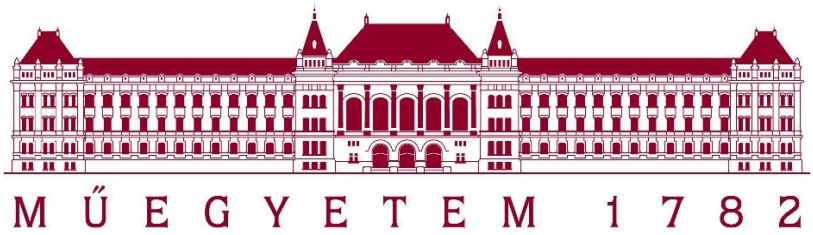

Mobile and Web Development

Department of Automation and Applied Informatics

JavaScript

Created: 1/2024/2025

Dr. Mohammad Saleem

msaleem@aut.bme.hu

JavaScript

Introduction

- Created by Brendan Eich, a software engineer of Netscape
  - First it was named Mocha, then LiveScript
  - It is called JavaScript from 1995, after the Netscape and Sun license agreement
  - Today „JavaScript" is a registered trademark of Oracle
- The original goal was to create a language that extends Java and can also be used by non-professional developers to create interactive web pages
  - Just like Microsoft Visual Basic and C++.
- JScript (1996 August)
  - New dialect published by Microsoft in order to avoid legal problems
  - JScript 9.0 used in Internet Explorer 9 is compatible with JavaScript 1.8.1 and ECMA-262 5.
- ECMAScript (version 1 published in July 1996, version 6 published in July 2015)
  - ECMAScript is a *specification* for scripting languages
  - Standardized by ECMA: ECMA-262 (the concrete name of the standard)
  - Contains addition features compared to both JavaScript and JScript
  - *„ECMAScript was always an unwanted trade name that sounds like a skin disease."*

Brendan Eich

- Eich, however decided that Java was too complicated with all its rules and so set out to create a simpler language that even a beginner could code in. This is evident in such things like the relaxing of the need to have a semicolon.
- About a year or two after JavaScript's release in the browser, Microsoft's IE took the language and started making its own implementations such as JScript. At the same time, IE was dominating the market and not long after Netscape had to shut its project.
- Before Netscape went down, they decided to start a *standard* that would guide the path of JavaScript, named ECMAScript (ECMA-262).
- JavaScript is the most popular ☞ **implementation** of the ECMAScript Standard.

Differences between Java and JavaScript

- Dynamically-typed languages perform type checking at runtime, while statically typed languages perform type checking at compile time.
- In computer programming, a programming language is strongly typed if it demands the specification of data types. A programming language is loosely typed, or weakly typed,

when it does not require the explicit specification of different types of objects and variables.

- Java bytecode is a low-level representation of Java source code that is designed to be executed by the Java Virtual Machine (JVM). It serves as an intermediary between the source code written by developers and the machine code executed by the CPU.
- **Prototype-based Programming**: Does not use classes. Instead, objects are created by cloning existing objects, which serve as prototypes.

```
// Create a prototype object
const animalPrototype = {
  speak: function() {
    console.log(`${this.name} makes a noise.`);
  }
};

// Create a new object based on the prototype
const dog = Object.create(animalPrototype);
dog.name = 'Dog';
dog.speak(); // Output: Dog makes a noise.
```

Java	JavaScript
Static types	Dynamically typed
Strongly typed	Weekly typed
Loaded from byte code	Loaded from source code
Class-based objects	Prototype-based objects

Variations of JavaScript

- ActionScript
  - > Dialect created by Macromedia (later changed to Adobe) to program Flash
- TypeScript
  - > Extension created by Microsoft
  - > Adds types and real class-based object orientation to JavaScript
  - > Builds on the features set and the syntax of ECMAScript 6

JavaScript, CSS and HTML

- HTML (HyperText Markup Language): content
- CSS (Cascading Style Sheets): presentation
- JavaScript: interactivity

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

HTML and JavaScript

- The <script> reference always has to be used as a non self-closing tag
- JS references can also be place inside of the <body> tag
  - > For performance reasons it is placed at the end of the page (before the closing </body>)
- JS code can also be embedded into HTML files by enclosing the code to <script> tags
  - > Maintainability
  - > Separating functionalities
  - > For performance reasons it is not advised
- Events of HTML elements can also be handled in the HTML file
  - > Code gets hard to read

Elements of the language

Comments	// single line; /* multi-line */
Arithmetic operators	+, -, /, *, %, ++, --
Assignment operators	=, +=, -=, *=, /=, %=
Bitwise operators	&, |, ^, ~, <<, >>, >>>
Logical operators	||, &&
Comparison operators	==, ===, !=, !==, <, >, <=, >=
Conditions	if..else, switch..case (break, default), instanceof, typeof, .. ? .. : ..
Loops	for, for..in, while, do..while, break, continue
Error handling	try..catch..finally, throw
Handling objects	new, delete
Functions	function, return

- == equal to
- === equal value and equal type

Built-in types

- String
- Number
- Boolean
- Null
- Undefined
- Object
-

`undefined` is used in JavaScript to indicate that a variable exists but has not been assigned a value. It differs from `null`, which is used to represent an intentional absence of any value. Understanding `undefined` helps avoid common pitfalls related to uninitialized variables and unexpected property lookups.

