Web Application Development
(LTAT.05.004)

JAVASCRIPT - II

MOHAMAD GHARIB
UNIVERSITY OF TARTU
JavaScript history - Recap

1993
Mosaic

1994
Netscape

1995
- Mocha
- liveScript
- JS
- Brendan Eich at Netscape

1995
IE

1997
ECMA-1 (European Computer Manufacturers Association) - ES1

1998
ECMAScript 2 (ES2)

2000-2008
ES4 - Never released

2000-2005
AJAX

2001
JSON

2009
ES5

2015
ES6 / ECMAScript 2015

2016
ECMAScript 2016

2020
ECMAScript 2020

2020
There are **five different data types** that can contain values:

- Number
- String
- Object
- Boolean
- Function

There are **two data types** that cannot contain values:

- Null → Object
- Undefined

You can use the `typeof` operator to find the data type of a JavaScript variable.
JavaScript - II
Before we start ...

What do you need to know before starting?
• HTML, CSS and Document Object Model.

What we will cover:
• What JavaScript is,
• JavaScript data types, variables, loops, etc.
• How we can use JavaScript to dynamically create, update our page content, etc.

What can JavaScript do?
JavaScript can dynamically modify/manipulate an HTML page, validate user input (Form validation), react to events and user input (special effects), basic math calculations, etc.

How this lecture contributes to the course

HTML
The structure of the webpage

CSS
The styling of the webpage

JavaScript
Makes a webpage interactive
Functions
A JavaScript function is a block of code designed to perform a particular task, and it is executed when "something" invokes it.

Functions can pass parameters. Parameters can be initialized with default values if no value or undefined is passed.

Functions can return values. Functions can be user defined or built-in.

```
//function declaration
function sayMyName(name = 'No name was passed') {
    return name
//return stops the execution and returns a value. There is a virtual semicolon after it
}

//function invocation
sayMyName('Sarah');
```

```
//built-in function
console.log(sayMyName('Sarah'))
```
The **rest parameter** allows a function to accept an indefinite number of arguments as an array.

A **rest parameter** is required to be prefixed with ... (three dots).

Only one **rest parameter** is allowed, and it should be the last parameter in a function definition, which will cause all remaining parameters to be placed within an array.

```javascript
function sayMyNames(fname, mname, lname) {
  return `My name is ${fname}${mname}${lname}`;
  //Template literals allow variables in strings
}
sayMyNames('Sarah', 'J.', 'Jones');

// rest parameters
function sayMyNamesRP(...names) {
  return names;
}
sayMyNameRP('Sarah', 'J.', 'K.', 'Jones');
```
JS supports **nested functions**, which are **functions** that are defined inside or within another function.

Nested functions have access to the **scope** containing them.

```javascript
// Ex 1
function sayHiToMe(fName, lName) {
  let msg = `Hi`;
  // Nested Function
  function composeMsg() {
    msg = `${msg} ${fName} ${lName}`;
  }
  composeMsg();
  return msg;
}
console.log(sayHiToMe("Jane", "Doe"));
```
JS supports nested functions, which are functions that are defined inside or within another function.

Nested functions have access to the scope containing them.

```javascript
// Ex 2
function sayHiToMe(fName, lName) {
    let msg = `Hi`;
    // Nested Function
    function composeMsg() {
        return `${msg} ${fName} ${lName}`;
    }
    return composeMsg();
}
console.log(sayHiToMe("Jane", "Doe"));
```
JS supports nested functions, which are functions that are defined inside or within another function.

Nested functions have access to the scope containing them.

// Ex 3
function sayHiToMe(fName, lName) {
    let msg = `Hi`;
    // Nested Function
    function composeMsg() {
        function composeName() {
            return `${fName} ${lName}`;
        }
        return `${msg} ${composeName()}`;
    }
    return composeMsg();
}
console.log(sayHiToMe("Jane", "Doe"));
Scopes specify current context of execution. In which, values and expressions can be referenced.

