

# JS Syntax Fundamentals

Syntax, Conditional Statements, Loops, Data  
Type and Variables, Array



SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.bg>

# Table of Contents

1. JavaScript Syntax
2. Data Types and Variables
3. Conditional Statements
4. Loops
5. Undefined and Null
6. Debugging Techniques



Have a Question?



sli.do

#js-front-end



# JavaScript Overview

Definition, Execution, IDE Setup

# What is JavaScript?

- JavaScript (**JS**) is a **high-level** programming language
  - One of the **core technologies** of the World Wide Web
  - Enables **interactive** web pages and applications
  - Can be **executed** on the **server** and on the **client**
- Features:
  - C-like **syntax** (curly-brackets, identifiers, operator)
  - **Multi-paradigm** (imperative, functional, OOP)
  - Dynamic **typing**

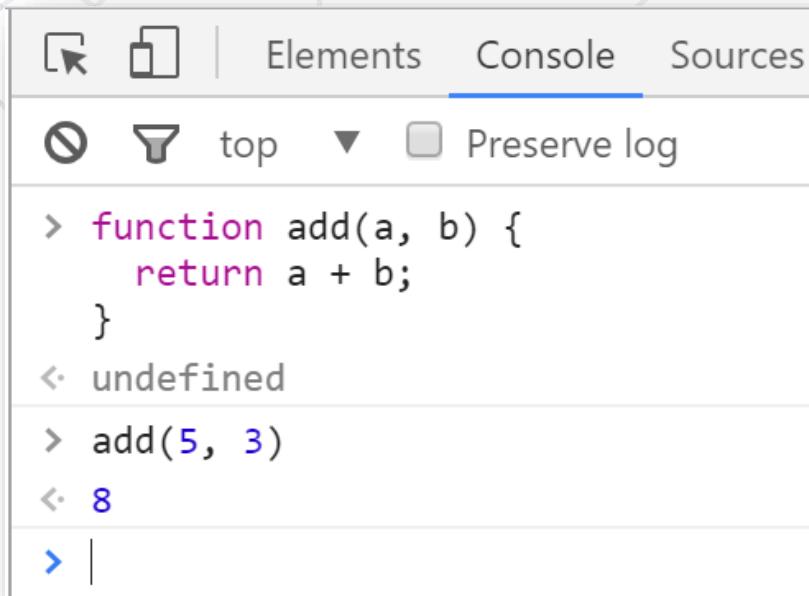


# Dynamic Programming Language

- JavaScript is a **dynamic programming language**
  - Operations otherwise done at **compile-time** can be done at **run-time**
- It is **possible** to change the **type** of a variable or add new properties or methods to an object **while** the program is **running**
- In **static programming languages**, such changes are normally **not possible**

# Chrome Web Browser

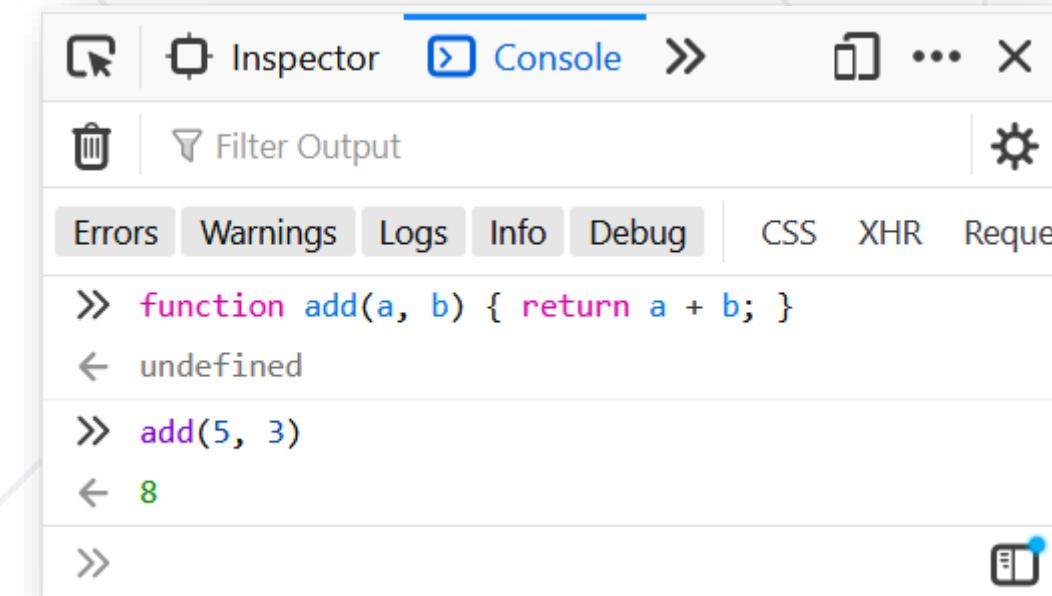
Developer Console: [F12]



Elements    Console    Sources

Preserve log

```
> function add(a, b) {  
    return a + b;  
}  
< undefined  
> add(5, 3)  
< 8  
>
```



Inspector    Console    CSS    XHR    Requests

Filter Output

Errors    Warnings    Logs    Info    Debug

```
>> function add(a, b) { return a + b; }  
< undefined  
>> add(5, 3)  
< 8  
>>
```

# Node.js

- What is **Node.js**?
  - Server-side JavaScript runtime
  - Chrome V8 JavaScript engine
  - NPM **package manager**
  - Install node packages



```
Command Prompt - node
>node
> let a = 5;
undefined
> console.log(a);
5
undefined
>
```

# Install the Latest Node.js

## Downloads

Latest LTS Version: 14.15.4 (includes npm 6.14.10)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS Recommended For Most Users	Current Latest Features
 <a href="#">Windows Installer</a> node-v14.15.4-x64.msi	 <a href="#">macOS Installer</a> node-v14.15.4.pkg
	 <a href="#">Source Code</a> node-v14.15.4.tar.gz

[Windows Installer \(.msi\)](#)

[Windows Binary \(.zip\)](#)

[macOS Installer \(.pkg\)](#)

[macOS Binary \(.tar.gz\)](#)

[Linux Binaries \(x64\)](#)

[Linux Binaries \(ARM\)](#)

[Source Code](#)

32-bit	64-bit
<b>32-bit</b>	64-bit
	64-bit
	64-bit
	64-bit
ARMv7	ARMv8
	node-v14.15.4.tar.gz

# Using Visual Studio Code

- **Visual Studio Code** is powerful text editor for JavaScript and other projects
- In order to create your **first project**:





# JavaScript Syntax

Functions, Operators, Input and Output

# JavaScript Syntax

- C-like **syntax** (curly-brackets, identifiers, operator)
- Defining and Initializing variables:



Declare a variable with let

```
let a = 5;  
let b = 10;
```

Variable name

Variable value

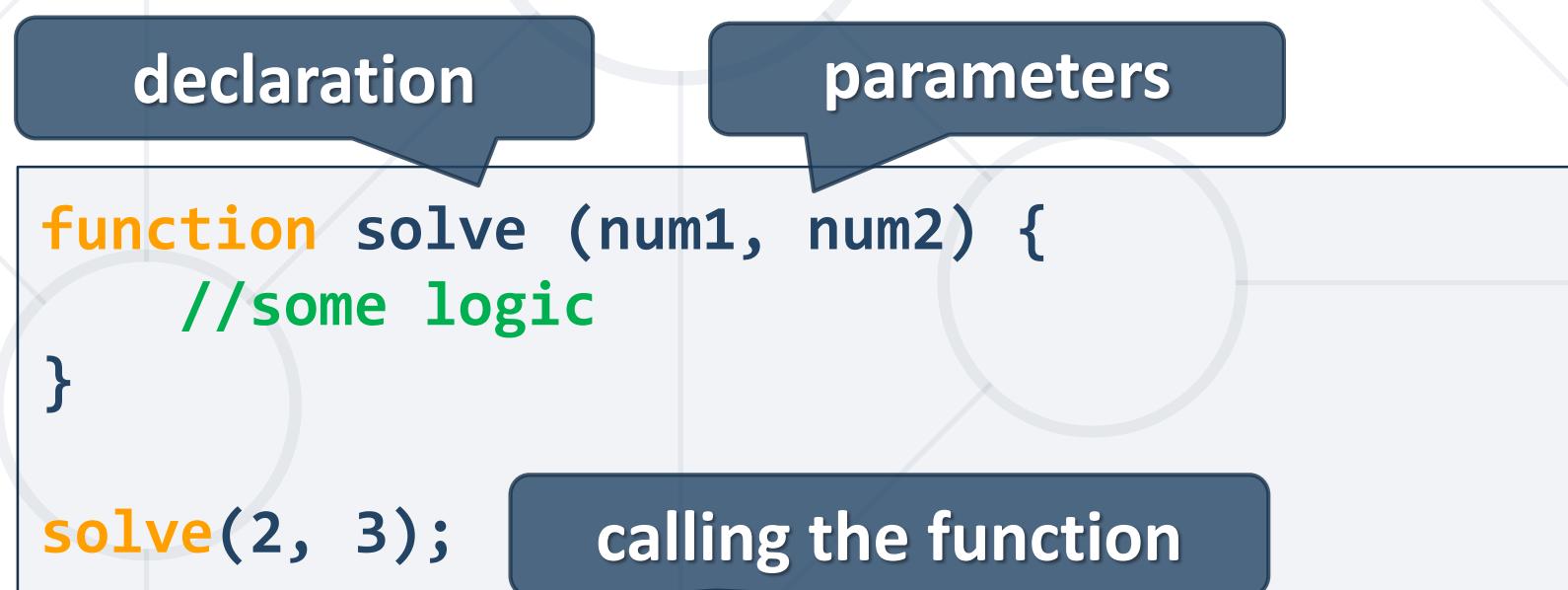
- Conditional statement:

```
if (b > a) {  
    console.log(b);  
}
```

Body of the conditional statement

# Functions and Input Parameters

- In order to solve different problems, we are going to use **functions** and the **input** will come as **parameters**
- A function is similar to a **procedure**, that executes when called



# Printing to the Console

- We use the **console.log()** method to print to console:

```
function solve (name, grade) {  
    console.log('The name is: ' + name + ', grade: ' + grade);  
}  
solve('Peter', 3.555);  
//The name is: Peter, grade: 3.555
```

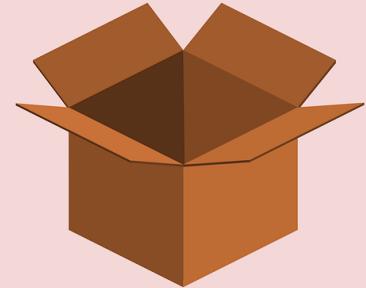
- Text can be composed easier using interpolated strings:

```
console.log(`The name is: ${name}, grade: ${grade}`);
```

- To format a number, use the **toFixed()** method (converts to **string**):

Number of decimal places

```
grade.toFixed(2); //The name is: Petar, grade: 3.56
```



# Data Types and Variables

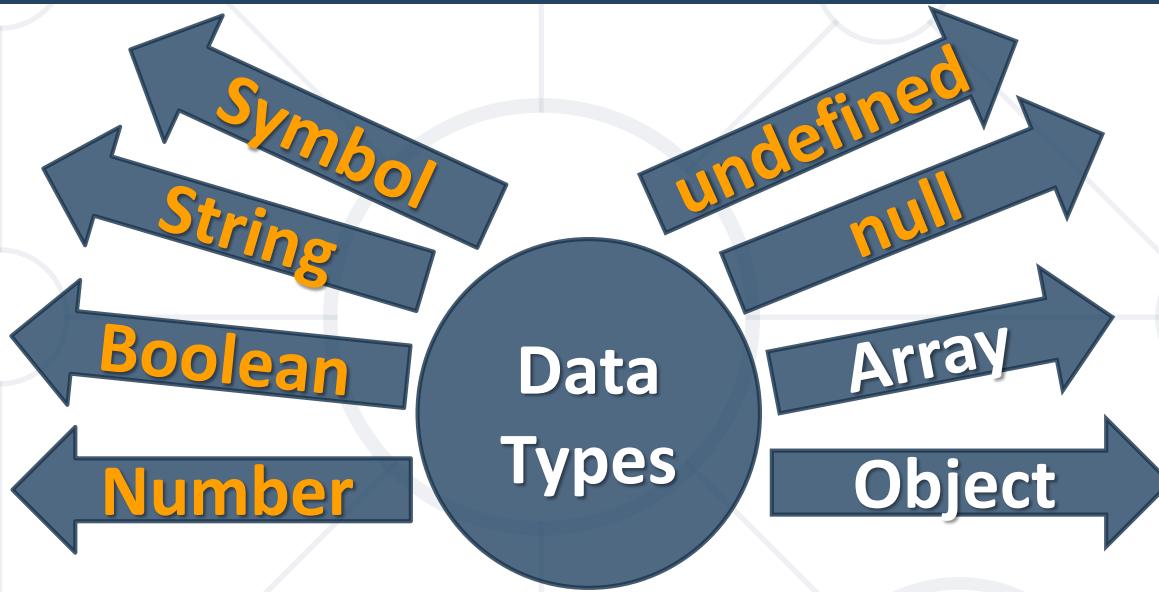
Definition and Examples

# What is a Data Type?

- A **data type** is a classification that specifies what type of operations can be applied to it and the way values of that type are stored
- After **ECMAScript** 2015 there are **seven primitive** data types:
  - Seven **primitive**: Boolean, null, undefined, Number, String, Symbol, BigInt
  - and **Objects** (including Functions and Arrays)



