alert( napok[ i ] );    // hétfő, kedd, szerda
}
```

Elem hozzáadása és elvétele

- A tömbhöz új elemet a push()-al tudnk adni
- Tömbből az utolsó elemet a pop()-al tudjuk kivenni.

```
napok.push( 'csütörtök' );
napok.push( 'péntek' );
let utolso = napok.pop();
alert(utolso); // péntek

for( let i in napok ) {
      alert( napok[ i ] );
      // hétfő,kedd,szerda,csütörtök, de péntek nem!
}
```

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

- **Index**: a tömb módosítása melyik elemnél kezdődik
  - ha negatív a tömb végéről számol
- **howMany**: az eredeti tömbből hány elemet szeretnénk törölni
- **el1...elN**: opcionális, a tömbbe beszúrandó új elemek.

```
let szinek = [ 'piros', 'sárga', 'kék' ];
let toroltek = szinek.splice( 1, 2, 'fehér', 'zöld' );
alert("Törölt színek:");
for( let t in toroltek ) {
   alert( toroltek[ t ] );    // Törölt elemek: sárga,kék
}
alert("Ami maradt:");
for( let sz in szinek ) {
   alert( szinek[ sz ] );     // Tömb elemei: piros,fehér,zöld
}
```

Függvények

- Nem lehet megadni a
  - paramétereknek típusát
  - visszatérési érték típusát
- Ha a paraméterek száma a híváskor és a deklarációkor eltért nem okoz gondot.
  - Minden paraméter opcionális, ha valamit nem adunk meg undefined lesz az értéke.
  - Ha több paraméterrel hívjuk meg figyelmen kívül hagyja.
  - Függvény híváskor a paraméterekre névvel nem lehet hivatkozni, ezért csak a paraméter lista végéről tudunk elhagyni elemeket.
- Nincs overload
  - Ha van két azonos nevű függvény, a később deklarált nyer.

Függvények

```
function kiir(szoveg) {
      alert(szoveg);
}
kiir('Ébresztő!');

function osszead( a, b ) {
      return a + b;
}
let osszeg = osszead( 2, 3 );
kiir(osszeg);      // => 5
```

Paraméter alapértelmezett értékkel

- ES6-ban már lehetőség van a függvény létrehozásakor megadni a paramétereknek alapértelmezett értéket. Így ha az alábbi példában elhagyjuk a b paramétert, akkor az nem undefined lesz, hanem 4.

```
function osszeadAlap(a, b = 4) {
      return a + b;
}
alert(osszeadAlap( 2 ) )     // => 6
```

A függvény is egy teljes értékű típus

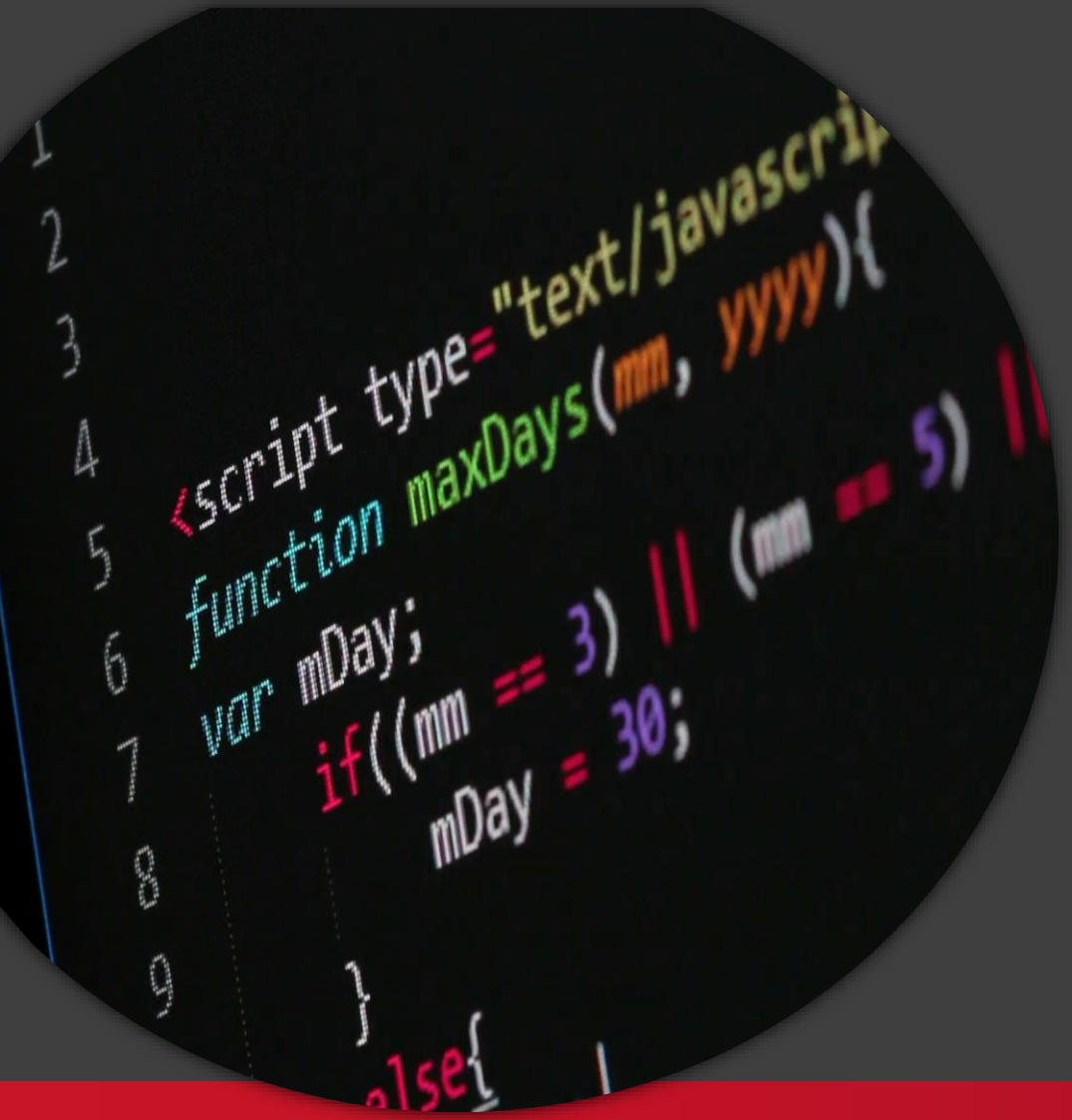
- Meglepőnek tűnhet, de JavaScripben a function egy teljes értékű típus.

```
function kiir(szoveg) {
    alert(szoveg);
}
alert( typeof( kiir ) );    // => 'function'
```

Függvényt paraméterül is átadhatunk

- Mivel teljes értékű típus a function ezért egy függvénynek lehet függvény a bemenő paramétere
  - A paraméterként megkapott függvényt meg is tudjuk hívni.
  - Figyelni kell, hogy tényleg függvény-e a paraméter.

```
function mind( tomb, fv ) {
      for( let i in tomb ) {
            fv( tomb[ i ] );
      }
}
mind( napok, kiir );
```



Változók láthatósága

Kliensalkalmazások

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Gincsai Gábor
gincsai@aut.bme.hu

window objektum

- Böngészőben futtatott JavaScript kód esetén a `window` objektum segítségével hivatkozhatunk az oldalhoz tartozó ablakra vagy keretre (frame).
- Mivel felső szintű objektumról van szó, nagyon sok általános tulajdonság, metódus és esemény rajta keresztül érhető el, például (nem teljes lista):
  - document, history, location, navigator, screen, status, applicationCache, console, …
  - alert(), confirm(), focus(), open(), close(), print(), scrollTo(), setInterval(), setTimeout(), …
  - onload, onbeforeunload, ondragdrop, onerror, onresize, …

Hol jön létre a változó

- A **var**-al létrehozott változók amik nincsennek függvénybe zárva a window objecten jönnek létre.

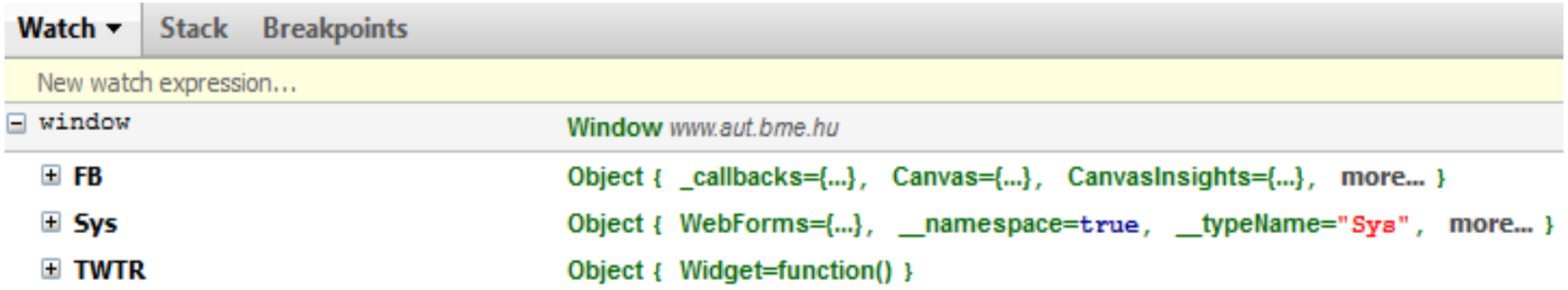
```
var a = 'Szia Világ';
alert(window.a);
```

  - A függvények is a window objectre kerülnek

```
function say(name) {
    alert('Szia ' + name)
}
window.say('Világ');
```

Globális névtér szennyezés

- Törekednünk kell arra, hogy elkerüljük a globális névtér beszennyezését (global scope pollution).
- Mindig be kell zárnunk a változóinkat egy objektumba.



Lokális változók

- Más programozási nyelvekhez hasonlóan a függvényeken belül létrehozott változók JavaScriptben is lokális (helyi) változók, azaz csak az adott függvényen belül érhetők el.