If a variable or expression is not "in the current scope", then, it is unavailable for use.

```javascript
var a
let b

function sayHiToMe(fName, lName) {
  let msg = `Hi`;
  //msg can only be used in sayHiToMe

  return composeMsg();
}
```
JS Scopes

Global vs Function scopes

Global scope

- Any variable that is defined outside any function has a Global scope.
- Global variables can be accessed from anywhere in a JavaScript program.
- Variables declared with `var` or `let` or constants (`const`) are quite similar when declared outside a block.

```javascript
//global scope
var a //global variable
let b //global variable

function sayHiToMe(fName, lName) {
    let msg = `Hi`;
    return composeMsg();
}

console.log(`${a}, ${b}`); // 10, 20
```
JS Scopes

Global vs Function scopes

Function scope

- Each function creates its own new scope.
- Any variable that is defined inside a function are **not accessible** (visible) from outside the function.
- Variables declared with `var` or `let` and **constants** (const) are quite similar when declared inside a function, i.e., they all have Function scope.

```
//global scope
var a //global variable
let b //global variable

console.log(`${a}, ${b}`); // 10, 20

//function scope
function sayHiToMe(fName, lName) {
    let msg = `Hi`;
    //msg can only be used in sayHiToMe
    return composeMsg();
}

console.log(`$\{a\}, $\{b\}`); // 10, 20
console.log(msg); // error
```
Scopes can also be layered in a hierarchy, so that child scopes have access to parent scopes, but not vice versa.

```javascript
//global scope
var a //global variable
let b //global variable

console.log(`${a}, ${b}`); // 10, 20
console.log(msg); // error

console.log(sayHiToMe("Jane", "Doe"));

//function scope
function sayHiToMe(fName, lName) {

    let msg = `Hi`;
    //msg can only be used in sayHiToMe

    return composeMsg();
}

function composeMsg() {

    function composeName() {
        return `_${fName} ${lName}`;
    }

    return `${msg} ${composeName()}`;

    return composeMsg();
}
```

An anonymous function is a function that does not have any name associated with it.

An anonymous function is not accessible after its creation, it can only be accessed by a variable it is stored in as a function as a value.

```javascript
function sayMyName(name) {
    return name;
}
sayMyName(‘Sarah’);

// Anonymous function
let sayMyName = function(name) {
    return name;
} sayMyName(‘John’);
```
Arrow functions were introduced in ES6.

Arrow functions allow us to write shorter function syntax.

If the function has only one statement, and this statement returns a value, you can remove the brackets and the return keyword.

```javascript
function sayHi() {
  return 'hi';
}
sayHi();

// arrow function
let sayHi = () => {
  return 'hi';
}
sayHi();

// arrow function - shorter version
let sayHi = () => 'hi';
// we can use _ instead of ()
sayHi();
```
JS Arrow Functions – param(s)

If the function have one parameter, you can remove the parentheses.

If the function have parameters, you pass them inside the parentheses.

```javascript
// Arrow function – one param
let funArrow = (name) => {
    return name;
}
//let funArrow = name => name;
funArrow('John Arrow');

// Arrow function – two or more params
let funArrow = (fName, lName) => {
    return `My name is ${fName} ${lName}`;
}
funArrow('John', 'Arrow');
```
A **callback function** is a function passed into another **function** as an **argument**, which is then invoked inside the outer function to complete a task.

**Note:** this example is a **synchronous callback**, as it is executed immediately.

```javascript
let calledFun = () => {
    console.log('Hello');
}

function synchCallBck(string, callback) {
    callback();
    console.log(string);
}

synchCallBck('World', calledFun)
// Hello
// World
```
A higher-order function is a special category of functions that accepts functions as parameters and/or returns a function.

There are plenty of higher-order functions.

Popular examples are the Array methods such as map(), filter(), reduce(), which takes another function as an argument to apply it to the elements of an array.
JS Higher-order functions – map()

The `.map()` method creates a new array populated with the results of calling a provided function on every element in the calling array.

The `.map()` method does not change the original array.

Parameters:

- `function()` - Required - to be run for each array element.

- `currentValue` – Required - the value of the current element.

```javascript
//Syntax: array.map(function(currentValue, index, arr), thisValue)

let numArr = [1, 2, 3, 4, 5];
arrMap = numArr.map((a) => a + 1);
console.log(arrMap); // [2, 3, 4, 5, 6]
```
JS Higher-order functions – filter()

The .filter() method executes a callback function on each element in an array, which returns either true or false.