# Examples



```
let number = 10;           // Number
let person = {name: 'George', age: 25}; // Object
let array = [1, 2, 3];      // Array
let isTrue = true;          // Boolean
let name = 'George';       // String
let empty = null;           // null
let unknown = undefined;    // undefined
```

# Variable Scope

- **var** – use **function scope** – can be accessed anywhere in the function, including outside the initial block

```
{  
  var x = 2;  
}  
console.log(x); // 2
```

- **let** and **const** – use **block scope** – when declared inside a block {} can **NOT** be accessed from outside the block

```
{  
  let x = 2;  
}  
console.log(x); // Error
```

# Variable Comparison let vs const

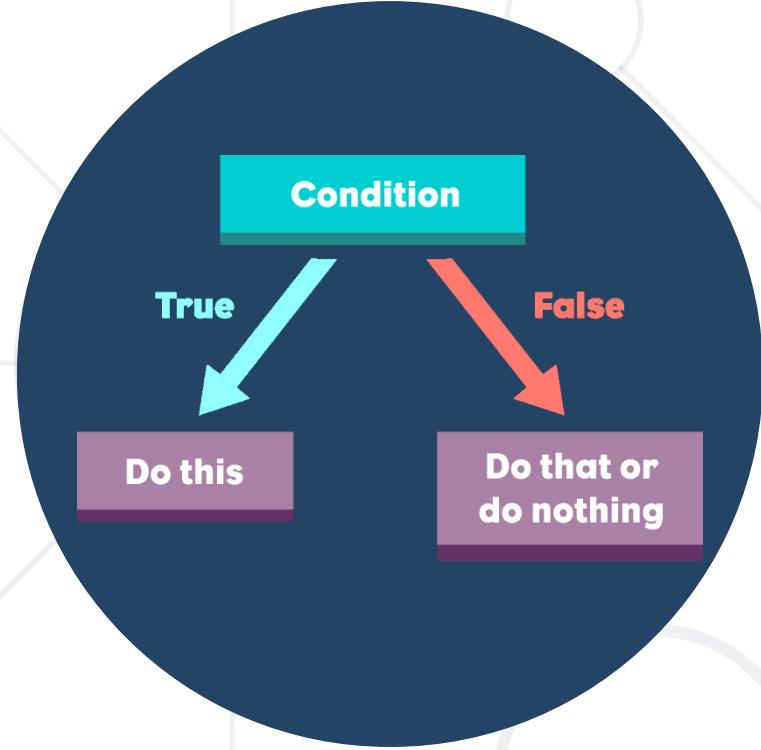
## ■ **let**

- Can be reassigned after initial assignment
- Variable's value can change
- **let** is used when reassignment is necessary

## ■ **const**

- Cannot be reassigned after initial assignment, remains constant
- Variable's value remains fixed
- **const** is used when variable will not be reassigned





# Conditional Statements

## Implementing Control-Flow Logic

# Arithmetic Operators

- **Arithmetic operators** - take numerical values (either literals or variables) as their operands
  - Return a single numerical value
    - Addition (+)
    - Subtraction (-)
    - Multiplication (\*)
    - Division (/)
    - Remainder (%)
    - Exponentiation (\*\*)



```
let a = 15;  
let b = 5;  
let c;  
c = a + b; // 20  
c = a - b; // 10  
c = a * b; // 75  
c = a / b; // 3  
c = a % b; // 0  
c = a ** b; // 155  
= 759375c
```

# Comparison Operators

```
console.log(1 == '1'); // true
console.log(1 === '1'); // false
console.log(3 != '3'); // false
console.log(3 !== '3'); // true
console.log(5 < 5.5); // true
console.log(5 <= 4); // false
console.log(2 > 1.5); // true
console.log(2 >= 2); // true
console.log((5 > 7) ? 4 : 10); // 10
```



Ternary operator

# What is a Conditional Statement?

The **if-else** statement:

- Do action depending on condition

```
let a = 5;  
if (a >= 5) {  
    console.log(a);  
}
```

If the condition **is met**,  
the code will execute

- You can chain conditions

```
else {  
    console.log('no');  
}
```

Continue on the **next**  
**condition**, if the first is **not met**



# Chained Conditional Statements

- The **if / else - if / else...** construct is a series of checks

```
let a = 5;  
if (a > 10)  
    console.log("Bigger than 10");  
else if (a < 10)  
    console.log("Less than 10");  
else  
    console.log("Equal to 10");
```

Only "Less than 10" will be printed

- If one condition is true, it does not proceed to verify the following conditions

# The Switch-case Statement

- Works as a series of **if / else if / else if...**

List of conditions  
(values) for the  
inspection

```
switch (...){  
    case ...:  
        // code  
        break;  
    case ...:  
        // code  
        break;  
    default:  
        // code  
        break;  
}
```

The condition in  
the **switch case** is  
a value

Code to be executed if  
there is no match with any  
case

# Logical Operators

- **Logical operators** are used to determine the logic between variables or values. They return the value of one of the operands based on certain rules, not always just (**true** or **false**).

Operator	Description	Example
!	NOT	<code>!false -&gt; true</code>
<code>&amp;&amp;</code>	AND	<code>true &amp;&amp; false -&gt; false</code>
<code>  </code>	OR	<code>true    false -&gt; true</code>

# Logical Operators: Examples

- Logical "AND"

- Checks the fulfillment of several conditions simultaneously

```
let a = 3;  
let b = -2;  
console.log(a > 0 && b > 0); // expected output: false
```

- Logical "OR"

- Checks that at least one of several conditions is met

```
let a = 3;  
let b = -2;  
console.log(a > 0 || b > 0); // expected output: true
```

# Logical Operators: Examples (2)

- Logical "NOT"
  - Checks if a condition is **not** met

```
let a = 3;  
let b = -2;  
console.log(!(a > 0 || b > 0));  
// expected output: false
```

# Typeof Operator

- The **typeof** operator returns a string indicating the type of an operand



```
const val = 5;
console.log(typeof val);      // number
```

```
const str = 'hello';
console.log(typeof str);      // string
```

```
const obj = {name: 'Maria', age:18};
console.log(typeof obj);      // object
```

# Truthy and Falsy Values

- "truthy" - a value that **coerces to true** when **evaluated** in a boolean context
- The following values are "falsy" - **false, null, undefined, NaN, 0, On** and **""**

```
function logTruthiness (val) {  
  if (val) {  
    console.log("Truthy!");  
  } else {  
    console.log("Falsy.");  
  }  
}
```

logTruthiness (3.14);	//Truthy!
logTruthiness ({});	//Truthy!
logTruthiness (NaN);	//Falsy.
logTruthiness ("NaN");	//Truthy!
logTruthiness ([]);	//Truthy!
logTruthiness (null);	//Falsy.
logTruthiness ("");	//Falsy.
logTruthiness (undefined);	//Falsy.
logTruthiness (0);	//Falsy.



**Loops**

**Code Block Repetition**

# What is a Loop?

## The **for** loop:

- Repeats until the condition is evaluated

```
for (let i = 1; i <= 5; i++){  
    console.log(i)  
}
```

Incrementation **in**  
the condition

## The **while** loop:

- Does the same, but has different structure

```
let i = 1  
while (i <= 5) {  
    console.log(i)  
    i++  
}
```

Incrementation  
**outside** the  
condition





**Undefined and Null**

Non-Existent and Empty

# Undefined

- A variable without a value has the value **undefined**.  
The **typeof** is also **undefined**



```
let car; // Value is undefined, type is undefined
```

- A variable can be emptied, by setting the value to **undefined**. The type will also be **undefined**

```
let car = undefined;  
// Value is undefined, type is undefined
```

# Null

- Null is "nothing". It is supposed to be something that doesn't exist.
- The `typeof` null is an object

```
let person = {  
    firstName:"John",  
    lastName:"Doe",  
    age:50  
};  
person = null;  
console.log(person);          // null  
console.log(typeof(person)); // object
```



# Null and Undefined

- **Null** is an assigned value. It means nothing
- **Undefined** typically means a variable has been declared but not defined yet
- **Null** and **Undefined** are **falsy** values
- **Undefined** and **Null** are equal in value but different in type:

```
null !== undefined // true  
null == undefined // true
```



# Debugging Techniques

Strict Mode, IDE Debugging Tools

# Strict Mode

- **Strict mode** limits certain "sloppy" language features
  - Silent errors will **throw Exception** instead



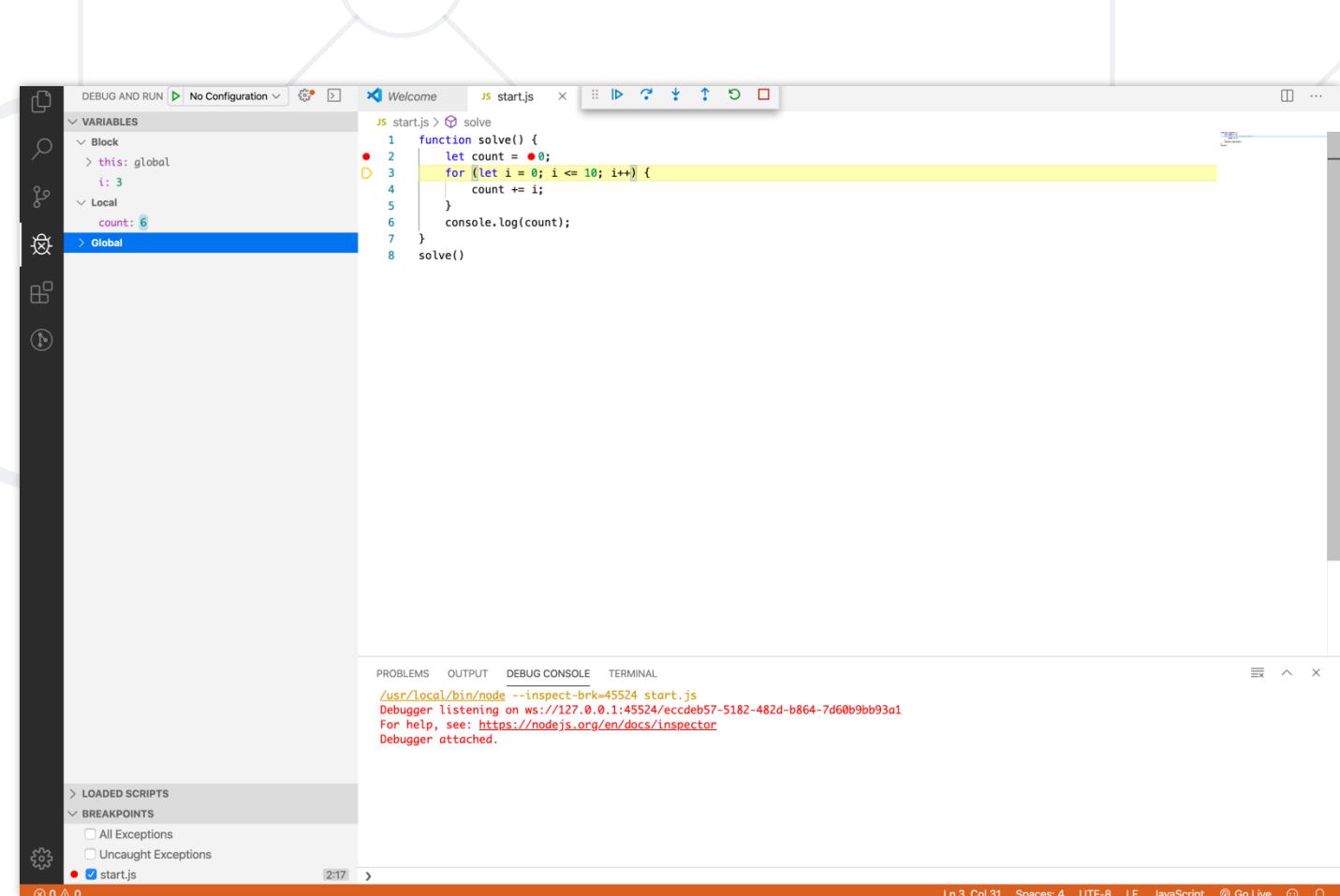
```
'use strict';           // File-Level
mistypeVariable = 17; // ReferenceError
```

```
function strict() {
  'use strict';           // Function-Level
  mistypeVariable = 17;
}
```