```
function f() {
    var x = 8;
    alert('Belül: ' + x);  // Belül: 8
}
f();
alert('Kívül: ' + x);  // ReferenceError: x is not defined
```

Változók láthatósága: function vs block scoping

- A **var** kulcsszóval létrehozott változók function scoping-ot használnak, azaz a teljes függvényben elérhetőek, nem veszi figyelembe a belső blokkokat.
- A **let** kulcsszóval létrehozott változók block scopingot használnak, azaz csak az adott blokkban { } érhetők el.

```
for( var i = 0; i < 2; i++ ){
    let loc = i;
}
console.log(loc); // ReferenceError: loc is not defined
```

Változók láthatósága

- Nincs közvetlen lehetőség a privát és publikus tagok megjelölésére.
- A függvények adta lehetőségeket kell használnunk.
  - JavaScriptben a `var` kulcsszóval létrehozott változóknak csak és kizárólag a függvény blokkok határozzák meg a láthatóságát, más néven hatókört (functional scoping).
  - A `let` kulcsszóval létrehozott változók viszont blokk szinten lesznek csak láthatók.

Lokális vs külső változók

- A változók megtalálása a legmélyebb szintről felfelé történik.
- Ha egy változó helyben nem található, akkor a futtató motor megnézi, hogy a külső függvényben megtalálható-e, és ha ott sem, akkor így halad felfelé egészen a globális névtérig (window).
- Tehát a lokális változók elfedik a külső változókat
  - Shadowing

Egy példa a shadowingra

```
let x = 5;
function f() {
    let x = 8;
    alert('Belül: ' + x);
}

f();
alert('Kívül: ' + x);
```

DOM manipuláció

Kliensalkalmazások

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Gincsai Gábor
gincsai@aut.bme.hu

JavaScript API-k

- Browser API-k:
  - DOM API: HTML és CSS manipulálására.
    - HTML tagek dinamikus létrehozása, törlése, módosítása
    - HTML elemek stílusának módosítása futási időben
  - Szerver kommunikációt lehetővé tevő API
    - XmlHttpRequest amin keresztül kéréseket indíthatunk a szerver felé. Tipikusan REST API-n keresztül.
  - Client Storage API
    - Kliens oldali adattárolást tesz lehetővé. Pl.: web storage, IndexedDB
- Külső API-k:
  - Google Maps API
  - Twitter API

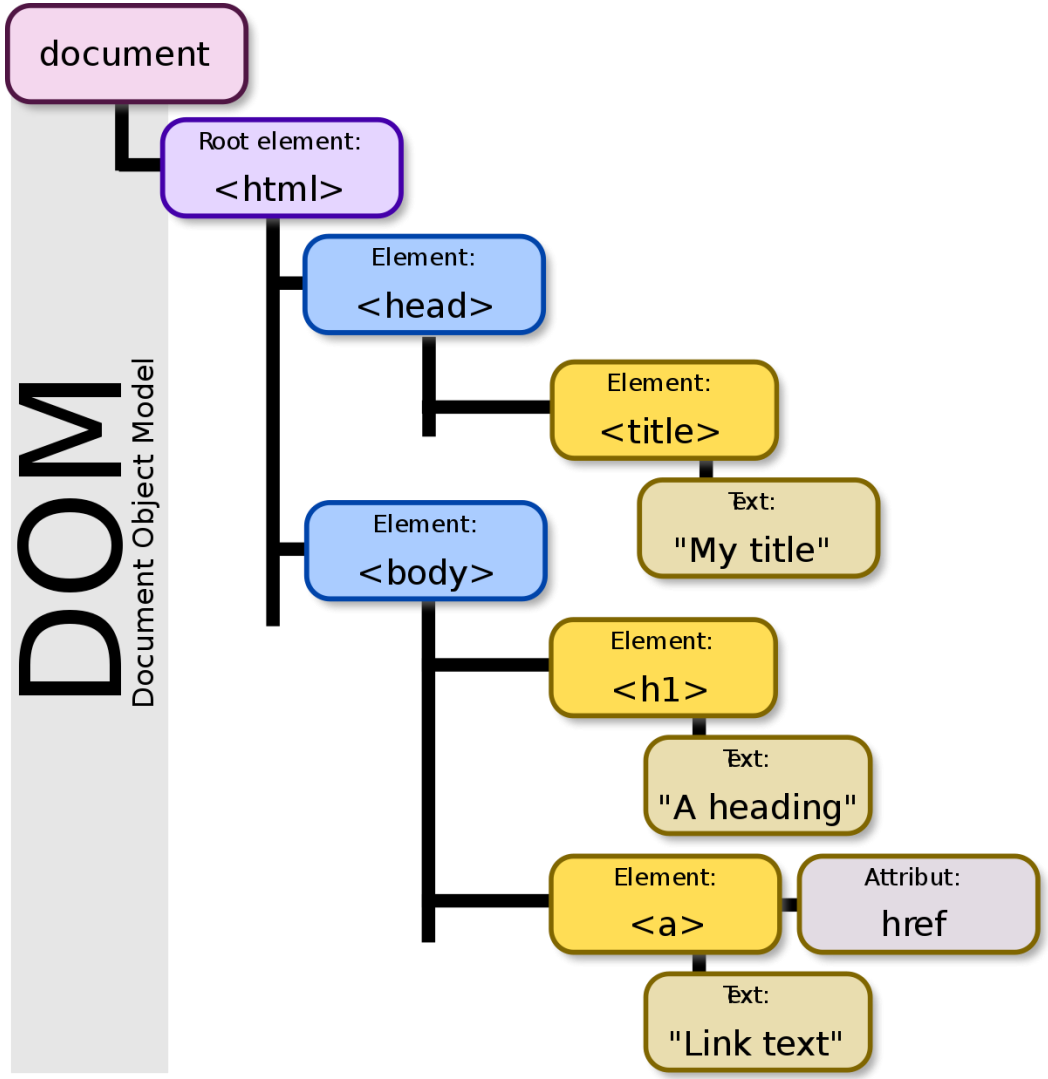
Legfontosabb böngészőbeli objektumok

A legfontosabb objektumok

- window objektum: A böngésző tabfüle, amibe a weboldal betöltődik
  - Itt érhetjük el a console-t.
  - Itt jönnek létre a globális objektumok.
  - Az oldal URL-jét (location) is itt találjuk és módosíthatjuk.
  - Vagy a history-t, ami az előre / hátra lapozáshoz kell.
- Navigator objektum: A böngésző állapotát tárolja
  - Lekérdezhető a felhasználó által preferált nyelv, geolokáció…
- Document objektum: Maga a DOM, ami a window objektumba betöltődik.
  - Ezen keresztül tudjuk a HTML tageket módosítani.

DOM – Document Object Model

- 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 DOM-ból

- 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 lekérdezésére selectorral
  - **`document.querySelector('a');`**
  - **`document.querySelectorAll('a');`**

Elemek létrehozása dinamikusan

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

```
var para = document.createElement('p');
```

- Állítsuk be a szükséges attribútumokat, vagy a tartalmát

```
para.textContent = 'Új paragrafus';
```

- Keressünk ki egy a HTML-ben már létező elemet és ahhoz fűzzük hozzá az újat, hogy megjelenjen.

```
var sect = document.querySelector('section');
sect.appendChild(para);
```

Egy példa elem létrehozására

```
// Első section lekérdezése
var sect = document.querySelector('section');

// <p> létrehozása
var para = document.createElement('p');
para.textContent = 'Új paragrafus';

// <p> hozzáfűzése a section végéhez
sect.appendChild(para);

// Sima szöveg létrehozása tag nélkül
var text = document.createTextNode(' — további szöveg ');
// Az első <p> tag megkeresése
var linkPara = document.querySelector('p');
// <p>-hez hozzáfűzzük a szöveget.
linkPara.appendChild(text);
```

Oldal elemeinek módosítása

- A lekérdezett adatokat módosítani is lehet.

```
var para = document.querySelector('p');
para.style.color = 'white';
para.style.backgroundColor = 'black';
para.style.padding = '10px';
para.style.width = '250px';
para.style.textAlign = 'center’;
```

```
<p style="color: white; background-color: black; padding: 10px;
          width: 250px; text-align: center;">
   Új paragrafus
</p>
```

Tulajdonságok CSS-ben és JS-ben

- Ügyeljünk arra, hogy a tulajdonságokat másképpen írjuk CSS-ben és JS-ben.
  - míg CSS-ben csupa kisbetűvel és kötőjellen
  - addig JS-ben camelCase-el írjuk
- `CSS: background-color`
- `JS: backgroundColor`

Oldal elemeinek módosítása II.

- Arra is van lehetőség, hogy a HTML tagekre egy-egy új attribútumot aggassunk futási időben.
- Az alábbi kódrészlet a class attribútumot állítja be highlight-ra.

```
para.setAttribute('class', 'highlight');
```

DOM bejárása

```
function walkTheDOM(node, func) {
  // Aktuális elemen meghívjuk a megkapott függvényt.
  func(node);
  // Vesszük az első gyerek elemét
  node = node.firstChild;
  while (node) {
    // Rekurzívan tovább hívjuk. (mélységi bejárás.)
    walkTheDOM(node, func);
    // Ha már nincs gyerek, akkor nézzük a következő testvért.
    node = node.nextSibling;
  }
}
```

Események kezelése

Kliensalkalmazások

BME AUT
Automatizálási és Alkalmazott Informatikai Tanszék

Gincsai Gábor
gincsai@aut.bme.hu

Felhasználói interakció kezelése

- Ahhoz, hogy a JavaScript segítségével dinamikus weboldalakat tudjuk létrehozni a felhasználói interakciókat kliens oldali kezelni kell
- Ehhez számos eseményre tudunk feliratkozni
  - Elemre kattintás esemény: **click**
  - Billentyű lenyomása: **keypress**
  - Elem fókuszba kerülése: **focus**
  - Egeret egy elem fölé visszük: **mouseover**
  - Beviteli mező tartalma megváltozott: **change**
  - **...**

Inline eseménykezelő

- A HTML-ből közvetlenül inline fel tudunk iratkozni az adott elem egy-egy eseményére.