The returned array is a new array with any elements for which the callback function returns true.

Parameters:

- function() - Required - to be run for each array element.
- currentValue – Required - the value of the current element.

```javascript
let numArr = [1, 2, 3, 4, 5];
arrFilter = numArr.filter((a) => a % 2 === 0);

console.log(arrFilter); // [2, 4]
```

//Syntax: array.filter(function(currentValue, index, arr), thisValue)
The `.reduce()` method iterates through an array and returns a single value.

Parameters:

- **function()** - Required - to be run for each array element.
- **accumulator** - Required - accumulates the return values.
- **currentValue** - Required - the value of the current element.
- **initialValue** - optional - passed to callback() on first call.

```javascript
let numArr = [1, 2, 3, 4, 5];
arrReduce = numArr.reduce((acc, cur) => acc + cur);
console.log(arrReduce); // 15
```
JS Higher-order functions – forEach ()

The `.forEach()` method executes a callback function on each of the elements in an array in order.

Parameters:

- `function()` - **Required** - to be run for each array element.
- `currentValue` – **Required** - the value of the current element.

```javascript
let numArr = [1, 2, 3, 4, 5];
arrForEach = numArr.forEach((a) => console.log(a)); // 1 2 3 4 5
console.log(arrForEach); // undefined
```

//Syntax: array.forEach(function(current Value, index, arr), thisValue)
Objects
Objects

Objects are one of the JavaScript data types, and can be used to store various **keyed collections** and more complex entities.

An **Object** is an entity that have a **state** and **behaviour** (properties and method).

```javascript
const person = {
  // Properties
  name: ['Bob', 'Smith'],
  age: 32,
  gender: 'male',
  // Methods
  bio: function() {
    console.log(this.name[0] + ' ' + this.name[1]);
  }
};
```

```javascript
console> window.location.assign('http://google.com') //check what the result will be
```
Objects

Object properties can be accessed in two ways:

**Dot notation**
- `person.age`

**Bracket notation**
- `person[‘gender’]`

```javascript
const person = {
  // Properties
  name: ['Bob', 'Smith'],
  age: 32,
  gender: 'male',
  // Methods
  bio: function() {
    console.log(this.name[0] + ' ' + this.name[1]);
  }
};

person.bio();
console.log(person.age);
console.log(person[‘gender’]);
```
Objects

**Object properties** can be accessed in two ways:

**Dot notation**

`person.age`

**Bracket notation**

`person[‘gender’]`

Object properties can be set using the same method:

`person.age = 34;`

```javascript
const person = {
    // Properties
    name: ['Bob', 'Smith'],
    age: 32,
    gender: 'male',
    // Methods
    bio: function() {
        console.log(this.name[0] + ' ' + this.name[1]);
    }
};
person.age = 34;
console.log(person.age); // 34
```
The **this** keyword refers to the current object.

```javascript
const person = {

  // Properties
  name: ['Bob', 'Smith'],
  age: 32,
  gender: 'male',

  // Methods
  bio: function() {
    console.log(this.name[0] + ' ' + this.name[1]);
  }

};

person.age = 34;
console.log(person.age); // 34

console> console.log(this); // check what the result will be
```
Create a new object

A new object can be created using the “new” keyword.

A new property/method for an object can be set using the dot notation.

A new object can be also created using the create method, which use an existing object as the prototype of the newly object.

A new object can be also created using the assign method, which copies all enumerable own properties from one or more source objects.

```javascript
// create an object
let human = new Object({
  name: 'John',
  age: 21,
});

// Set a new property for an object
human.gender = 'male';

// Set a new method for an object
human.sayMyName = function() {
  return this.name;
};

let newHuman1 = Object.create(human);
let newHuman2 = Object.assign(human);

console.log(human);
console.log(newHuman1);
console.log(newHuman2);
```
Sometimes we need a "blueprint" for creating many objects of the same "type".

The way to create an "object type", is to use an object constructor function.