- Enabled by default in **modules**

# Debugging in Visual Studio Code

- Visual Studio Code has a built-in **debugger**
- It provides:
  - **Breakpoints**
  - Ability to **trace** the code execution
  - Ability to **inspect** variables at runtime



```
JS start.js > solve
1 function solve() {
2     let count = 0;
3     for (let i = 0; i <= 10; i++) {
4         count += i;
5     }
6     console.log(count);
7 }
8 solve()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

/usr/local/bin/node --inspect-brk=45524 start.js  
Debugger listening on ws://127.0.0.1:45524/eccde857-5182-482d-b864-7d60b9bb93a1  
For help, see: <https://nodejs.org/en/docs/inspector>  
Debugger attached.

LOADED SCRIPTS  
BREAKPOINTS  
All Exceptions  
Uncaught Exceptions  
start.js

Ln 3, Col 31 Spaces: 4 UTF-8 LF JavaScript Go Live

# Using the Debugger in Visual Studio Code

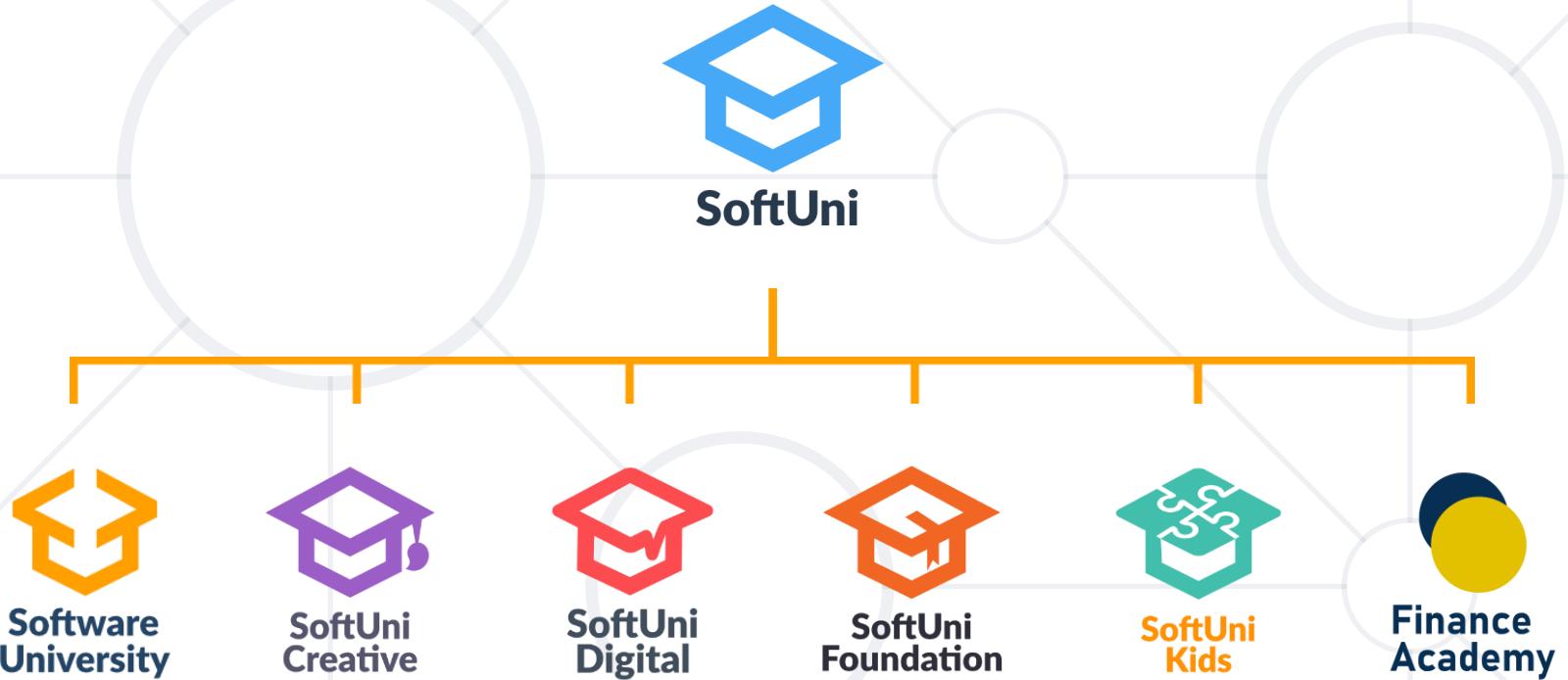
- Start without Debugger: **[Ctrl+F5]**
- Start with Debugger: **[F5]**
- Toggle a breakpoint: **[F9]**
- Trace step by step: **[F10]**
- Force step into: **[F11]**

# Summary

- JS is a **high-level** programming language
- Conditional statement – **If-else, Switch-case**
- Loops – **For-loop, While-loop**
- Data Types
  - **String, Number, Boolean, Null, Undefined**



# Questions?



# SoftUni Diamond Partners



**SUPER  
HOSTING  
.BG**

**INDEAVR**  
Serving the high achievers

 **SOFTWARE  
GROUP**

 **PHAR  
VISION**



**Coca-Cola HBC  
Bulgaria**

 **AMBITIONED**



**BOSCH**

 **SmartIT**

 **DXC  
TECHNOLOGY**

 **Flutter  
International**™

 **DRAFT  
KINGS**

 **Postbank**  
*Решения за твоето утре*

 **createX**

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



# JS Arrays and Strings

## Arrays and Strings



SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.bg>

# Table of Contents

1. Arrays
2. Arrays Methods
3. Text Processing
4. Regular Expressions

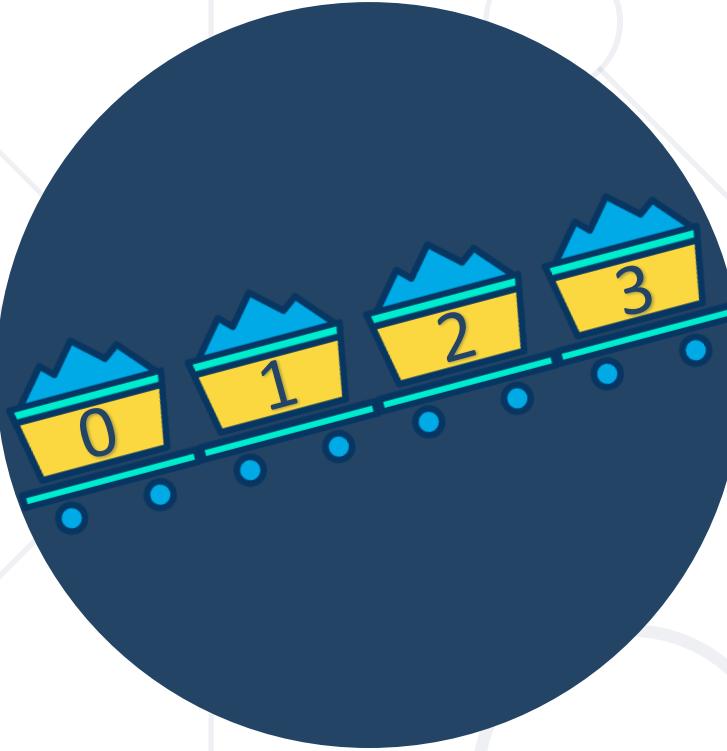


Have a Question?



sli.do

#js-front-end



# Working with Arrays of Elements

Arrays in JavaScript

# What is an Array?

- Arrays are **list-like objects**
- Arrays are a **reference type**, the variable points to an address in memory
- Elements are **numbered** from **0** to **length - 1**
- Creating an array using **an array literal**



```
let numbers = [10, 20, 30, 40, 50];
```



# What is an Array?

- Neither the **length** of a JavaScript array **nor** the **types** of its elements are **fixed**
- An array's **length can be changed** at any time
- Data can be stored at non-contiguous locations in the array
- JavaScript arrays are not guaranteed to be dense



# Arrays of Different Types



```
// Array holding numbers
let numbers = [10, 20, 30, 40, 50];
```

```
// Array holding strings
let weekDays = ['Monday', 'Tuesday', 'Wednesday',
    'Thursday', 'Friday', 'Saturday', 'Sunday'];
```

```
// Array holding mixed data (not a good practice)
let mixedArr = [20, new Date(), 'hello', {x:5, y:8}];
```

# Accessing Elements

- Array elements are accessed using their **index**

```
let cars = ['BMW', 'Audi', 'Opel'];
let firstCar = cars[0];      // BMW
let lastCar = cars[cars.length - 1]; // Opel
```

- Accessing indexes that do not exist in the array returns **undefined**

```
console.log(cars[3]);    // undefined
console.log(cars[-1]);   // undefined
```



# Destructuring Syntax

- Expression that **unpacks values from arrays or objects**, into distinct **variables**

```
let numbers = [10, 20, 30, 40, 50];  
let [a, b, ...elems] = numbers;
```

Rest operator

```
console.log(a) // 10  
console.log(b) // 20  
console.log(elems) // [30, 40, 50]
```

- The **rest operator** can also be used to collect function parameters into an array



# For-of Loop

- Iterates through all **elements** in a collection
- Cannot access the current index



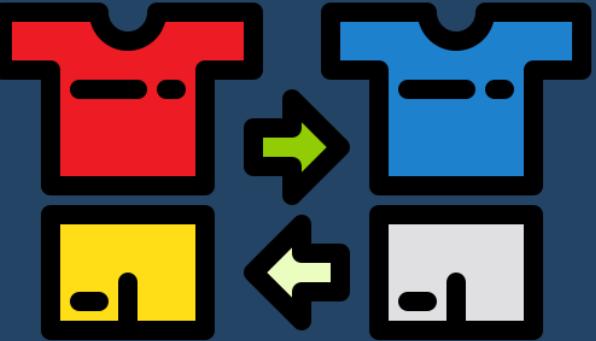
```
for (let el of collection) {  
    // Process the value here  
}
```

# Print an Array with For-of

```
let numbers = [ 1, 2, 3, 4, 5 ];  
  
let output = '';  
  
for (let number of numbers)  
    output += `${number}`;  
  
console.log(output);
```



1 2 3 4 5



# Methods

## Modify the Array

# Pop

- Removes the **last element** from an array and returns that element
- This method **changes** the **length** of the array



```
let nums = [10, 20, 30, 40, 50, 60, 70];
console.log(nums.length); // 7
console.log(nums.pop()); // 70
console.log(nums.length); // 6
console.log(nums); // [ 10, 20, 30, 40, 50, 60 ]
```

# Push

- The **push()** method **adds one or more elements** to the **end** of an array and **returns** the new **length** of the array



```
let nums = [10, 20, 30, 40, 50, 60, 70];
console.log(nums.length); // 7
console.log(nums.push(80)); // 8 (nums.length)
console.log(nums); // [ 10, 20, 30, 40, 50, 60, 70, 80 ]
```

# Shift

- The **shift()** method **removes** the **first element** from an array and **returns** that **removed element**
- This method **changes** the **length** of the array



```
let nums = [10, 20, 30, 40, 50, 60, 70];
console.log(nums.length); // 7
console.log(nums.shift()); // 10 (removed element)
console.log(nums); // [ 20, 30, 40, 50, 60, 70 ]
```

# Unshift

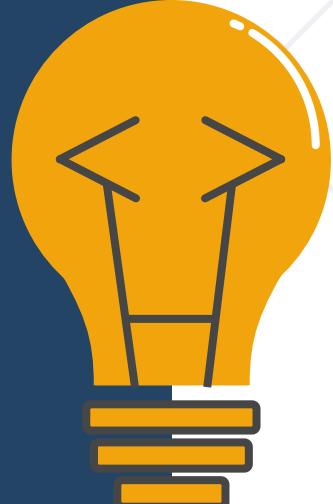
- The **unshift()** method **adds one or more** elements to the **beginning** of an array and **returns** the new **length** of the array



```
let nums = [40, 50, 60];
console.log(nums.length);           // 3
console.log(nums.unshift(30));     // 4 (nums.Length)
console.log(nums.unshift(10,20));   // 6 (nums.Length)
console.log(nums);                // [ 10, 20, 30, 40, 50, 60 ]
```

# Splice

- Changes the contents of an array by **removing** or **replacing** existing **elements** and / or **adding new elements**



```
let nums = [1, 3, 4, 5, 6];
nums.splice(1, 0, 2); // inserts at index 1
console.log(nums); // [ 1, 2, 3, 4, 5, 6 ]
nums.splice(4, 1, 19); // replaces 1 element at index 4
console.log(nums); // [ 1, 2, 3, 4, 19, 6 ]
let el = nums.splice(2, 1); // removes 1 element at index 2
console.log(nums); // [ 1, 2, 4, 19, 6 ]
console.log(el); // [ 3 ]
```

# Reverse

- Reverses the array
  - The **first** array **element becomes** the **last**, and the last array element becomes the first