- The first five are the primitive data type
- The first three has got an Object typed wrapper

Primitive: A primitive is a basic, immutable data type that represents a single value. JavaScript's primitive data types are `number`, `string`, `boolean`, `undefined`, `null`, `symbol`, and `bigint`. **[If you assign a new value to a variable holding a primitive, you're creating a new value in memory.]**

Object: An object is a complex data structure that can store multiple values and more complex entities. Objects are mutable and can contain properties and methods.

Boolean type

truthy	falsey
true	false
'0'	0
123 vagy -123	Nan
'something'	'', (empty string)
[]	null
{}	undefined

In JavaScript, certain values are considered "truthy" or "falsy" when evaluated in a Boolean context:

- **Truthy values**: Any value that is not `false`, `0`, `-0`, `0n` (BigInt zero), `""` (empty string), `null`, `undefined`, or `NaN` is considered "truthy".

```
console.log(Boolean(1)); // Output: true
console.log(Boolean(0)); // Output: false
```

```
console.log(Boolean('Hello')); // Output: true
console.log(Boolean('')); // Output: false
console.log(Boolean(null)); // Output: false
console.log(Boolean(undefined)); // Output: false
```

window object

- In case of a JavaScript code that runs in a browser the window object is used to identify the window frame of the current page.
- As it is a top level object it provides many general properties and events (the list is not complete):
  - > document, history, location, navigator, screen, status, applicationCache, console, …
  - > alert(), confirm(), focus(), open(), close(), print(), scrollTo(), setInterval(), setTimeout(), …
  - > onload, onbeforeunload, ondragdrop, onerror, onresize, …

Where are the variables created?

- Variables defines with the var keyword belongs to the window object

```
var a = 'Hello world';
alert(window.a);
```

- Functions also belong to the window

```
function say(name) {
    alert('Hello ' + name)
}
window.say('World');
```

- What happens when a page has got many JavaScript file references and all these files create a separate variable named index?
  - > It is legal, as they are loaded from different files
  - > The one loaded later will overwrite the earlier ones.
  - > The code won't start.
  - > They will be created in different namespaces -> no conflict

Global scope pollution

When variables or functions are declared without any kind of encapsulation (such as inside a function, block, or module), they become properties of the global `window` object. This can result in:

Name Collisions: Variables or functions from different parts of the codebase may have the same name, leading to overwriting and unintended behavior.

Memory Issues: Too many global variables can consume memory unnecessarily and affect performance.

Difficulty in Maintenance: Code becomes harder to maintain and debug when variables and functions are spread across the global scope.

```
// Declaring variables in the global scope
var userName = "Alice";
var userAge = 30;

// Any part of the code can modify these variables, leading to potential
conflicts
function changeUserName() {
  userName = "Bob"; // This overwrites the original userName
}

changeUserName();
console.log(userName); // Outputs "Bob"
```

```
// Script 1
var theme = "dark";

// Script 2
var theme = "light"; // This overwrites the 'theme' from Script 1
```

Local variables

- Variables created inside a function are local and can only be accessed locally – just like in many other programming languages

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

Var vs. Let vs. Const

- **var** (depricated): function scoped **undefined** when accessing a variable before it's declared
- **let**: block scoped **ReferenceError** when accessing a variable before it's declared
- **const**: same as let, but you can't reassign it to a new value.

- Where is the variable created if the variable is created without the var keyword?
  - Not created, we get an error
  - Inside the function
  - **Inside the window object**
    - Only works for assignment, like **x = 1**. It first tries to resolve **x** against scope chain. If it finds it anywhere in that scope chain, it performs assignment; if it doesn't find **x**, only then does it create x property on a global object (which is a top level object in a scope chain).

The scope of the variables

- No explicit way to define public and private members
- Have to play with functions
  - > In JavaScript the visibility of variables defined with the var keyword is exclusively determined by function scopes.

What is the output?

```
function f() {
    if (true) {
        var x = 8;
    }
    alert('In: ' + x);
}
f();              // 8
```

```
function f() {
    if (true) {
        let x = 8;
    }
    alert('In: ' + x); // x is not defined
}
f();
alert(' Out: ' + x);  // x is not defined
```

```
function f() {
    if (true) {
        var x = 8;
    }
    alert('In: ' + x);
}
f();
alert(' Out: ' + x); // x is not defined
```

Finding a variable

- It happens from inside out.