```
<button onclick="alert('click')">Gomb</button>
```

- Ez a megoldás viszonylag egyszerű, hiszen amint létrejön a gomb azonnal fel is iratkozunk a kattintás eseményre
- Átláthatóság szempontjából viszont nem jó, mert a HTML-ből állítjuk a gomb viselkedését.

Oldal betöltődése esemény

- Eseménykezelőt csak akkor tudunk egy elemhez regisztrálni, ha az az elem már létezik.
- Szükséges egy olyan globális esemény, ami azt jelzi, hogy a HTML elemek már betöltődtek
  - onLoad esemény pont erre való
  - A window-n találjuk meg, hiszen az adott oldalhoz tartozik.

```
window.onload = function(){
    alert('Az oldal betöltődött')
}
```

Eseménykezelő regisztrálása JS-ből

- Az onLoad lefutása után szabad csak beregisztrálni eseménykezelőt, mert előtt nem biztos, hogy létezik a HTML elem.
- Keressük meg az a HTML elemet.

```
var btn = document.querySelector('#myBtn');
```

- Adjuk meg, hogy melyik eseménykezelőre, melyik függvényt szeretnénk regisztrálni.
  - Az esemény neve elé kell tenni az **on** prefixet

```
btn.onclick = function() { … }
```

```
btn.onclick = kiir;
```

Több eseménykezelő regisztrálása

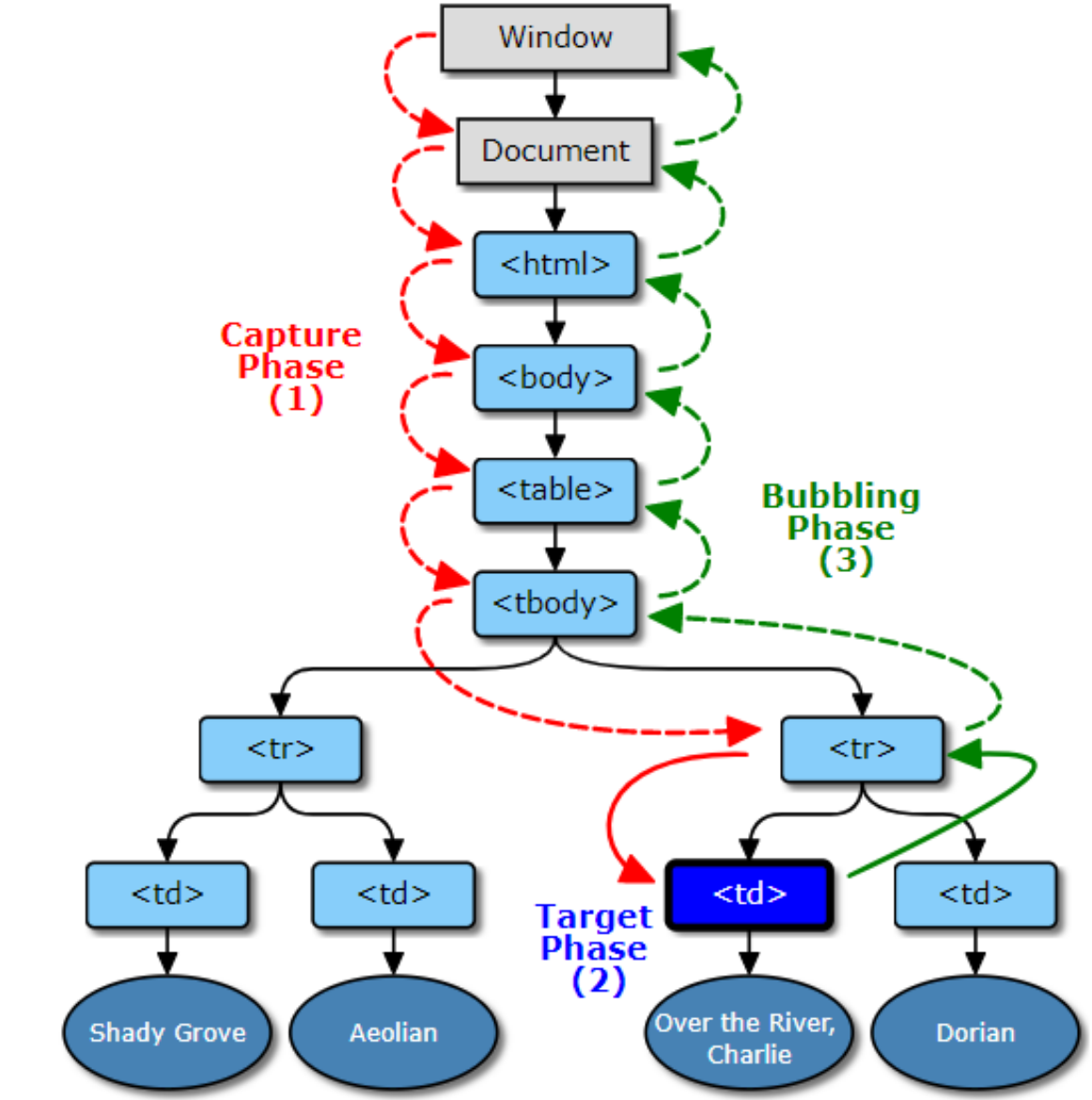
- Az előző példával egy elem egy eseményéhez csak egy eseménykezelőt tudunk megadni.
  - Csak 1db onclick lehet, ha újat adunk meg felülírjuk a korábbi.
- Ha több eseménykezelőt szeretnénk regisztrálni, akkor a megoldás az `addEventListener()`.

```
var btn = document.querySelector('#myBtn');

btn.addEventListener("click", modifyText);
```

- Az eseménykezelők a regisztráció sorrendjében egymás után futnak le.

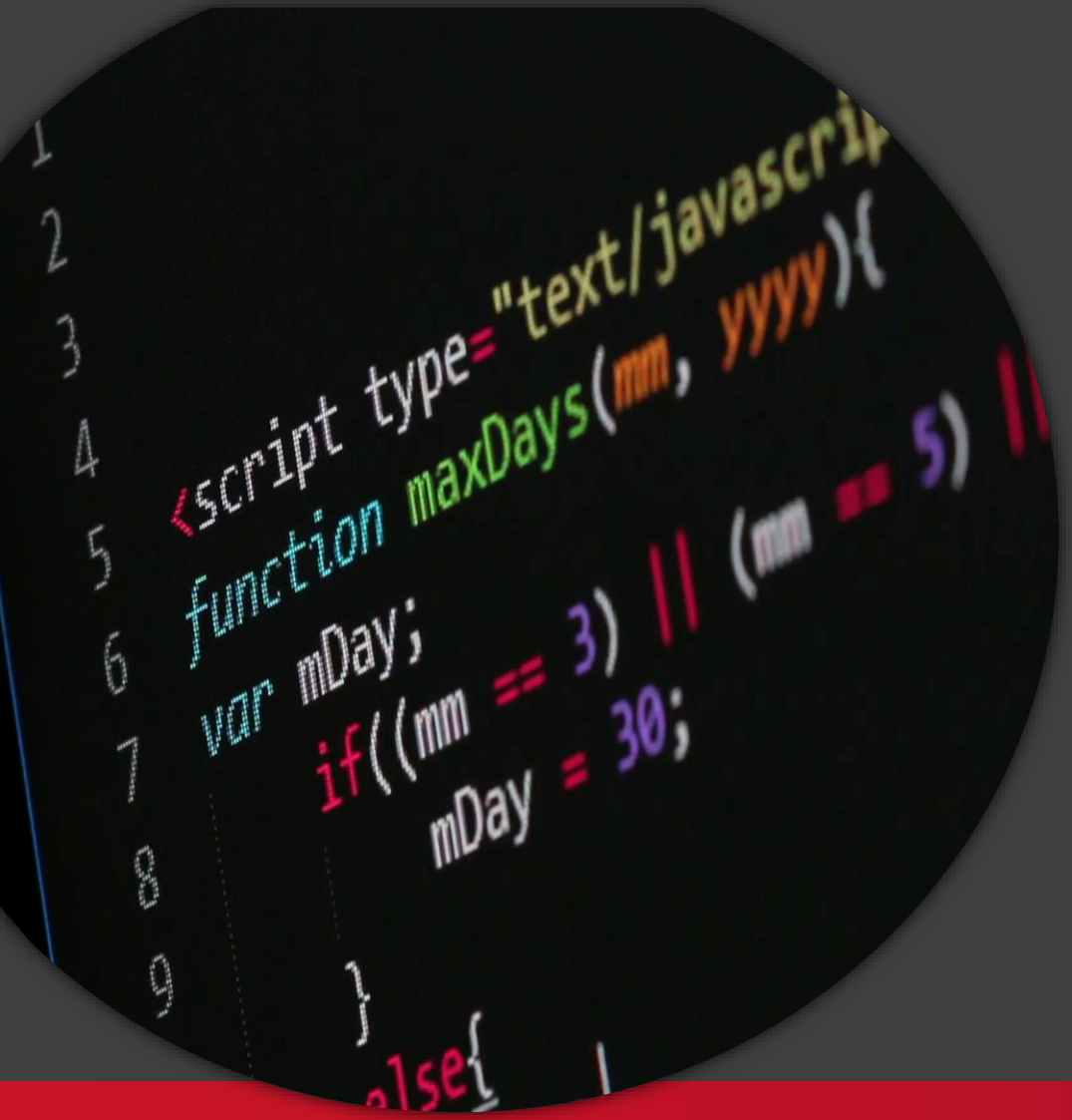
Mit jelent a bubble?

- Egy a fában több elemnél is feliratkozunk például a kattintás eseményre, akkor
  - Először fentről lefelé megkeresi a böngésző, hogy melyik elemre kattintottak (ahova beregisztráltuk az eseménykezelőt)
  - Meghívja az ott beregisztrált eseménykezelőt
  - Majd ha lefutott, akkor a gyökér elemig felgyűrűzik (bubble up) az esemény.
  - Ezt később is tudjuk kezelni.
- A `stopPropagation()`-el leállíthatjuk a felgyűrűzést.



Graphical representation of an event dispatched in a DOM tree using the DOM event flow

Forrás: https://www.w3.org/TR/DOM-Level-3-Events/#event-flow



Állapotkezelési megoldások

Kliensalkalmazások

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Gincsai Gábor
gincsai@aut.bme.hu

Web Storage (DOM Storage)

- Probléma a cookie-kal:
  - Korlátozott méret.
  - Minden HTTP kérésben jelen van
    - biztonság, teljesítmény gondok.
  - Több böngésző ablakban egy időben nehezen használható (HTTP kérésekhez kötődik).
- Megoldás: **Web Storage** (más néven DOM Storage)
- Eredetileg a HTML 5 specifikáció része, már önálló
  - http://www.w3.org/TR/webstorage/

Web Storage (DOM Storage)

- Kulcs-érték párok tárolása.
- A kulcs és az érték is csak string lehet.
- Más típusú objektumokat a böngészők automatikusan stringre konvertálnak.
- Komplex struktúrákat célszerű JSON formátumra konvertálni.
  - **JSON.parse()**, **JSON.stringify()**

Web Storage – Méret korlát

- A böngészőknek kvótát kell alkalmazniuk
  - kérhet a felhasználótól jóváhagyást több adat tárolására, amikor a kvótát eléri.
- Javasolt limit 5MB/origin.
  - JavaScript string UTF-16 → 1 karakter = 2 byte → egyes böngészőknél (főleg a Chromium alapúaknál) csak 2 500 000 karakter tárolható.
  - Teszt oldal: http://dev-test.nemikor.com/web-storage/support-test/

Web Storage - Típusai

- **Session storage:** az információk csak a böngésző (tab) bezárásáig maradnak meg.
  - Csak az adott böngésző ablak érheti el a tárolt adatokat.
  - Ha a HTML oldalt lokális fájlként nyitjuk meg a böngészőben, akkor a session storage nem érhető el.
- **Local storage:** megőrzi az adatokat, amíg a felhasználó le nem törli.
  - Az adott domainhez tartozó összes oldal elérheti a tárolt adatokat.
  - A felhasználó bármikor törölheti → fel kell készülni rá.

Local Storage használata

- Elem hozzáadása

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

- Elem kiolvasása