```
let arr = [1, 2, 3, 4];
arr.reverse();
console.log(arr); // [ 4, 3, 2, 1 ]
```



# Join

- Creates and returns a **new string** by **concatenating** all of the elements in an array (or an array-like object), **separated** by commas or a **specified separator** string

```
let elements = ['Fire', 'Air', 'Water'];
console.log(elements.join()); // "Fire,Air,Water"
console.log(elements.join(' ')); // "Fire Air Water"
console.log(elements.join('-')); // "Fire-Air-Water"
console.log(['Fire'].join(".")); // Fire
```



# Slice

- The **slice()** method **returns** a shallow **copy** of a **portion** of an array into a **new array** object selected from begin to end (end not included)
- The **original array** will **not** be **modified**



```
let fruits = ['Banana', 'Orange', 'Lemon', 'Apple'];

let citrus = fruits.slice(1, 3);

let fruitsCopy = fruits.slice();

// fruits contains ['Banana', 'Orange', 'Lemon', 'Apple']
// citrus contains ['Orange', 'Lemon']
```

# Includes

- Determines whether an array contains a certain element, returning **true** or **false** as appropriate

```
// array length is 3
// fromIndex is -100
// computed index is 3 + (-100) = -97

let arr = ['a', 'b', 'c'];

arr.includes('a', -100); // true
arr.includes('b', -100); // true
arr.includes('c', -100); // true
arr.includes('a', -2); // false
```



# IndexOf

- The **indexOf()** method **returns** the **first index** at which a given **element** can be **found** in the array
  - Output is **-1** if element is **not present**



```
const beasts = ['ant', 'bison', 'camel', 'duck', 'bison'];

console.log(beasts.indexOf('bison')) // 1
// start from index 2
console.log(beasts.indexOf('bison', 2)) // 4
console.log(beasts.indexOf('giraffe')) // -1
```

# ForEach

- The **forEach()** method **executes a provided function** once for each array element
- Converting a for loop to forEach



```
const items = ['item1', 'item2', 'item3'];
const copy = [];

// For Loop
for (let i = 0; i < items.length; i++) {
    copy.push(items[i]);
}

// ForEach
items.forEach(item => { copy.push(item); });
```

# Map

- Creates a new array with the results of calling a provided function on every element in the calling array

```
let numbers = [1, 4, 9];
let roots = numbers.map(function(num, i, arr) {
    return Math.sqrt(num)
});
// roots is now [1, 2, 3]
// numbers is still [1, 4, 9]
```



# Find

- Returns the **first found value** in the array, if an **element** in the array **satisfies the provided testing function** or **undefined** if not found



```
let array1 = [5, 12, 8, 130, 44];
let found = array1.find(function(element) {
  return element > 10;
});
console.log(found); // 12
```

# Filter

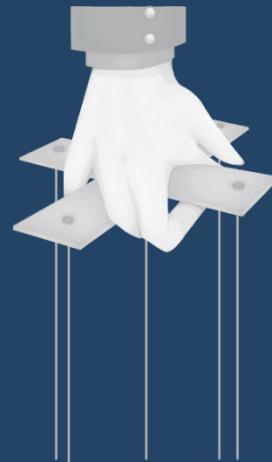
- Creates a **new array** with **filtered elements only**
- Calls a **provided callback function** once for each element in an array
- **Does not mutate** the **array** on which it is called



```
let fruits = ['apple', 'banana', 'grapes', 'mango', 'orange'];
// Filter array items based on search criteria (query)

function filterItems(arr, query) {
  return arr.filter(function(el) {
    return el.toLowerCase().indexOf(query.toLowerCase()) !== -1;
  });
}

console.log(filterItems(fruits, 'ap')) // ['apple', 'grapes']
```



# Manipulating Strings

# Concatenating

- Use the "+" or the "+=" operators

```
let text = "Hello" + ", ";
// Expected output: "Hello, "
text += "JS!"; // "Hello, JS!"
```

- Use the concat() method

```
let greet = "Hello, ";
let name = "John";
let result = greet.concat(name);
console.log(result); // Expected output: "Hello, John"
```

# Searching for Substrings

- **indexOf(substr)**

```
let str = "I am JavaScript developer";
console.log(str.indexOf("Java")); // Expected output: 5
console.log(str.indexOf("java")); // Expected output: -1
```

- **lastIndexOf(substr)**

```
let str = "Intro to programming";
let last = str.lastIndexOf("o");
console.log(last); // Expected output: 11
```

# Extracting Substrings

- **substring(startIndex, endIndex)**

```
let str = "I am JavaScript developer";
let sub = str.substring(5, 10);
console.log(sub); // Expected output: JavaS
```

- **replace(search, replacement)**

```
let text = "Hello, john@softuni.bg, you have been  
using john@softuni.bg in your registration.";  
  
let replacedText = text.replace(".bg", ".com");  
  
console.log(replacedText);  
// Hello, john@softuni.com, you have been using  
john@softuni.bg in your registration.
```

# Problem: Substring

- Receives a **string**, a **start index**, and **count** characters
- Print the **substring** of the received string

"ASentence", 1, 8



Sentence

"JavaScript", 4, 6



Script

# Solution: Substring

```
function solve(text, startIndex, count) {  
    let substring = text  
        .substring(startIndex, startIndex + count);  
  
    console.log(substring);  
}
```

- **split(separator)**

```
let text = "I love fruits";
let words = text.split(' ');
console.log(words); // Expected output: ['I', 'Love', 'fruits']
```

- **includes(substr)**

```
let text = "I love fruits.";
console.log(text.includes("fruits")); // Expected output: True
console.log(text.includes("banana")); // Expected output: False
```

# Repeating Strings

- **repeat(count)** - Creates a new string repeated count times

```
let n = 3;  
  
for(let i = 1; i <= n; i++) {  
    console.log('*'.repeat(i));  
}
```



```
// *  
// **  
// ***
```

# Problem: Censored Words

- Receives a **text** and a **single word**
- Find all **occurrences** of that word in the text and **replace** them with the corresponding amount of '**\***'

A small sentence with some words,  
**small**



A \*\*\*\*\* sentence with some words

# Solution: Censored Words

```
function solve(text, word) {  
    while (text.includes(word)) {  
        text = text.replace(word, '*' .repeat(word.length));  
    }  
    console.log(text);  
}
```

# Trimming Strings

- Use **trim()** method to remove **whitespaces** (spaces, tabs, no-break space, etc. ) from **both ends** of a string

```
let text = "    Annoying spaces      ";
console.log(text.trim()); // Expected output: "Annoying spaces"
```

- Use **trimStart()** or **trimEnd()** to remove whitespaces **only** at the beginning or at the end

```
let text = "    Annoying spaces      ";
text = text.trimStart(); text = text.trimEnd();
console.log(text); // Expected output: "Annoying spaces"
```

# Starts With/Ends with

- Use **startsWith()** to determine whether a string **begins** with the characters of a specified substring

```
let text = "My name is John";  
console.log(text.startsWith('My')); // Expected output: true
```

- Use **endsWith()** to determine whether a string **ends** with the characters of a specified substring

```
let text = "My name is John";  
console.log(text.endsWith('John')); // Expected output: true
```

# Padding at the Start and End

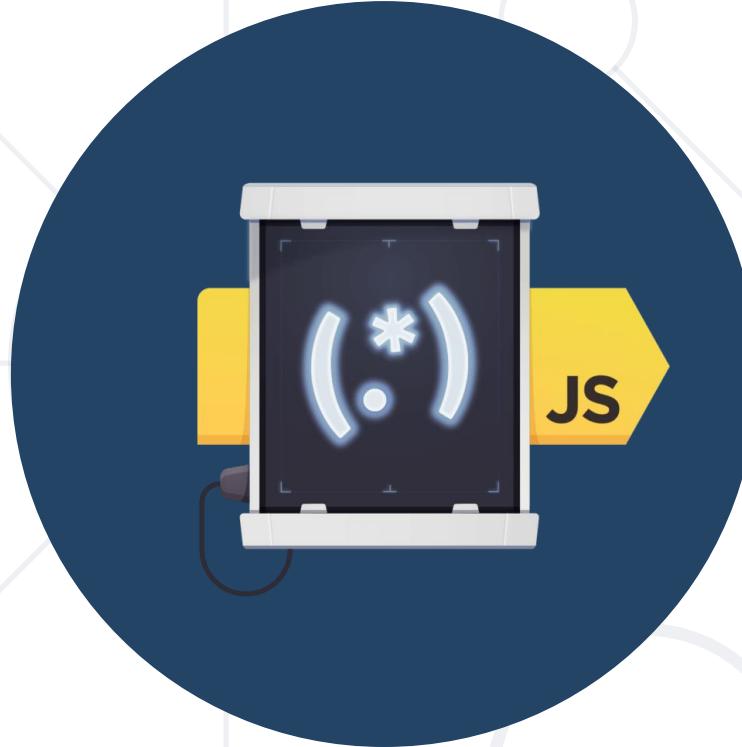
- Use **padStart()** to add to the current string **another substring** at the **start** until a **length** is reached

Receives **length** and **substring**

```
let text = "010";  
  
console.log(text.padStart(8, '0')); // Expected output: 00000010
```

- Use **padEnd()** to add to the current string **another substring** at the **end** until a **length** is reached

```
let sentence = "He passed away";  
  
console.log(sentence.padEnd(20, '.'));  
  
// Expected output: He passed away.....
```



# Regular Expressions in JS

- In JS you construct a regular expression in one of two ways:
  - Regular Expression Literal
  - The constructor function **RegExp**

```
// Provides compilation when the script is Loaded
```

```
let regLiteral = /[A-Za-z]+/g
```

```
// Provides runtime compilation
```

```
// Used when the pattern is from another source
```

```
let regExp = new RegExp('[A-Za-z]+', 'g');
```

# Validating String by Pattern

- The method **test(string)**
  - Determines whether there is a match

```
let text = 'Today is 2015-05-11';
let regexp = /\d{4}-\d{2}-\d{2}/g;

let containsValidDate = regexp.test(text);
console.log(containsValidDate); // true
```

# Checking for Matches

- The method **match(regexp)**
  - Returns an **array** of all matches (strings)

```
let text = 'Peter: 123 Mark: 456';

let regexp = /([A-Z][a-z]+): (\d+)/g;

let matches = text.match(regexp);

console.log(matches.length); // 2
console.log(matches[0]); // Peter: 123
console.log(matches[1]); // Mark: 456
```

# Using the exec() Method

- The method **exec(string, text)**
  - Works with a pointer & returns the **groups**

```
const text = "Peter: 123 Mark: 456";
const regexp = /([A-Z][a-z]+): (\d+)/g;
const firstMatch = regexp.exec(text);

console.log(firstMatch[0]); // Peter: 123
console.log(firstMatch[1]); // Peter
console.log(firstMatch[2]); // 123
```

# Replacing with RegExp

- The method **replace(regexp, stringReplacement)**
  - Replaces all strings that **match the pattern** with the provided replacement

```
const text = 'Peter: 123 Mark: 456';
const replacement = '999';
const regexp = /\d{3}/g;
const result = text.replace(regexp, replacement);
// Peter: 999 Mark: 999
```

- The method **matchAll(regexp)**
  - returns an iterator of all results matching a string against a **regular expression**, including **capturing groups**

```
const regexp = /t(e)(st(\d?))/g;  
const str = 'test1test2';  
const array = [...str.matchAll(regexp)];  
console.log(array[0]);  
// ['test1', 'e', 'st1', '1', index: 0, input:'test1test2', length: 4]
```

# Splitting with RegExp

- The method **split(regexp)**

- Splits the text by the pattern
- Returns an array of strings

```
let text = '1 2 3 4';
let regexp = /\s+/g;
let result = text.split(regexp);
console.log(result) // ['1', '2', '3', '4'];
```

# Problem: Count String Occurrences

- Receive a **text** and a **word** that you need to **search**
- Find the number of **all occurrences** of that word and print it

```
"This is a word and it also is a sentence",  
"is"
```

→ 2

# Solution: Count String Occurrences