- If the variable is not found in a the local function the runtime engine searches for it one function higher, etc, until it either reaches the global scope (window) or finds the variable.
- Local variables hide the external ones
  - Shadowing

```
var x = 5;
function outer() {
    var x = 8;
    function inner() {
        var x = 9;
        alert('In: ' + x);
    }
    inner();
    alert('Out: ' + x);
}
outer();
alert('Even outer: ' + x);
```

Functions as types

- In JavaScript, functions are full-fledged types that provide us with many interesting facilities

```
var outer = function () {
    var x = 8;
    var inner = function () {
        alert(x);
    };
    return inner;
};
var b = outer();
b();
```

Closure

- Functions are embedded and the outer function makes the inner function accessible for the outside world
- The inner function preserves the state that belongs to its creation.
- When a function makes an internal function available for the external world that is called a **closure**, that means the internal function and its preserved state together.

A closure is created whenever a function is defined inside another function. The inner function maintains access to the variables of the outer function, even after the outer function has returned.

```
function createCounter() {
  let count = 0; // count is private to the closure
  return {
    increment: function() {
      count++;
      return count;
    },
    decrement: function() {
      count--;
      return count;
    },
    getCount: function() {
      return count;
    }
  };
}

const counter = createCounter();
console.log(counter.increment()); // 1
console.log(counter.increment()); // 2
console.log(counter.decrement()); // 1
console.log(counter.getCount()); // 1
```

Anonymous functions

- In the previous example it is unnecessary to name the inner function

```
var outer= function () {
    var x = 8;
    x++;
    return function () {
        alert(x);
    }
};
var b = outer();
b();
```

Self-executing (self-invoking) functions

- Another interesting feature of JS is to get functions executed automatically

```
var fv = function (name) {
    alert('Hello ' + name);
};
fv('World');
```

- We only named it to be able to execute it

```
(function (name) {
    alert('Hello ' + name);
})('World');
```

Module design pattern

- We can wrap our code into a custom module that helps avoiding global scope pollution.
- In JavaScript, a module is essentially an object that exposes certain methods and properties, while keeping the internal implementation hidden. The Module Pattern allows you to create private variables and functions that cannot be accessed directly from outside the module, thus ensuring data encapsulation.
- By the help of the constructor function we can pass parameters to the module (import) and can specify after the return keyword what is public for the outside world (export).
- The code of a module can be placed into multiple files, the public members will access each other

```
// Module Pattern Example
var myModule = (function() {
  // Private variables and functions
  let privateVar = "I'm private!";

  function privateFunction() {
    console.log(privateVar);
  }

  // Public API (methods that can be accessed outside the module)
  return {
    publicMethod: function() {
      privateFunction(); // Accessing private function inside the module
    }
  };
})();

// Accessing public method
myModule.publicMethod(); // Output: I'm private!

// Attempting to access private variable will fail
console.log(myModule.privateVar); // undefined
```

The new operator

- By omitting the self-invoking function call and applying the new operator we can achieve a class-like behavior as well.

```
var MyClass = function () {
    var priVar = 3;
    var priFn = function () { alert('Private!'); };
    return {
        pubVar: 5,
        pubFn: function () { alert('Public'); }
    };
};

var c = new MyClass();
c.pubFn();
alert(c.pubVar);
```

The this keyword

- There is a this keyword in JavaScript as well, that usually point to the object the function was called on.
- In the following example the **function belongs to the window** object so the this points to **the window**

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

Modifying the previous example by using the new operator changes the behavior

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

- The new operator creates a new Object with the given constructor function and sets the **this** to point to **this object**.
    - > The new operator returns this object (when the constructor function doesn't have a return value).

This is not always that *this*

- *This* may sometimes not point to the object in which it is used. E.g. in the event handler of a button this usually points to the button DOM element.

> It could be confusing that *this* means something else here.

- To avoid potential errors resulting from misunderstanding it is straightforward to avoid using the *this*, use something else (e.g. that, self) instead.

`self = this` pattern: When working with nested functions or asynchronous callbacks (e.g., in `setTimeout` or event listeners), the value of `this` can change unexpectedly. The `self = this` pattern is used to keep a reference to the correct value of `this`.

that = its a variable (can be named anything) to keep the reference of ***this*** so that it can be used inside.

Saving *this*

```
var MyClass = function () {
    var self = this;
    var priVar = 3;
    var priFn = function () { self.pubFn(); };
    self.pubVar = 5;
    self.pubFn = function () {
        alert('Priate: ' + priVar);
        alert('Public: ' + self.pubVar);
    };
    return self;
};
var c = new MyClass();
c.pubFv();
```

Creating classes in ES6

```
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 `

- The string concatenation of the previous example can be written nicer.

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

- Use backtick instead of apostrophe

The order is important

- Functions can be called before they are declared. However you cannot do the same with classes.

```
function say(name) {
    alert('Hello ' + name)
}
window.say('World');
```

Inheritance

- Classes can derive from each other by using the *extends* keyword. A method of the base class can be called, e.g. super.toString(), or the constructor: super()

```
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
```

USEFUL LINKS AND REFERENCES:

Mobile and Web Development