```
> const cat = localStorage.getItem('myCat');
```

- Elem törlése

```
> localStorage.removeItem('myCat');
```

- Összes elem törlése

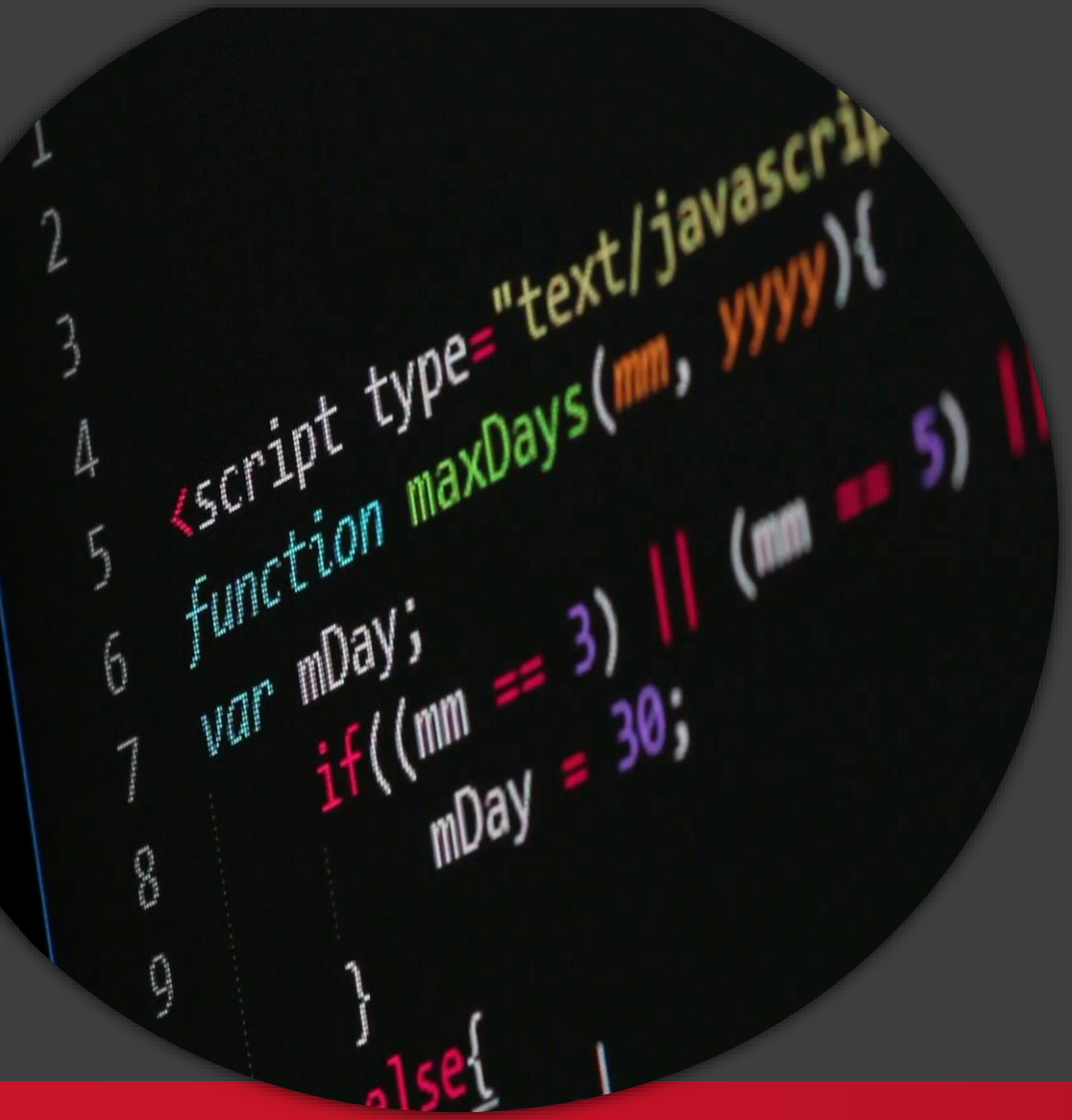
```
> localStorage.clear();
```

Sesison Storage példa

```
// Input mező keresése a HTML-ben ID alapján
let field = document.getElementById("field");

// Ha a mező korábbi állapotát már elmentettük.
if (sessionStorage.getItem("autosave")) {
        // Mező értékének visszaállítása
        field.value = sessionStorage.getItem("autosave");
}

// Feliratkozás a mező értékének változására
field.addEventListener("change", function() {
        // Érték mentése a session storage-ba.
        sessionStorage.setItem("autosave", field.value);
});
```

Cookie vs Storage

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



A függvény teljes értékű típus

Kliensalkalmazások

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Gincsai Gábor
gincsai@aut.bme.hu

Függvények teljes értékű típusok

- A JavaScriptben a függvények teljes értékű típusként viselkednek, ami további izgalmas lehetőségeket biztosít.

```
var kulso = function () {
    var x = 8;
    var belso = function () {
        alert(++x);
    };
    return belso;
};
var b = kulso(); // A belso függvényt adja vissza
b(); // Kiírja, hogy 9
```

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.
- 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.

Névtelen függvények

- A korábbi példában felesleges nevet adni a belső függvénynek.

```
var kulso = function () {
    var x = 8;
    return function () {
        alert(++x);
    }
};
var b = kulso();
b(); // 9
b(); // 10
```

Self-executing (self-invoking) functions

- Érdekes lehetőség a JavaScript nyelvben a függvények automatikus futtatása.

```
var fv = function (nev) {
    alert('Szia ' + nev);
};
fv('Világ');
```

- Csak azért adtunk neki nevet, hogy meg tudjuk hívni.