```
function solve(text, search) {  
    let words = text.split(' ');  
    let counter = 0;  
    for (let w of words) {  
        if (w === search) {  
            counter++;  
        }  
    }  
    console.log(counter);  
}
```



# Live Exercises

# Summary

- **Array**
  - **Methods**
- **Associative Array**
- **Text Processing**
  - **Regular expressions** describe **patterns** for searching through text.



# Questions?



SoftUni



Software  
University



SoftUni  
Creative



SoftUni  
Digital



SoftUni  
Foundation



SoftUni  
Kids



Finance  
Academy

# SoftUni Diamond Partners



**SUPER  
HOSTING  
.BG**

**INDEAVR**  
Serving the high achievers

 **SOFTWARE  
GROUP**

 **PHAR  
VISION**



**Coca-Cola HBC  
Bulgaria**

 **AMBITIONED**



**BOSCH**

 **SmartIT**

 **DXC  
TECHNOLOGY**

 **Flutter  
International™**

 **DRAFT  
KINGS**

 **Postbank**  
*Решения за твоето утре*

 **createX**

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



# Functions and Statements

$f(x)$

SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.bg>

# Table of Contents

1. Declaring and Invoking Functions
2. Nested Functions
3. Value vs Reference Types
4. First-class Functions
5. Arrow Functions
6. Naming and Best Practices

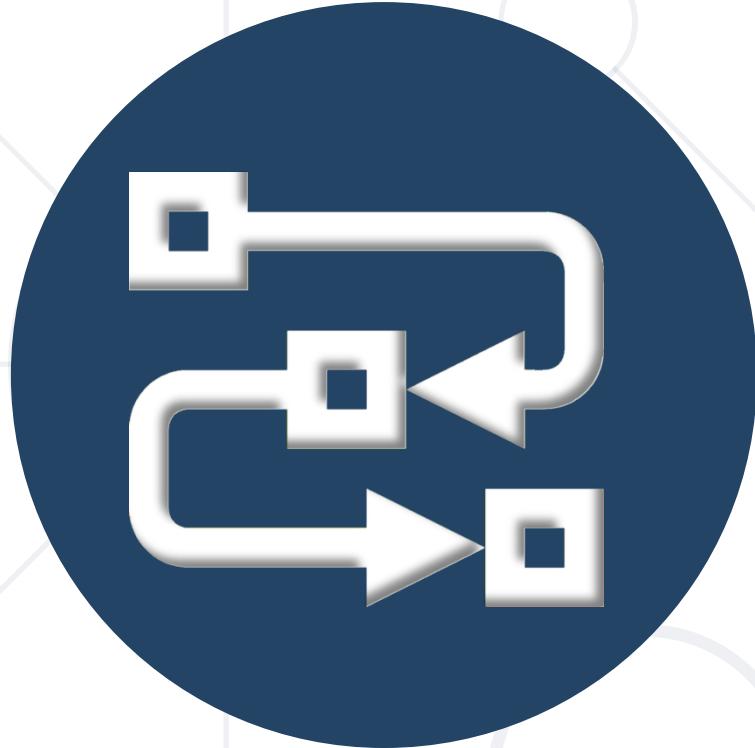


Have a Question?



**sli.do**

**#js-front-end**



# Functions Overview

## Definition and Objectives

# Functions in JS

- A **function** is a **named subprogram** designed to perform a particular task
- Functions are executed when they are called. This is known as **invoking** a function
- Values can be **passed** into functions and used within the function



```
function printStars(count) {  
    console.log("*".repeat(count));  
}
```

Use **camelCase**

Parameter

# Why Use Functions?

- More **manageable programming**
  - **Splits** large problems into small pieces
  - Better **organization** of the program
  - Improves code **readability** and **understandability**
- Avoiding **repeating code**
  - Improves code maintainability
- Code **reusability**
  - Using existing functions several times





# Declaring and Invoking Functions

# Declaring Function

- Functions can be declared in two ways:
  - **Function declaration** (recommended way)

```
function printText(text){  
    console.log(text);  
}
```

- **Function expression** (useful in functional programming)

```
let printText = function(text){  
    console.log(text);  
}
```



# Declaring Function

- Functions can have **parameters**
- Functions **always** return a value (custom or default)



```
function printText(text){  
    console.log(text);  
}
```

The code snippet illustrates the components of a function declaration. It consists of three main parts: 'Name' (the identifier 'printText'), 'Parameters' (the parameter 'text' in parentheses), and 'Body' (the code block 'console.log(text)').

# Invoking a Function

- Functions are first **declared**, then **invoked** (many times)

```
function hLine() {  
    console.log("-----");  
}
```

Function  
Declaration

- Functions can be **invoked (called)** by their name

```
hLine();
```

Function  
Invocation

# Invoking a Function (2)

- Invocation from another function:

```
function printDocument() {  
    printLabel();  
    printContent();  
}
```

Function invoking  
functions

- Self-invocation (**recursion**):

```
function countDown(x) {  
    console.log(x);  
    if (x > 0) { countDown(x - 1); }  
}
```

Function invoking  
itself

# Functions Without Parameters

- Does **not** receive arguments when invoked
- The result is **always the same** (unless it reads data from outside)

```
function printHeader() {  
    console.log('~~~- {@} -~~~');  
    console.log('~- Certificate ~-');  
    console.log('~~~- ----- -~~~');  
}  
printHeader(); // Output is always the same
```

# Functions With Parameters

- Can receive **any number** and **type** of arguments when invoked

```
function multiply(a, b) {  
    console.log(a*b);  
}  
multiply(5, 7); // 35
```

Pass two numbers

```
function printName(nameArr) {  
    console.log(nameArr[0] + ' ' + nameArr[1]);  
}  
printName(['John', 'Smith']); // John Smith
```

Pass array of strings

# Problem : Format Grade

- Write a function that **receives a grade** between 2.00 and 6.00 and prints a formatted line with **grade and description**
  - Grade < 3.00 → **Fail**
  - Grade  $\geq$  3.00 and < 3.50 → **Poor**
  - Grade  $\geq$  3.50 and < 4.50 → **Good**
  - Grade  $\geq$  4.50 and < 5.50 → **Very good**
  - Grade  $\geq$  5.50 → **Excellent**

Input	Output
3.33	Poor (3.33)
4.50	Very good (4.50)
2.99	Fail (2)

# Solution: Format Grade

```
function formatGrade(grade) {  
    if (grade < 3.00) {  
        console.log('Fail (2)');  
    } else if (grade < 3.5) {  
        console.log(`Poor ${grade}`);  
    }  
    // TODO: Add other conditions  
}
```

# Problem : Math Power

- Create a function that **calculates** the result of a number, raised to the given power
  - **Print** the result to the console

Input	Output	Details
2,8	256	$2^8 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 256$
3,4	81	$3^4 = 3 * 3 * 3 * 3 = 81$

# Solution: Math Power

```
function pow(num, power){  
    let result = 1;  
    // Loop exponent times  
    for(let i = 0; i < power; i++){  
        //multiply the base value  
        result *= num;  
    }  
    console.log(result);  
}
```



**Returning Values**

# The Return Statement

- The **return** keyword immediately **stops the function's execution**
- **Returns** the specified value to the caller



```
function readFullName(firstName, lastName) {  
    return firstName + " " + lastName;  
}  
  
const fullName = readFullName("John", "Smith");  
console.log(fullName) //John Smith
```

# Using the Return Values

- Return value can be:

- Assigned to a variable

```
let max = getMax(5, 10);
```

- Used in expression

```
let total = getPrice() * quantity * 1.20;
```

- Passed to another function

```
multiply(getMax(5,10), 20)
```



# Returning Values: Examples

- Check if **array index** is valid:

```
function isValid(index, arr) {  
    if (Number.isInteger(index) && index >= 0 && index < arr.length) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

- Does the student pass the exam:

```
function pass(grade) {  
    return grade >= 3;  
}
```

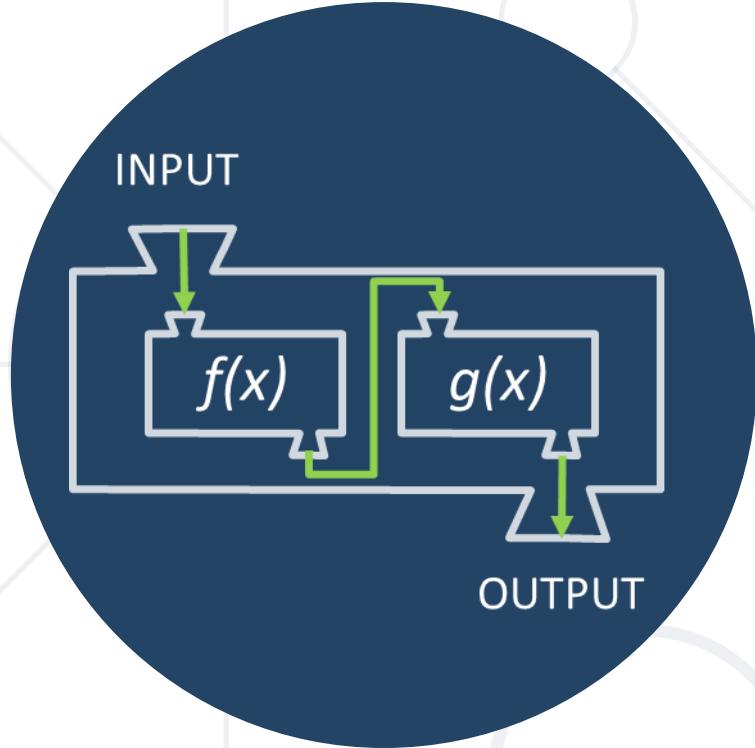
# Problem : Repeat String

- Create a function that takes a **string** and a **number n** and returns the string repeated **n times**
  - **Return** the result as a string

Input	Output
"abc", 3	abcabcabc
"String", 2	StringString

# Solution: Repeat String

```
function repeat(str, n) {  
    let result = '';  
    for (let i = 0; i < n; i++) {  
        result += str;  
    }  
    return result;  
}
```



# Nested Functions

# Nested Functions: Example

- Functions can be **nested**, i.e. hold other functions

```
function swapElements(arr) {  
    for (let i = 0; i < arr.length/2; i++) {  
        swap(arr, i, arr.length - 1 - i);  
    }  
    console.log(arr.join(' '));  
    function swap(elements, i, j) {  
        let temp = elements[i];  
        elements[i] = elements[j];  
        elements[j] = temp;  
    }  
}
```

Nested function

# Problem: Print Certificate

- Write a function that receives a **grade** and an **array**, containing two strings and **prints** a formatted certificate
  - If the student failed, **print "Student does not qualify"**

```
printCertificate(5.25, ['Peter', 'Carter']);
// ~~~- {@} -~~~
// ~- Certificate -~
// ~~~- ~~~- -~~~
// Peter Carter
// Very good (5.25)
```

# Solution: Print Certificate

- Use the functions we declared in **earlier examples**:

```
function printCertificate(grade, nameArr) {  
    if (pass(grade)) {  
        printHeader();  
        printName(nameArr);  
        formatGrade(grade);  
    } else {  
        let msg = `${nameArr[0]} ${nameArr[1]} does not qualify`;  
        console.log(msg);  
    }  
}
```



*f(x)*

# Functional Programming in JS

## First Class

# First-Class Functions

- First-class functions are treated like any other variable
  - Passed as an argument
  - Returned by another function
  - Assigned as a value to a variable



The term "first-class" means that something is just a value. A first-class function is one that can go anywhere that any other value can go - there are few to no restrictions.

*Michael Fogus, Functional Javascript*

# First-Class Functions

- Can be passed as an **argument** to another function



```
function sayHello() {  
    return "Hello, ";  
}
```

```
function greeting(helloMessage, name) {  
    return helloMessage() + name;  
}
```

```
console.log(greeting(sayHello, "JavaScript!"));  
// Hello, JavaScript!
```

# First-Class Functions

- Can be **returned** by another function
  - We can do that, because we treated functions in JavaScript as a **value**



```
function sayHello() {  
    return function () {  
        console.log('Hello!');  
    }  
}
```