**Note:** any change made to the constructor function will apply to all objects created based on it.

```javascript
// constructor function
function Person(first, last, age, salary) {
    this.name = {first : first, last : last};
    this.age = age;
    this.salary = salary;
}
const bob = new Person('Bob','Smith', 32, 2300);
console.log(bob);
console.log(bob.bio());
```
Since ES2015 (ES6), there is a new syntax for constructor functions.

//Select the function name and it will suggest converting its style to ES2015

```javascript
function Person(first, last, age, salary)
{
    this.name = {first : first,
                 last : last};
    this.age = age;
    this.salary = salary;
}
const bob = new Person('Bob','Smith', 32, 2300);
console.log(bob);
console.log(bob.bio());
```
Since **ES2015** (**ES6**), there is a new syntax for constructor functions. In **ES6** we can create **classes**.

The **class** basically creates a **template** that we can use to create **instances** of the **class**.

The **constructor()** method is a special method called when an instance of a class is created, i.e., it works the same as the **constructor** function defined before **ES6**.

```javascript
//New syntax for constructor function
Class Person {
constructor(first, last, age, salary)
{
    this.name = {first : first,
                last : last};
    this.age = age;
    this.salary = salary;
}
}

const bob = new Person('Bob','Smith', 32, 2300);
culo.log(bob);
culo.log(bob.bio());
```
The **static** keyword defines a **static property** or **method** for a **class**.

**Static properties** and **static methods** can be called on the class itself, and cannot be called on instances of the class.

**Note:** removing the **static** keyword will allow “static” properties and “static” methods to be called on instances of the class.

```javascript
Class Person {
  // Static property
  static count = 0;
  constructor(first, last, age, salary) {
    ...
    Person.count++;
  }
  // Static method
  static countInstances() { // no function KW
    return `${this.count} ins. created`;
  }
}
const bob = new Person(...);
console.log(Person.countInstances()); // 1 ins. created
```
Class Inheritance

Class inheritance is a way for one class to extend another class, i.e., we can create new functionality on top of the existing.

The `super()` method refers to the parent class.

By calling the `super()` method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.

Note: static properties and methods are inherited as well.

```javascript
Class Person {
    ...
}

Class SuperHero extends Person{
    constructor(first, last, superpower) {
        super{first, last});
        this.sp = superpower;
    }
}

const batman= new SuperHero ('Bruce', 'Wayne', 'rich');
console.log(batman);
```
Generator Functions
Generators

Generator functions provide a powerful alternative to iterators. Generator allow you to define a single function whose execution is not continuous.

Generators are written using the function* syntax.

```javascript
function* generateNumbers() {
    yield 1;
    console.log("After yield 1");
    yield 2;
    yield 3;
    yield 4;
}
```

A simple guide to JavaScript Iterators: https://www.javascripttutorial.net/es6/javascript-iterator/
Generators

When **called**, generator functions do not execute any of their code, instead returning a type of iterator called a Generator.
Generators

When calling the generator's `next` method again, the Generator function executes until it encounters the `yield` keyword.
Generators

When calling the generator's `next` method again, the Generator function executes until it encounters the `yield` keyword.
Generators

When calling the generator's `next` method again, the Generator function executes until it encounters the `yield` keyword.
Generators

When calling the generator's `next` method again, the Generator function executes until it encounters the `yield` keyword.
Generators

When calling the generator's next method again, the Generator function executes until it encounters the yield keyword. It will stop when the done: true.
Regular expressions (RegExp)
Regular expressions are patterns used to match character combinations in strings.

// valid email address
name.surename@gmail.com
name.surename@ut.ee // if only ut emails?

// valid password
8-16 characters
contain lowercase and uppercase
Contain “_”
must not include real name or surname
...

// accepted url?
https://courses.cs.ut.ee/2022/WAD/fall
https://.courses.cs.ut.ee/2022/WAD/fall
https://courses.cs.ut.ee/2022/WAD/fall/
...
A Regular Expression can be created by two ways:

1. Regular Expression Constructor;
2. Regular Expression Literal.

```javascript
// Regular Expression Constructor:
// Syntax: new RegExp(pattern[, flags])
var regexConst = new RegExp('abc');

// Regular Expression Literal:
// Syntax: /pattern/flags
var regexLiteral = /abc/;

/* Flags
   i - Case-insensitive search   // /abc/i
   g - Global search, don’t return after the first match   // /abc/g
*/
```

Regular expressions

To test how Regular expressions works, we can use two method:

The **match()** method retrieves the result of matching a string against a regular expression.