```
(function (nev) {
    alert('Szia ' + nev);
})('világ');
```

Modul tervezési minta

- A modul tervezési minta segítségével a kódunkat egy minden mástól független névtérbe csomagolhatjuk, elkerülve a globális névtér szennyezését.

Modul tervezési minta

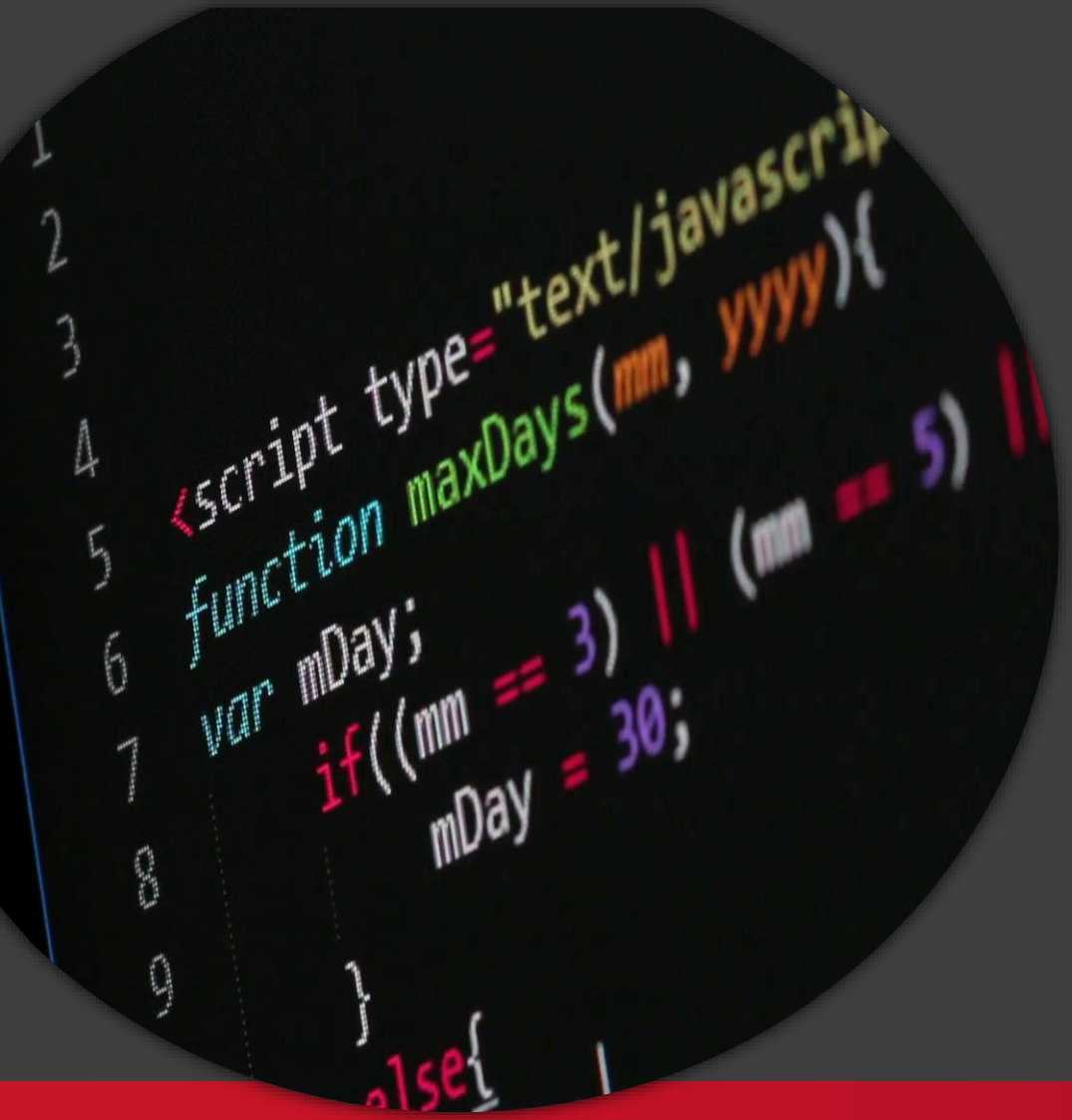
```
var N = (function () {
    var priValt = 3;
    var priFv = function () { alert('Privát!'); };
    return {
        pubValt: 5,
        pubFv: function () { alert('Publikus'); }
    };
})();

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

Modul tervezési minta

- A konstruktor függvény segítségével a modulnak adhatunk bemeneti paramétereket (import), a return kulcsszó után pedig azt határozhatjuk meg, hogy kívülről mi látszik a modulból (export).
- Az import-export lehetőségnek köszönhetően a modulunk kódját akár több fájlba is tehetjük, a publikus tagok el fogják érni egymást:

Akár több fájlra is darabolhatjuk

```
var N = (function (my) {
    my.pubFv = function () { alert('Publikus'); };
    return my;
}(N || {}));
// Másik fájl
var N = (function (my) {
    my.pubValt = 5;
    return my;
}(N || {}));
// Felhasználás
N.pubFv();
alert(N.pubValt);
```



A new és a this

Kliensalkalmazások

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Gincsai Gábor
gincsai@aut.bme.hu

A new operator

- Az automatikus futtatás elhagyásával és a new operátor használatával osztály-szerű viselkedést is el tudunk érni.

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

A this kulcsszó

- JavaScriptben is létezik this kulcsszó, és a legtöbb esetben arra az objektumra mutat, amin a függvényt meghívjuk.
- Az alábbi esetben például a függvény a window objektumhoz tartozik, ezért a this a window-ra mutat

```
var F = function () {
    this.A = 1;
};
F();
alert(window.A);
```

A this kulcsszó

- Ha az előző kódot kicsit módosítjuk és használjuk a new operátort, akkor megváltozik a viselkedés

```
var f = new F();
alert(window.A);    // undefined
alert(typeof f);    // object
alert(f.A);         // 1
```

- A new operátor a megadott konstruktor függvény segítségével **létrehoz egy új Object-et, és beállítja a this-t erre az objektumra**.
  - A new operátor ezzel az objektummal tér vissza (ha a konstruktor függvénynek nincs visszatérési értéke).

A this-el egyszerűsíthetjük a kódunkat

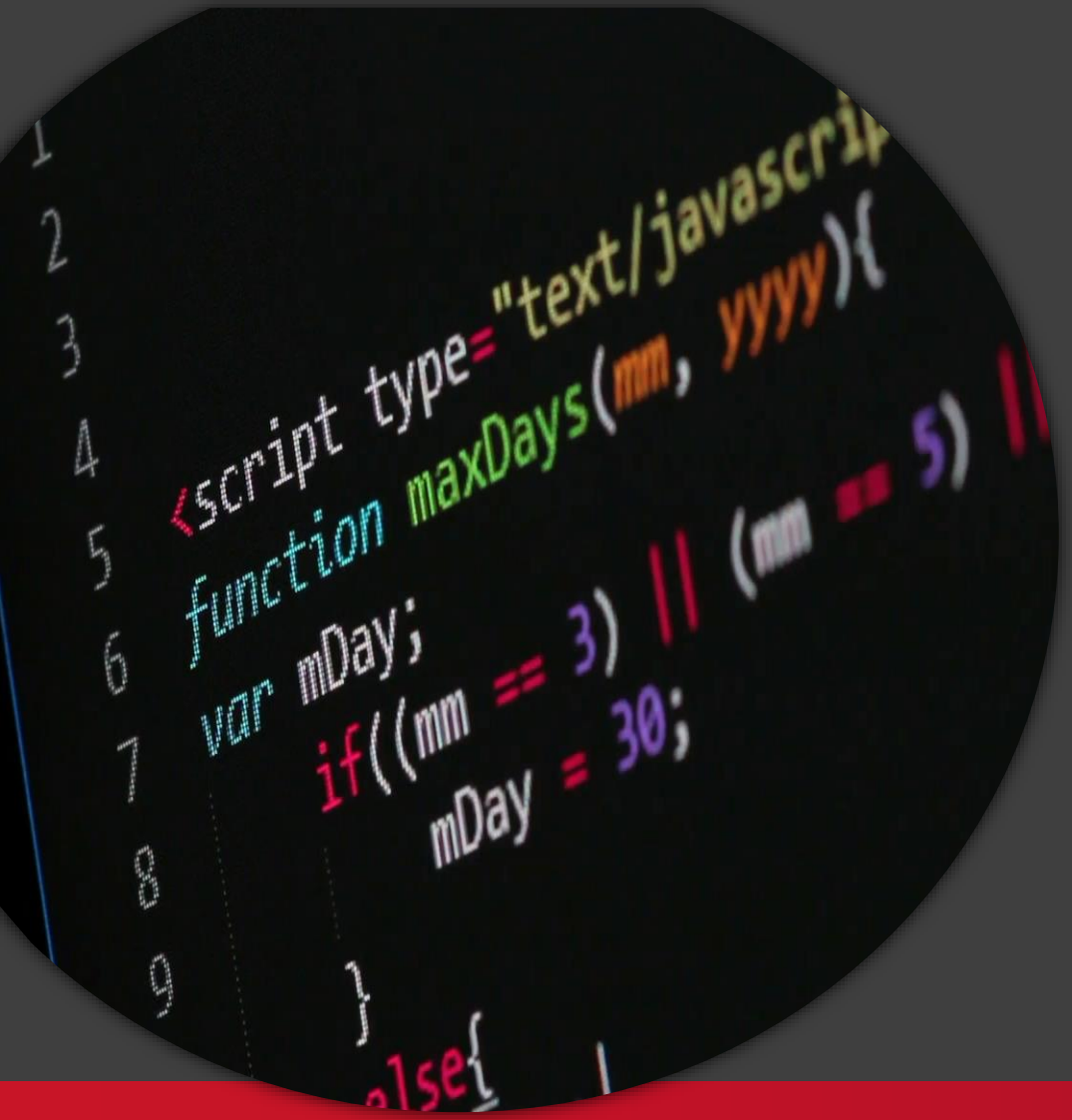
```
var MyClass = function () {
    var priValt = 3;
    var priFv = function () { alert('Privát!'); };

    this.pubValt = 5;
    this.pubFv = function () { alert('Publikus'); }
};

var c = new MyClass();
c.pubFv();
alert(c.pubValt);
```

A this nem mindig az a this

- Előfordul, hogy a this kulcsszó nem arra az objektumra mutat, amin belül használjuk.
  - Pl: egy gomb eseménykezelőjében a this gyakran a gombhoz tartozó DOM elemre mutat.
  - Ilyen esetekben zavaró, hogy az eseménykezelő függvényben a `this` mást jelent, mint az objektum más függvényeiben.
- A hibás működés elkerülése érdekében ezért a this használatát kerülni szoktuk, és az osztály tagjainak elérését más nevű változón (pl. that, self) keresztül végezzük.