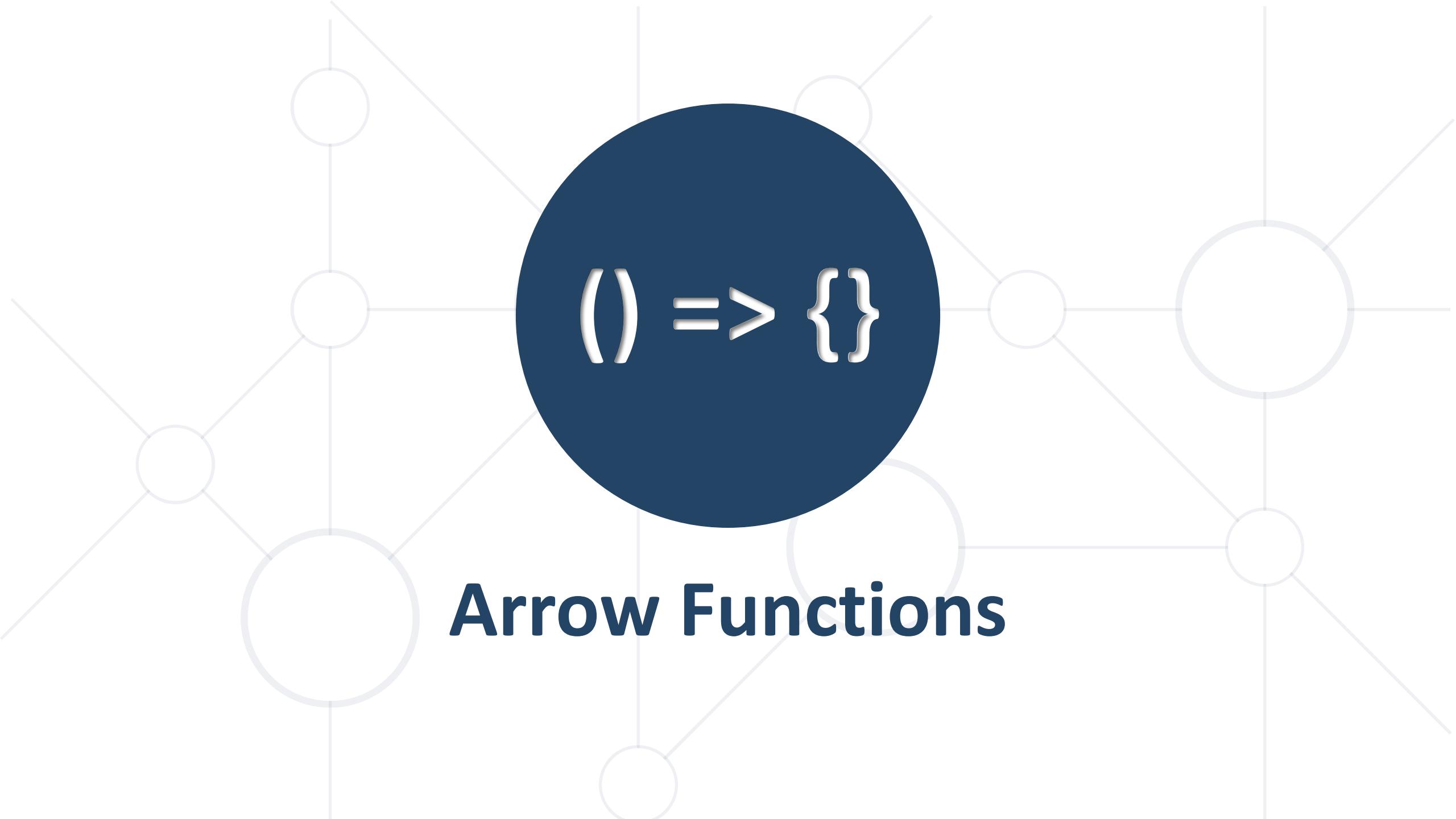
# First-Class Functions

- Can be assigned as a **value** to a **variable**



```
const write = function () {  
    return "Hello, world!";  
}
```

```
console.log(write());  
// Hello, world!
```



`() => {}`

# Arrow Functions

# Arrow Functions

- Special **shorthand syntax** for declaration
- They operate in the **context** of their **enclosing scope**
- Useful in **functional programming**



```
let increment = x => x + 1;  
console.log(increment(5)); // 6
```

"=>" (arrow)

```
let increment = function(x) {  
    return x + 1;  
}
```

This is the same as  
the function above

```
let sum = (a, b) => a + b;  
console.log(sum(5, 6)); // 11
```



# Naming and Best Practices

# Naming Functions



- Use **meaningful** names
  - Should be in **camelCase**
  - Names should answer the question:
    - **What does this function do?**
- `findStudent, loadReport, add`
- `Method1, DoSomething, handleStuff, DirtyHack`
- If you cannot find a good name for a function, think about whether it has a **clear intent**
- Self explaining
- Puzzling

# Naming Function Parameters

- Function parameter names:
  - Preferred form: [Noun] or [Adjective] + [Noun]
  - Should be in **camelCase**
  - Should be **meaningful**  
`firstName, report, speedKmH,  
usersList, fontSizeInPixels, font`
  - Unit of measure should be obvious

`p, p1, p2, populate, LastName, last_name, convertImage`

# Functions – Best Practices

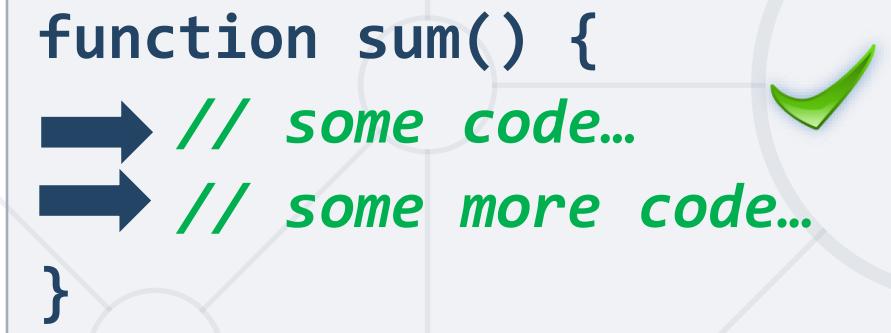
- Each **function** should perform a **single**, well-defined task
  - A name should **describe that task** in a clear and non-ambiguous way
- **Avoid functions longer than one screen**
  - **Split them** into several shorter functions

```
function printReceipt(){  
    printHeader();  
    printBody();  
    printFooter();  
}
```

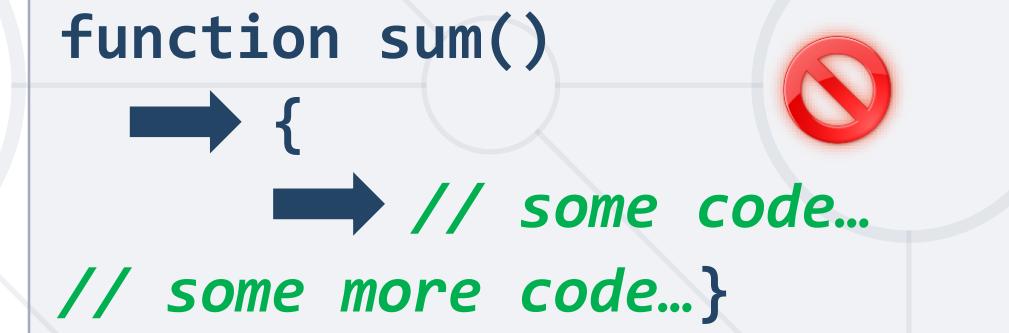
Self documenting  
and easy to test

# Code Structure and Code Formatting

- Make sure to use correct **indentation**



```
function sum() {  
    ➔ // some code...  
    ➔ // some more code...  
}
```

A code block enclosed in a light gray rounded rectangle. It contains a green checkmark icon in the top right corner. Two dark blue arrows point from the text to the first two lines of code, indicating they are part of the same block.

```
function sum()  
    ➔ {  
        ➔ // some code...  
        // some more code... }  
    }
```

A code block enclosed in a light gray rounded rectangle. It contains a large red circle with a diagonal slash (no symbol) in the top right corner. Two dark blue arrows point from the text to the first two lines of code, indicating they are part of the same block.

- Leave a **blank line** between **functions** and after **blocks**
- Always use **curly brackets** for **conditional** and **loop bodies**
- **Avoid long lines** and **complex expressions**

# Problem: Simple Calculator

- Write a function that **receives three parameters** and calculates the result, depending on a given operator
- The operator can be '**multiply**', '**divide**', '**add**', '**subtract**'
- The input comes as three parameters - two **numbers** and an operator as a **string**

Input	Output
5, 10, 'multiply'	50

- **Bonus task:** use **arrow functions** for the solution

# Solution: Simple Calculator

```
function solve(a, b, operator) {  
    switch (operator) {  
        case 'multiply':  
            multiply(a, b);  
            break;  
        //TODO: other cases  
    }  
    function multiply(a, b) { // ...body }  
    //TODO: other operations  
}
```

- **Functions:**
  - Break large programs into simple functions that solve small sub-problems
  - Consist of **declaration** and **body**
  - Are invoked by their **name**
  - Can accept **parameters**



# Questions?



Software  
University



SoftUni  
Creative



SoftUni  
Digital



SoftUni  
Foundation



SoftUni  
Kids



Finance  
Academy

# SoftUni Diamond Partners



**SUPER  
HOSTING  
.BG**

**INDEAVR**  
Serving the high achievers

 **SOFTWARE  
GROUP**

 **PHAR  
VISION**



**Coca-Cola HBC  
Bulgaria**

 **AMBITIONED**



**BOSCH**

 **SmartIT**

 **DXC  
TECHNOLOGY**

 **Flutter  
International**™

 **DRAFT  
KINGS**

 **Postbank**  
*Решения за твоето утре*

 **createX**

# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg)
  - Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University

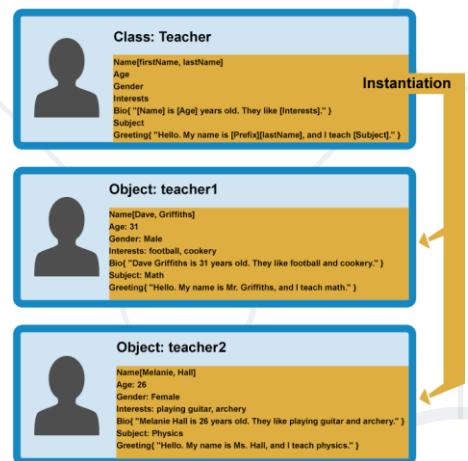


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



# Objects and Classes

Using Objects and Classes  
Defining Simple Classes



SoftUni Team  
Technical Trainers



# SoftUni



Software University  
<https://softuni.bg>

# Table of Contents

1. Objects (definition, properties, and methods)
2. JSON
3. Associative Arrays
4. Classes

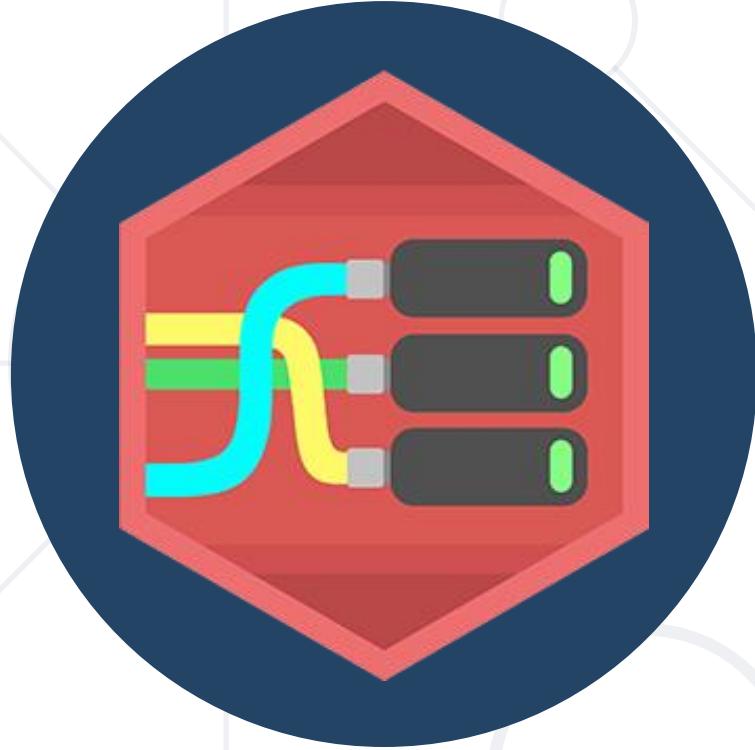


Have a Question?