The **test()** method searches a string for a pattern, and returns **true** or **false**, depending on the result.

```javascript
// Regular Expression Constructor:
// Syntax: new RegExp(pattern[, flags])
var regexConst = new RegExp('abc');

let str = "Welcome to the WAD course - the wad team";

let regex = /wad/;
console.log(str.match(regex)); //wad, index: 32

let regex = /wad/i;
console.log(str.match(regex)); //WAD, index: 15

let regex = /wad/ig;
console.log(str.match(regex)); //['WAD', 'wad']
```
 Regular expressions - Ranges

// (X|Y) any of the alternatives separated with "|", X, Or Y
// [0-9] any number from 0 to 9
// [^0-9] any character that is not 0 To 9
// [a-z] any lowercase character from a to z
// [^a-z] any character that is not a lowercase character from A to Z
// [A-Z] any uppercase character from a to z
// [^A-Z] any character that is not an uppercase character from A to Z
// [abc] Find any of the characters between the brackets
// [^abc] Find any characters that is not between the brackets

let str2 = "Web2022App";
let regex2 = /beW[0-2]0[0-2][0-2][A-Z][a-z][a-z]/;
console.log(regex2.test(str2)); // true
Regular exp. - Character classes

// .  matches any character, except newline or line terminators.
// \w  matches word characters. [a-z, A-Z, 0-9, and Underscore]
// \W  matches Non word characters
// \d  matches digits from 0 to 9.
// \D  matches non-digit characters.
// \s  matches whitespace character.
// \S  matches non whitespace character.
// \b  matches at the beginning or end of a word.
// \B  matches NOT at the beginning/end of a word.

let str3 = "Web 2022 App";
let regex3 = /.\w\w\s\d\d\d\d\d\d\s[A-Z][a-z]\S/;
console.log(regex3.test(str3)); // true

let regex4 = /\bWeb\s\d\d\d\d\d\d\s\bApp/;
console.log(regex4.test(str3)); // true
// n+ any string that contains at least one n
// n* any string that contains zero or more occurrences of n
// n? any string that contains zero or one occurrences of n

// n{x} a string containing x numbers of n
// n{x,y} a string containing a range from x to y numbers of n
// n{x,} a string containing at least x numbers of n

let str3 = "Web 2022 App";

let regex5 = /\w+\s\d+\s\w+/;
console.log(regex5.test(str3)); // true

let regex6 = /\w{3}\s\d{4}\s\w{3}/;
console.log(regex6.test(str3)); // true
Regular expressions - Quantifiers

// ^    a string starting with something
// $    a string ending with something
// ?=   a string followed by some "string"
// ?!   a string not followed by some "string"

let str3 = "Web 2022 App";

let regex7 = /^Web\s\d{4}\s\w{2}p$/;
console.log(regex7.test(str3)); // true

let regex8 = /^Web\s\d{4}\s\w{2}(?=p)/;
console.log(regex8.test(str3)); // true
Regular expressions - Quantifiers

// ^  a string starting with something
// $  a string ending with something
// ?=  a string followed by some “string”
// ??  a string not followed by some “string”

let str3 = "Web 2022 App";

let regex7 = /^Web\s\d{4}\s\w{2}p$/;
console.log(regex7.test(str3)); // true

let regex8 = /^Web\s\d{4}\s\w{2}(?=p)/;
console.log(regex8.test(str3)); // true

Regular Expression websites
https://regexr.com/
https://www.regextester.com/
Extra reading/exercises

- **W3Schools** is a free educational website for learning to code online. With their "Try it Yourself" editor, you can edit the HTML code and view the result immediately: https://www.w3schools.com/js/

- **Edabit** (https://edabit.com/challenges) is a coding platform with over 10,000 interactive coding challenges ranging from Very Easy to Expert.

- **LeetCode** (https://edabit.com/challenges) is also a coding platform with interactive coding challenges, but its challenges are more complex than Edabit.
Thank You for your attention

Mohamad Gharib
mohamad.gharib@ut.ee