A this elmentése

```
var MyClass = function () {
    var self = this;
    var priValt = 3;
    var priFv = function () { self.pubFv(); };
    self.pubValt = 5;
    self.pubFv = function () {
        alert('Privát: ' + priValt);
        alert('Publikus: ' + self.pubValt);
    };
    return self;
};
var c = new MyClass();
c.pubFv();
```



Arrow function

Kliensalkalmazások

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Gincsai Gábor
gincsai@aut.bme.hu

Arrow function

- Függvényeket tömörebben leírhatjuk vele, így olvashatóbb lesz a kód.

```
nev => console.log(nev)

() => console.log("alma")

(varos, utca) => {
  let cim = varos + utca
  console.log(cim)
}
```

Arrow function példák

```
materials.map(function(material) {
    return material.length;
}); // [8, 6, 7, 9]

materials.map((material) => {
    return material.length;
}); // [8, 6, 7, 9]

materials.map(material => material.length); // [8,
6, 7, 9]
```

Az Arrow functionnek nincs saját this-e

- JavaScriptben minden függvénynek van saját *this*-e, ami időnként megnehezíti a kódolást.
  - Alapértelmezés szerint a *this* a függvény kontextusára mutat.
  - jQuery-s eseménykezelőben a *this* a DOM objektum, mert átállítja a jQuery.
- Arrow function esetén nincs saját *this*, tehát nem lehet átállítani, hogy magára mutasson.

this példa (hibásan)

```
function Person() {
// Person() konstruktor a `this`-t a
// létrehozott példányra állítja.
this.age = 0;

setInterval(function growUp() {
// Nem strict módban, a growUp() függvény a `this`-t
// globális objektumként definiálja, ami nem azonos
// azzal a `this`-szel, amit a Person() konstruktor állít be.
            this.age++;
    }, 1000);
}
```

this példa (javítva)

```
function Person() {
      var that = this;
      that.age = 0;

      setInterval(function growUp() {
      // A callback a `that` változóra mutat,
      // aminek az értéke az elvárt objektum.
            that.age++;
      }, 1000);
}
```

Arrow functionben használható a this

```
function Person(){
    this.age = 0;

    setInterval(() => {
        this.age++;
        // |this| helyesen a Person
        // objektumra mutat
    }, 1000);
}
```

Arrow function törzse

- Nem kell kiírni a returnt

```
> var func = x => x * x;
```

- Ha több soros a kód, akkor kell return

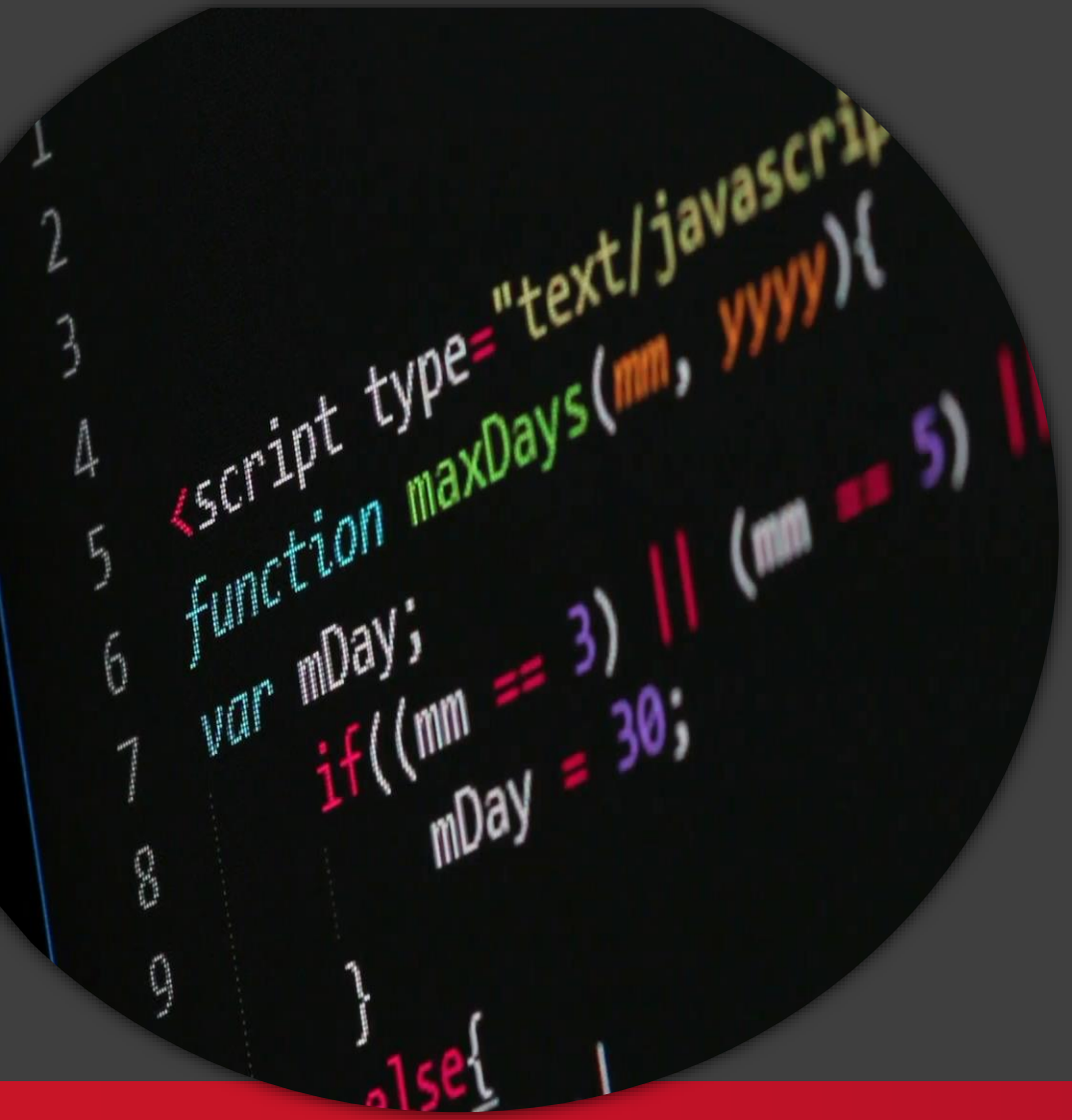
```
> var func = (x, y) => { return x + y; };
```

- Objektum nem adható vissza simán

```
> var func = () => { foo: 1 };
> var func = () => { foo: function() {} };
```

- Zárójelezni kell

```
> var func = () => ({foo: 1});
```



ES6 osztályok és öröklés

Kliensalkalmazások

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Gincsai Gábor
gincsai@aut.bme.hu

Prototípus alapú öröklés

- Egy objektumnak lehet legfeljebb 1 prototípusa:
  - A prototípus egy másik objektum, vagy null, ha nem létezik.
- Ha egy objektum egy property-jét olvasni akarjuk, de az nem létezik, akkor megnézzük, hogy az benne van-e a prototípusban és ha igen, akkor azt adjuk vissza.
  - Írni nem tudjuk a prototípus property-jét.
- Prototípus objektum elérése
  - `__proto__ (getter, setter)`
  - `Object.getPrototypeOf(obj)`
  - `Object.setPrototypeOf(obj)`

Prototípus objektum elérése

```
let o = {};


console.log(o.toString()); // --> "[object Object]"
console.log('toString' in o.__proto__); // --> true


o.__proto__ = null;
console.log(o.toString());
// --> TypeError: o.toString is not a function
```

Prototípus objektum elérése

```
let parent = {
    name: 'x',
    greet() { console.log(this.name); }
};
let child = {
    id: "ABC123"
};

Object.setPrototypeOf(child, parent);
child.__proto__ = parent;
```

Prototípus alapú öröklés

- A prototípus tehát egyfajta öröklést tesz lehetővé.
- Betartandó szabályok:
  - Nem lehet ciklus a prototípus öröklésben
  - A protoípusok csak objektumok, vagy null lehet, primitív típus nem.
- A `this` értékét a prototípus nem befolyásolja, azaz ha egy objektumon keresztül egy prototípus property-jét érjük el, attól még a this az aktuális leszármazott gyerekre mutat.

Konstruktor függvény

- Konstuktor függvényeknek van prototype propertyje
  - Nem azonos a `__proto__` property-vel!!!
- Amikor létrehozunk egy új objektumot a konstruktor függvénnyel, akkor annak a prototípusa ez az objektum lesz.
- Ez lehetővé teszi, hogy a prototípust akár futási időben újabb property-kkel egészítsük ki, amelyek ezután minden öröklött helyen elérhetővé válnak.

Konstruktor függvény

```
function User(name) {
    this.name = name;
}

let user = new User();

User.prototype.greet = function () { console.log("hello"); }
user.greet();
```

Factory minta

```
function User(name) {
    return {
      name: name,
      greet() {
      console.log(this.name);
      }
    };
  }
```

Osztályok létrehozása ES6-ban

- Az ECMAScript6 lehetőséget ad osztályok létrehozására is, a `class` kulcsszó segítségével. Ez a szintaxis egy kicsivel közelebb áll az OO nyelvekhez, mint a korábban megismert.
- Továbbra is prototípus alapú öröklés marad.
- Csak egy sintaktikai édesítőszer a class kulcsszó.