**sli.do**

**#js-front-end**



# Objects

Definition, Properties and Methods

# What Are Objects ?

- **Structure** of related data or functionality
- Contains **values** accessed by **string keys**
  - Data values are called **properties**
  - Function values are called **methods**



Object	
'name'	'Peter'
'age'	20

Property name (key)

Property value

- You can **add** and **remove** properties **during runtime**

# Object Definition

- We can create an object with an **object literal**

```
let person = { name:'Peter', age:20, height:183 };
```

- We can define an **empty object** and **add properties** later

```
let person = {};
person.name = 'Peter';
person.age = 20;
person.hairColor = 'black';
```

```
person['lastName'] = 'Parker';
```

Access and set  
properties using  
**string indexation**

# Problem: Person Info

- Create an **object** that has a first name, last name, and age
- **Return** the object at the end of your function

```
"Peter",  
"Pan",  
20
```



```
firstName: Peter  
lastName: Pan  
age: 20
```

```
"Jack",  
"Sparrow",  
"unknown"
```



```
firstName: Jack  
lastName: Sparrow  
age: unknown
```

# Solution: Person Info

- Create an object
- Set the properties **firstName**, **lastName**, and **age**
- Return the created object using the **return** keyword

```
function personInfo(firstName, lastName, age) {  
    let person = {};  
    person.firstName = firstName;  
    // TODO: Add other properties  
  
    return person;  
}
```

# Methods of Objects

- Functions within a JavaScript object are called **methods**
- We can **define** methods using several syntaxes:

```
let person = {  
    sayHello: function() {  
        console.log('Hi, guys');  
    }  
}
```

```
let person = {  
    sayHello() {  
        console.log('Hi, guys');  
    }  
}
```

- We can **add** a method to an already defined object

```
let person = { name:'Peter', age: 20 };  
person.sayHello = () => console.log('Hi, guys');
```

# Built-in Method Library

- Get array of all property **names** (keys)

```
Object.keys(cat); // ['name', 'age']
```

- Get array with of all property **values**

```
Object.values(cat); // ['Tom', 5]
```

- Get and array of all properties as **key-value tuples**

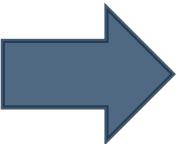
```
Object.entries(cat); // [['name', 'Tom'], ['age', 5]]
```

cat	
'name'	'Tom'
'age'	5

# Problem: City

- Receive an object, which holds **name**, **area**, **population**, **country**, and **postcode**
- Loop through all the keys and print them with their values

```
Sofia  
492  
1238438  
Bulgaria  
1000
```



```
name -> Sofia  
area -> 492  
population -> 1238438  
country -> Bulgaria  
postCode -> 1000
```

# Solution: City

- Get the object **entries**
- Loop through the object **entries** using **for-of** loop
- Print the object **keys** and **values**

```
function cityInfo(city) {  
  let entries = Object.entries(city);  
  for (let [ key, value ] of entries) {  
    console.log(`${key} -> ${value}`);  
  }  
}
```

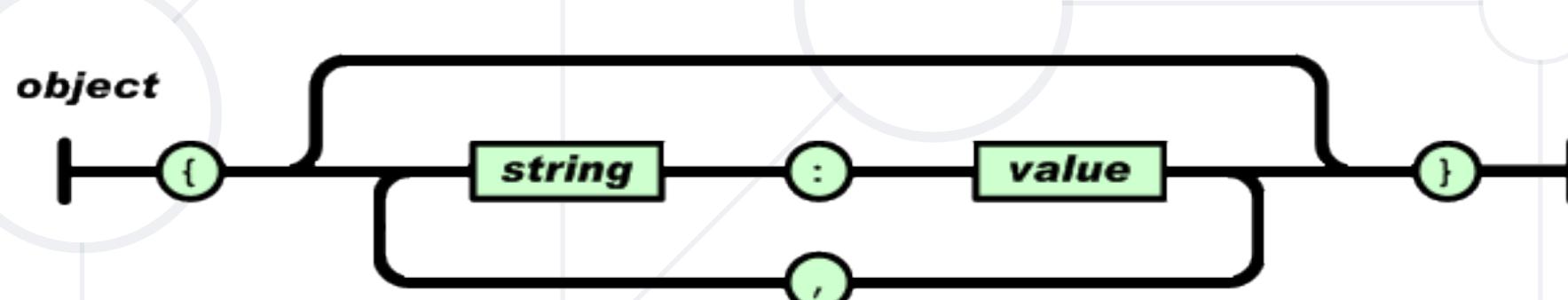


**JSON**

**JavaScript Object Notation**

# What is JSON

- JSON stands for JavaScript Object Notation
- Open-standard file format that uses text to transmit data objects
- JSON is language independent
- JSON is "self-describing" and easy to understand



# JSON Usage

- Exchange data between **browser** and **server**
- JSON is a **lightweight** format compared to XML
- JavaScript has built-in functions to **parse JSON** so it's easy to use
- JSON uses **human-readable** text to transmit data



# JSON Example

Keys are in double quotes

Keys and values separated by :

Brackets define a JSON

```
{  
  "name": "Ivan",  
  "age": 25,  
  "grades": {  
    "Math": [2.50, 3.50],  
    "Chemistry": [4.50]  
  }  
}
```

It is possible to have nested objects

In JSON we can have arrays

# JSON Methods

- We can convert object into **JSON** string using **JSON.stringify(object)** method

```
let text = JSON.stringify(obj);
```

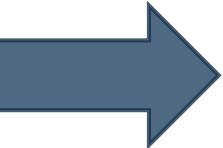
- We can convert JSON string into object using **JSON.parse(text)** method

```
let obj = JSON.parse(text);
```

# Problem: Convert to Object

- Write a function, that receives a string in **JSON** format and converts it to object
- Print the entries of the object

```
'{  
  "name": "George",  
  "age": 40,  
  "town": "Sofia"  
'
```



```
name: George  
age: 40  
town: Sofia
```

# Tips: Convert to Object

- Use `JSON.parse()` method to parse JSON string to an object
- Use `Object.entries()` method to get object's properties: names and values
- Loop through the entries and print them

```
function objConverter(json) {  
    // TODO: Use the tips to write the function  
}
```

# Solution: Convert to Object

```
function objConverter(json) {  
    let person= JSON.parse(json);  
  
    let entries = Object.entries(person);  
  
    for (let [key, value] of entries) {  
        console.log(` ${key}: ${value}`);  
    }  
}
```

# Problem: Convert to JSON

- Write a function that receives a first name, last name, hair color and sets them to an object.
- Convert the object to **JSON string** and print it.

```
'George',  
'Jones',  
'Brown'
```



```
{"name": "George", "lastName":  
"Jones", "hairColor": "Brown"}
```

# Tips: Convert to JSON

- Create an object with the given input
- Use **JSON.stringify()** method to parse object to JSON string
- Keep in mind that the property name in the JSON string will be **exactly the same** as the property name in the object

```
function solve(name, lastName, hairColor){  
    // TODO: Use the tips and write the code  
}
```

# Solution: Convert to JSON

```
function convertJSON(name, lastName, hairColor) {  
    let person = {  
        name,  
        lastName,  
        hairColor  
    };  
    console.log(JSON.stringify(person));  
}
```



# Associative Arrays

Storing Key-Value Pairs

# What is an Associative Array ?

- Arrays indexed by **string keys**
- Hold a set of pairs **[key => value]**
  - The key is a **string**
  - The **value** can be of **any type**



Key	Value
John Smith	+1-555-8976
Lisa Smith	+1-555-1234
Sam Doe	+1-555-5030

# Declaration

- An associative array in JavaScript is just an **object**
- We can declare it **dynamically**

```
let assocArr = {  
    'one': 1,  
    'two': 2,  
    'three': 3,  
    [key]: 6  
};
```

```
assocArr['four'] = 4;
```

```
assocArr.five = 5;
```

```
let key = 'six';  
assocArr[key] = 6;
```

Quotes are used if the key contains **special characters**

Valid ways to access values through **keys**

# Using for – in

- We can use **for-in** loop to iterate through the keys

```
let assocArr = {};
assocArr['one'] = 1;
assocArr['two'] = 2;
assocArr['three'] = 3;

for(let key in assocArr) {
  console.log(key + " = " + assocArr[key]);
}
```

```
// one = 1
// two = 2
// three = 3
```



# Problem: Phone Book

- Write a function that reads **names** and **numbers**
- Store them in an associative array and print them
- If the same name occurs, save the **latest** number

```
[ 'Tim 0834212554',  
  'Peter 0877547887',  
  'Bill 0896543112',  
  'Tim 0876566344' ]
```



```
Tim -> 0876566344  
Peter -> 0877547887  
Bill -> 0896543112
```

# Solution: Phone Book

```
function solve(input) {  
    let phonebook = {};  
    for (let line of input) {  
        let tokens = line.split(' ');  
        let name = tokens[0];  
        let number = tokens[1];  
        phonebook[name] = number;  
    }  
    for (let key in phonebook) {  
        console.log(`${key} -> ${phonebook[key]}`);  
    }  
}
```

# Manipulating Associative Arrays

- Check if a key is **present**:

```
let assocArr = { /* entries */ };
if (assocArr.hasOwnProperty('John Smith')) { /* Key found */ }
```

- Remove entries:

```
delete assocArr['John Smith'];
```

# Problem: Meetings

- Write a function that reads **weekdays** and **names**
- Print a **success** message for every successful appointment
- If the same weekday occurs a second time, print a **conflict message**
- In end, print a list of all meetings
- See **example** input and output on **next slide**

# Example: Meetings

- Parsing input and success/conflict messages

```
[ 'Monday Peter',  
  'Wednesday Bill',  
  'Monday Tim',  
  'Friday Tim' ]
```



Scheduled for Monday  
Scheduled for Wednesday  
Conflict on Monday!  
Scheduled for Friday

- Final list output

```
Monday -> Peter  
Wednesday -> Bill  
Friday -> Tim
```

# Solution: Meetings

```
function solve(input) {  
    let meetings = {};  
    for (let line of input) {  
        let [weekday, name] = line.split(' ');  
  
        if (meetings.hasOwnProperty(weekday)) {  
            console.log(`Conflict on ${weekday}!`);  
        } else {  
            meetings[weekday] = name;  
            console.log(`Scheduled for ${weekday}`);  
        }  
    }  
    // TODO: Print result  
}
```

# Sorting Associative Arrays

- Objects **cannot be sorted**; they must be converted first
  - Convert to **array** for **sorting**, **filtering** and **mapping**:

```
let phonebook = { 'Tim': '0876566344' ←  
                  'Bill': '0896543112' };  
  
let entries = Object.entries(phonebook);  
console.log(entries); //Array of arrays with two elements each  
// [ ['Tim', '0876566344'], ←  
  //   ['Bill', '0896543112'] ]  
  
let firstEntry = entries[0];  
console.log(firstEntry[0]); //Entry key -> 'Tim'  
console.log(firstEntry[1]); //Entry value -> '0876566344'
```

The entry is turned into an array of [key, value]

# Sorting By Key

- The **entries** array can be **sorted**, using a **Compare function**
  - To **sort by key**, use the **first element** of each entry

```
entries.sort((a, b) => {  
    keyA = a[0];  
    keyB = b[0];  
    // Perform comparison and return negative, 0 or positive  
});
```

- You can also **destructure** the entries

```
entries.sort(([keyA, valueA], [keyB, valueB]) => {  
    // Perform comparison and return negative, 0 or positive  
});
```

# Problem: Sort Address Book

- Write a function that reads **names** and **addresses**
- Values will be separated by ":"
- If same name occurs, save the **latest** address
- Print list, **sorted** alphabetically by **name**

```
['Tim:Doe Crossing',  
 'Bill:Nelson Place',  
 'Peter:Carlyle Ave',  
 'Bill:Ornery Rd']
```



```
Bill -> Ornery Rd  
Peter -> Carlyle Ave  
Tim -> Doe Crossing
```

# Solution: Sort Address Book

```
function solve(input) {
    let addressbook = {};
    for (let line of input) {

        let [name, address] = line.split(':');
        addressbook[name] = address;

    }
    let sorted = Object.entries(addressbook);
    sorted.sort((a, b) => a[0].localeCompare(b[0]));

    // TODO: Print result
}
```

# Array and Object Destructuring

- The **destructuring assignment** syntax makes it possible to unpack values from arrays, or properties from objects, into distinct variables.
- On the left-hand side of the assignment to define what values to unpack from the sourced variable.

```
const x = [1, 2, 3, 4, 5];
const [y, z] = x;
console.log(y); // 1
console.log(z); // 2
```

```
obj = { a: 1, b: 2 };
const { a, b } = obj;
// is equivalent to:
// const a = obj.a;
// const b = obj.b;
```

# Sorting By Value

- To **sort by value**, use the **second element** of each entry

```
entries.sort((a, b) => {  
    valueA = a[1];  
    valueB = b[1];  
    // Perform comparison and return negative, 0 or positive  
});
```

- You can also **destructure** the entries

```
entries.sort(([keyA, valueA],[keyB, valueB]) => {  
    // Perform comparison and return negative, 0 or positive  
});
```



# Classes

## Object Models

# What are Classes?

- **Templates** for creating objects
- Defines **structure** and **behavior**
- An object created by the class pattern is called an an **instance** of that class
- A class has a **constructor** – method called automatically to create an object
  - It **prepares** the new object for use
  - Can **receive** parameters and **assign** them to properties



# Class Declaration

Use the **class** keyword followed by a name

```
class Student {  
    constructor(name) {  
        this.name = name;  
    }  
}
```

The **constructor** is  
a special method for  
creating and  
initializing an object

# Class Example

- Creating a class:

**this** keyword is used to set a property of the object to a given value

```
class Student {  
    constructor(name, grade) {  
        this.name = name;  
        this.grade = grade;  
    }  
}
```

- Creating an **instance** of the class:

```
let student = new Student('Peter', 5.50);
```

# Functions in a Class

- Classes can also have functions as property, called **methods**:

```
class Dog {  
  constructor(name) {  
    this.name = name;  
  }  
  speak() {  
    console.log(`#${this.name} says Woof!`);  
  }  
}  
  
let dog = new Dog('Sparky');  
dog.speak(); // Sparky says Woof!
```

**this** in the object  
refers to itself

We access the  
method as a regular  
property

# Problem: Cat

- Write a function that receives **array of strings** in the following format:  
`'{cat name} {age}'`
- Create a class **Cat** that receives the **name** and the **age** parsed from the input
- It should also have a method named **meow()** that will print  
`"{cat name}, age {age} says Meow"` on the console
- For each of the strings provided you must create a cat object

```
[ 'Mellow 2', 'Tom 5' ]
```



```
Mellow, age 2 says Meow  
Tom, age 5 says Meow
```