A class kulcsszó ES6-ban

```
class Point {
  constructor(x, y) {
    this.x = x; this.y = y;
  }
  toString() {
    return '(' + this.x + ', ' + this.y + ')';
  }
}
var p = new Point(25, 8);
console.log(p.toString())          // '(25, 8)'
```

Backtick `

- A fenti példában a toString()-et átláthatóbban is megírhatjuk az alábbi módon.

```
return `(${this.x}, ${this.y})`;
```

- Így nincs szükség a szintring összefűzésekre.
- Ügyeljünk rá, hogy itt nem aposztróffal kell kezdeni a sztringet hanem backtick (`-tel)

Fontos a sorrend

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

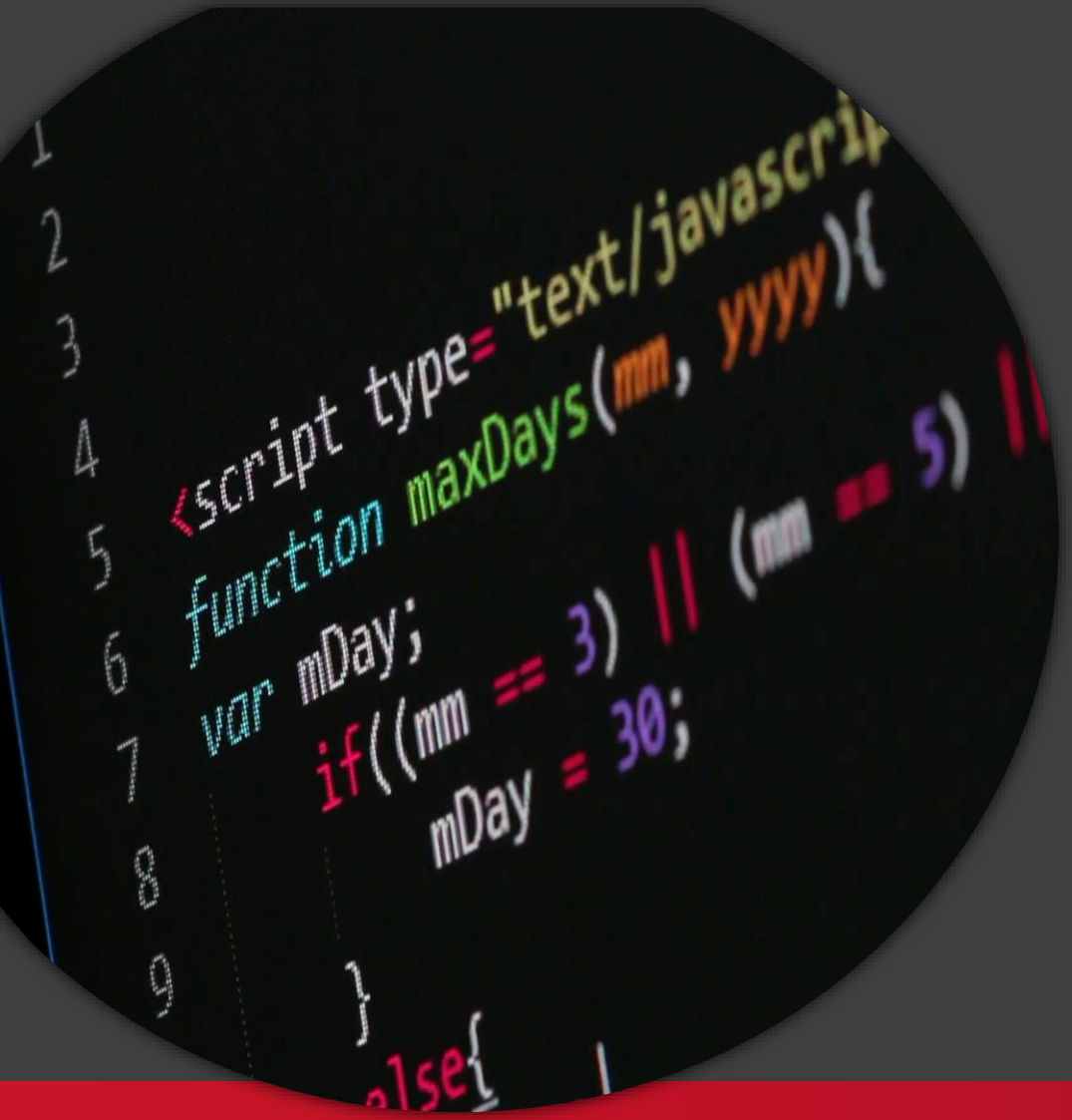
```
foo();                          // Működik
function foo() { alert("hi"); }


new Foo();
// ReferenceError: can't access lexical declaration `Foo'
before initialization
class Foo {}
```

Származtatás

- Arra is lehetőségünk van, hogy az osztályok származzanak egymásból. Ehhez az extends kulcsszót kell használnunk. Az ősosztály függvényét például a super.toString() a kontruktorát pedig a super() segítségével tudjuk meghívni.
- Ügyeljünk arra, hogy a konstruktornál az előtt kell meghívni a super()-t mielőtt a this-t használjuk!

Példa

```
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

Kliensalkalmazások

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Gincsai Gábor
gincsai@aut.bme.hu

A JavaScript alapvetően aszinkron.

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}

loadScript('/my/script.js'); // sayHello() impl.
sayHello();   // Error: nincs ilyen fv??
```

Callback

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('/my/script.js', function() {
    sayHello(); // Működik, a callback betötés után fut.
});
```

Ha sok a callback....

```
loadScript('1.js', function(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...
          }
        });
      }
    })
  }
});
```

Forrás: https://javascript.info/callbacks

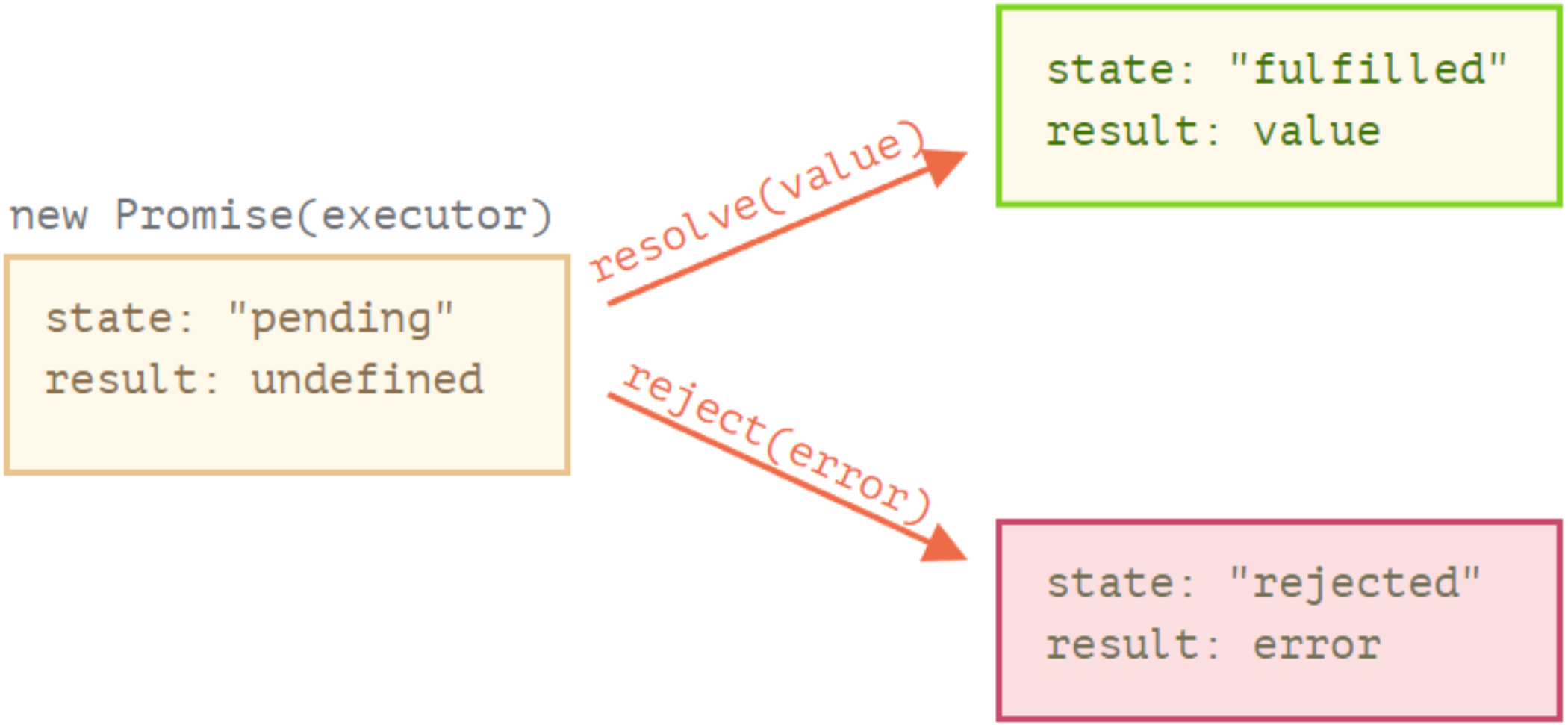
Promise

- A Promise egy olyan objektum, ami majd a jövőben visszaad egy értéket, de nem most.
- E miatt tökéletes aszinkron kérések kezelésére.
- Három állapota van
  - Pending – függőben van
  - Fulfilled – sikeresen lefutott
  - Rejected – hibára futott
- A promise mindig Pendig állapotból indul és Fulfilled vagy Rejected állapotban ér véget.

Promise példa

```
let completed = true;

let learnWeb = new Promise(function (resolve, reject) {
    if (completed) {
        resolve("I have completed learning Web.");
    } else {
        reject("I haven't completed learning Web yet.");
    }
});
```

Promise állapot átmenetei



Promise eredményének feldolgozása

- .then(success, error)
  - Ha lefutott a Promise akkor hívódik meg.
  - Ha sikeresen futott le akkor a success handler hívódik
  - Ha sikertelenül akkor az error handler (opcionális)
- .catch(f)
  - Csak akkor fut le ha a Promise hibával tért vissza
- .finally(f)
  - Minden esetben lefut, ha sikeres ha sikertelen a Promise, de az eseménykezelőben nem tudjuk eldönteni, hogy sikeresen vagy sikertelenül futott le.
  - Tipikusan takarításra használjuk. Pl.: loading indikátor eltűntetése.

Promise példa (then, catch, finally)

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

learnWeb.catch(alert);

learnWeb.finally( () => /* Stop loading */ )
```

loadScript példa Promise használatával

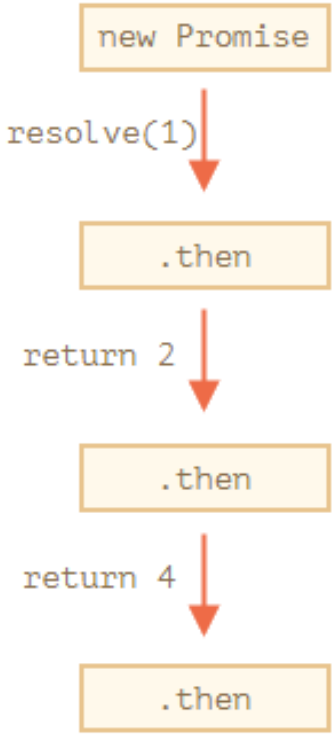
```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Error: ${src}`));

    document.head.append(script);
  });
}
```

loadScript példa Promise használatával

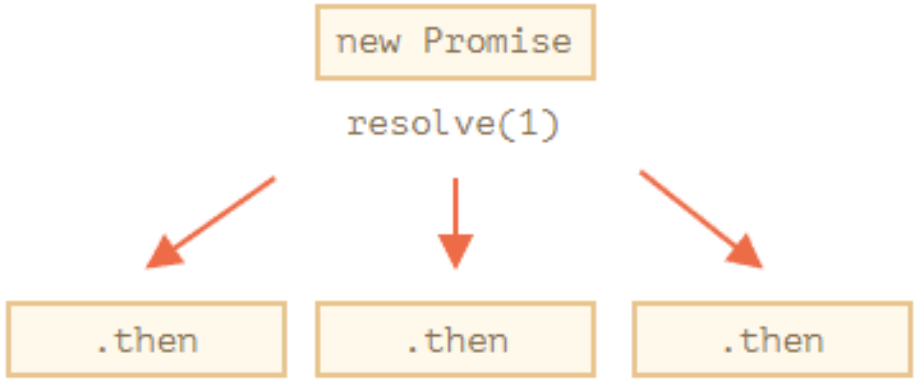
```
let promise = loadScript("https://cdnjs.cloudflare.com/ajax/
                          libs/lodash.js/4.17.11/lodash.js");

promise.then(
  script => alert(`${script.src} is loaded!`),
  error => alert(`Error: ${error.message}`)
);

promise.then(script => alert('Another handler...'));
```

Promise chain

```
promise.then(...).then(...).then(...)
```



A fenti kód nem azonos azzal,
ha a promiseon három thent hívunk.




```
promise.then(...);
promise.then(...);
promise.then(...);
```

Promise API

- Promise.all(promises)
  - Megvárja, hogy az összes Promise befejeződjön.
  - Ha bármelyik hibával ér véget, akkor az egész hibával tér vissza.
- Promise.allSettled(promises)
  - Megvárja, hogy az összes Promise befejeződjön
  - Visszaadja, hogy melyik volt sikeres és melyik hibás.
    - status: „fulfilled” vagy „rejected”
    - value ha sikeres, reason ha sikertelen.
- 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.
  - Ha mindegyik sikertelen akkor AggregateErrort ad vissza.

async / await

- Segítségével kényelmesebben kezelhetjük az Promise-okat.
- A függvény előtt lévő async azt jelenti, hogy a függvény egy Promise-sal tér vissza.
  - Ha nem promise-sal térne vissza a függvény, akkor becsomagolja egy promise-ba.

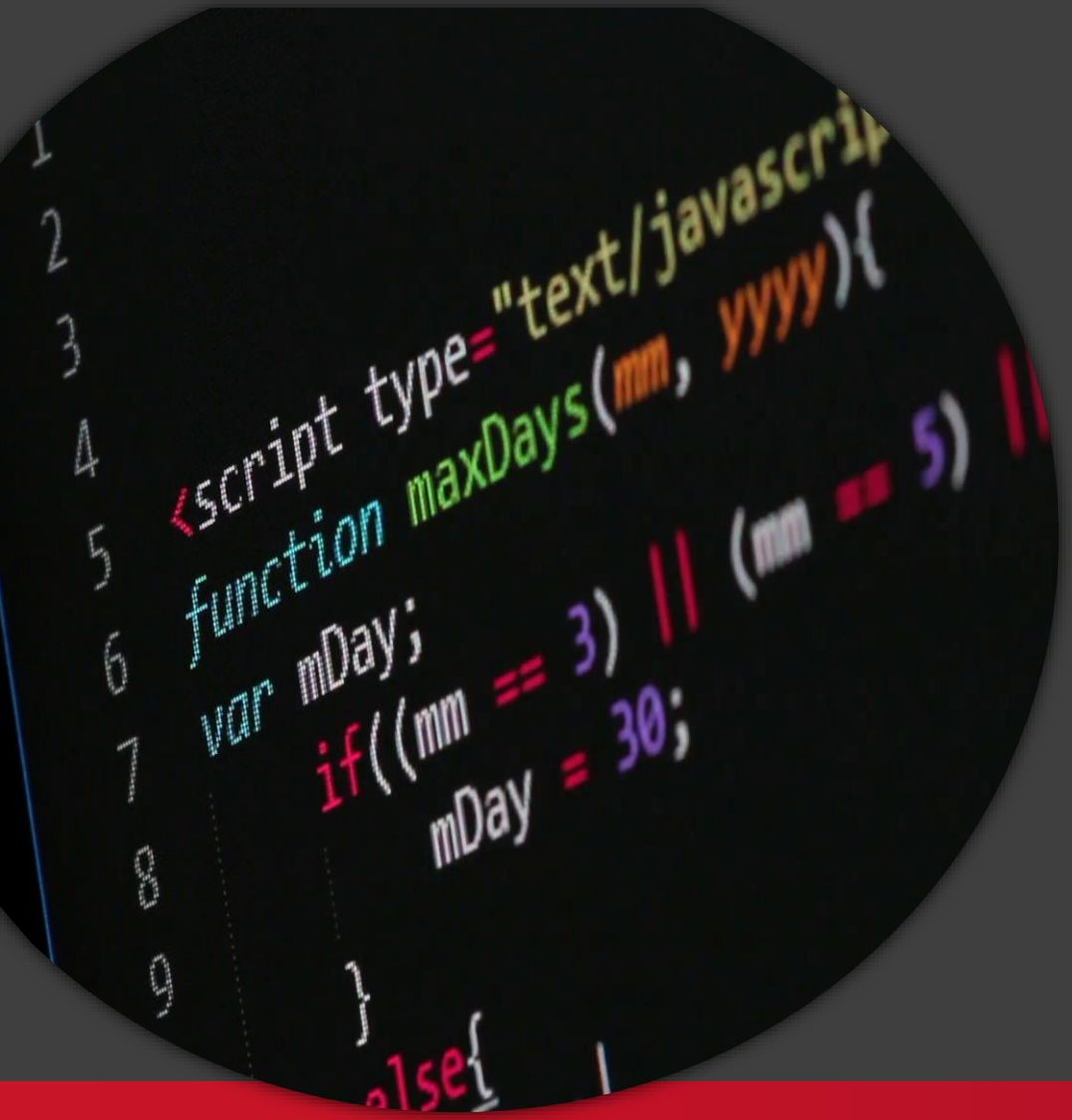
```
async function f() {
  return 1;
}
```

```
async function f() {
  return Promise.resolve(1);
}
```

```
f().then(alert); // 1
```

await

```
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();
```



Fetch API

Kliensalkalmazások

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Gincsai Gábor
gincsai@aut.bme.hu

Fetch API

- A fetch() segítségével hálózati kéréseket küldhetünk a szerver felé.
- Korábban ezeket a kéréseket az XmlHttpRequest segítségével lehetett megoldani,
- A Fetch API a modern böngészőkben megtalálható, polyfill is van a régebbi böngészőkhöz.
- Támogatja a CORS-t tehát tetszőleges szerver felé indíthatunk kéréseket.
- Promise-sal tér vissza.

Fetch API

- A Promise csak akkor reject-elődik ha hálózati hiba van.
- A HTTP státuszkódtól függetlenül sikeresen tér vissza, ha a szerver válaszolt.
  - A válaszban van egy Ok tulajdonság, ami jelzi, hogy a kérés sikeres-e
    - 200-299 közötti státuszkódokra true-ra állítja, egyéb esetben false-ra.
  - Cross-origin sütiket nem fogja elküldeni.

```
let promise = fetch(url, [options])
```

Fetch API

- A böngésző azonnal elindítja a kérést és egy Promise-t ad vissza, amiből majd az eredmény el lehet érni.
- A választ két lépésben lehet kinyerni.
  1. A fetch Promise a beépített Response osztályt adja vissza, amiben a szervertől visszakapott Header-ök találhatók.
     - Status: HTTP státuszkód.
     - Ok: Sikeres-e a kiszolgálás (Státusz kód 200-299)
  2. A response-ból a body-t egy újabb promose-sal kapjuk meg amit pl a json() függvény segítségével lehet elérni.

Fetch API 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);
}
```

Fetch API - Post

```
let user = { name: 'John', surname: 'Smith' };

let response = await fetch('/article/fetch/post/user', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json;charset=utf-8'
    },
    body: JSON.stringify(user)
});

let result = await response.json();
alert(result.message);
```