- Create a class
- Set properties name and age
- Set property '**meow**' to be a method that prints the result
- **Parse** the input data
- Create all objects using the class **constructor** and the parsed input data and store them in an array
- Loop through the array using **for...of** loop and invoke **.meow()** method

# Solution: Cat

```
function catCreator(arr) {  
    // TODO: Create the Cat class  
    let cats = [];  
    for (let i = 0; i < arr.length; i++) {  
        let catData = arr[i].split(' ');\n        cats.push(new Cat(catData[0], catData[1]));  
    }  
    // TODO: Iterate through cats[] and invoke .meow()  
    // using for..of Loop  
}
```



# Live Exercises

# Summary

- Objects hold **key-value pairs**
  - Access value by indexing with key
  - **Methods** are functions
- **References** point to data in memory
- **Parse** and **stringify** objects in **JSON**
- **Classes** are templates for objects



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg)
  - Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



# Questions?



SoftUni



Software  
University



SoftUni  
Creative



SoftUni  
Digital



SoftUni  
Foundation



SoftUni  
Kids



Finance  
Academy

# SoftUni Diamond Partners



**SUPER  
HOSTING  
.BG**

**INDEAVR**  
Serving the high achievers

 **SOFTWARE  
GROUP**

 **PHAR  
VISION**



**Coca-Cola HBC  
Bulgaria**

 **AMBITIONED**



**BOSCH**

 **SmartIT**

 **DXC  
TECHNOLOGY**

 **Flutter  
International**™

 **DRAFT  
KINGS**

 **Postbank**  
*Решения за твоето утре*

 **createX**

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



# DOM Introduction

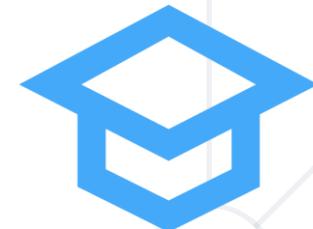
## Document Object Model



SoftUni Team

Technical Trainers

 Software  
University

The logo consists of a stylized orange 'S' shape followed by the text "Software University".

SoftUni



Software University

<https://softuni.bg>

# Table of Contents

1. Browser API
2. Document Object Model
3. HTML Elements
4. Targeting Elements
5. Using the DOM API



Have a Question?



**sli.do**

**#js-front-end**



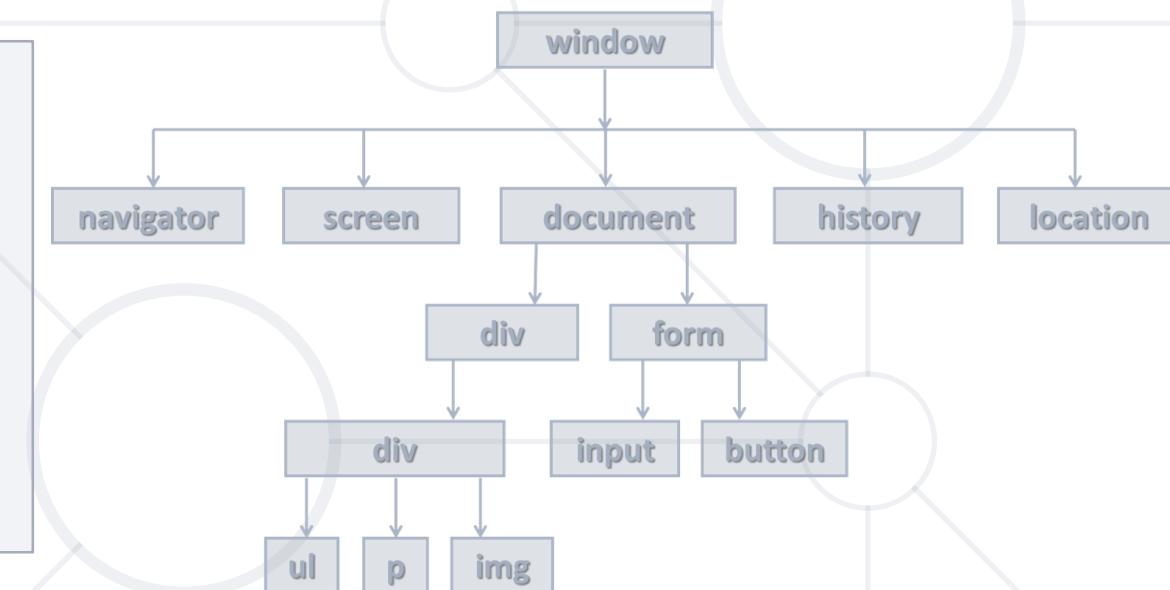
# Browser API

## Browser Object Model (BOM)

# Browser Object Model (BOM)

- Browsers expose some objects like **window**, **screen**, **navigator**, **history**, **location**, **document**, ...

```
console.dir(window);
console.dir(navigator);
console.dir(screen);
console.dir(location);
console.dir(history);
console.dir(document);
```



- Most of this **API** will be examined in the **next course**



# Global Context in the Browser

- The **global object** in the browser is **window**

```
let b = 8;  
console.log(this.b); // undefined
```

```
var a = 5;  
console.log(this.a); // 5
```

```
function foo() {  
  console.log("Simple function call");  
  console.log(this === window); // true  
}  
foo();
```



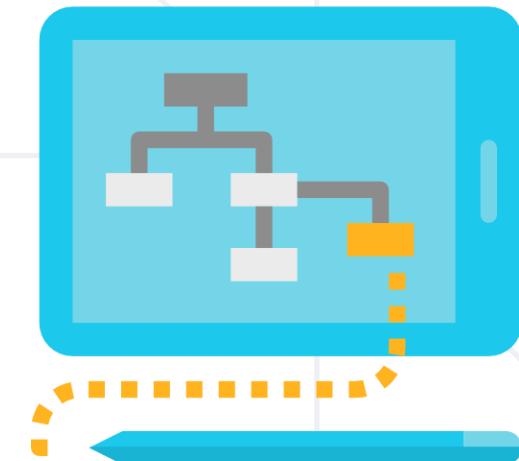


# Document Object Model (DOM)

Document with a Logical Tree

# Document Object Model

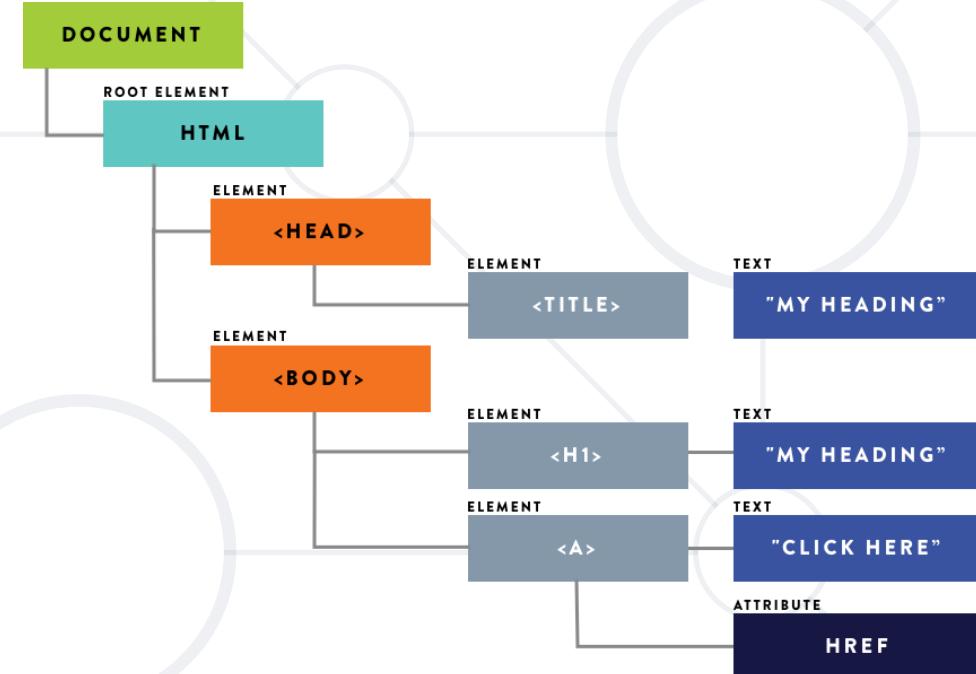
- The **DOM** represents the document as **nodes** and **objects**
  - That way, the programming languages **can connect** to the page
- The **HTML DOM** is an **Object Model for HTML**. It defines:
  - HTML elements as **objects**
  - **Properties**
  - **Methods**
  - **Events**



# From HTML to DOM Tree

- The browser **parses** HTML and creates a **DOM Tree**

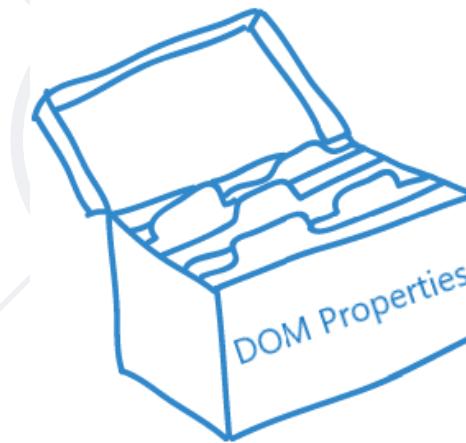
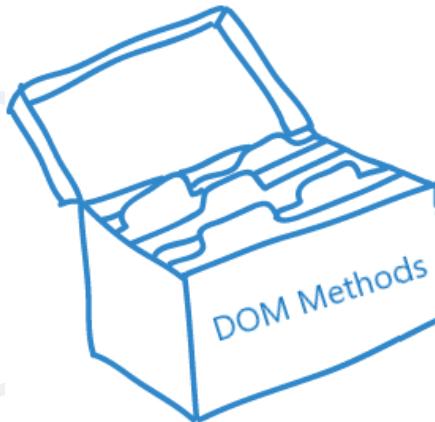
```
<html>
  <head>
    <title>My Heading</title>
  </head>
  <body>
    <h1>My Heading</h1>
    <a href="/about">Click Here</a>
  </body>
</html>
```



- The elements are **nested** in each other and create a **hierarchy**
  - Like the hierarchy of a **street address** – Country, City, Street, etc.

# DOM Methods

- **DOM Methods** - actions you can perform on HTML elements
- **DOM Properties** - values of HTML elements that you can set or change



# Example: DOM Methods

- HTML DOM **method** is an action you can do (like **add** or **delete** an HTML element)

```
<!doctype html>
...<html> == $0
  <head>
    <title>Intro to DOM</title>
  </head>
  <body>
    <h1>Introduction to DOM</h1>
    <ul>
      <li>DOM Methods example</li>
      <li>DOM Properties example</li>
    </ul>
  </body>
</html>
```

```
>
let h1Element = document.getElementsByTagName('h1')[0];
console.log(h1Element);
<h1>Introduction to DOM</h1>
```

# Example: DOM Methods

- HTML DOM **property** is a value that you can **get** or **set** (changing the content of an HTML element)

```
<!doctype html>
...<html> == $0
  <head>
    <title>Intro to DOM</title>
  </head>
  <body>
    <h1>Introduction to DOM</h1>
    <ul>
      <li>DOM Methods example</li>
      <li>DOM Properties example</li>
    </ul>
  </body>
</html>
```

```
let secondLi = document.getElementsByTagName('li')[1];
secondLi.innerHTML += " - DONE"
```

## Introduction to DOM

- DOM Methods example
- DOM Properties example - DONE

# Using the DOM API

- JavaScript can **interact** with web pages via the **DOM API**:
  - Check the **contents** and **structure** of elements on the page
  - Modify element **style** and **properties**
  - Read **user input** and react to **events**
  - **Create** and **remove** elements
- Most actions are performed when an **event** occurs
  - Events are "**fired**" when something of interest happens
- All of this **and more** will be examined in upcoming lessons

# JavaScript in the Browser

- Code can be **executed in the page** in different ways:
  - Directly in the **developer console** – when **debugging**
  - As a page **event handler** – e.g., user **clicks** on a button
- Via **inline script**, using **<script>** tags
- By **importing** from external file – most **flexible method**



```
<button onclick="console.log('Hello, DOM!')>Click Me</button> event
```

```
<script>
    function sum(a, b) {
        let result = a + b;
        return result;
    }
</script>
```



# HTML Elements

DOM Properties and HTML Attributes

# Elements and Properties

- The DOM Tree is comprised of **HTML elements**
- Elements are **JS objects** with **properties** and **methods**
  - They can be **accessed** and **modified** like regular objects
- To change the contents of the page:
  - **Select** an element to obtain a **reference**
  - **Modify** its **properties**

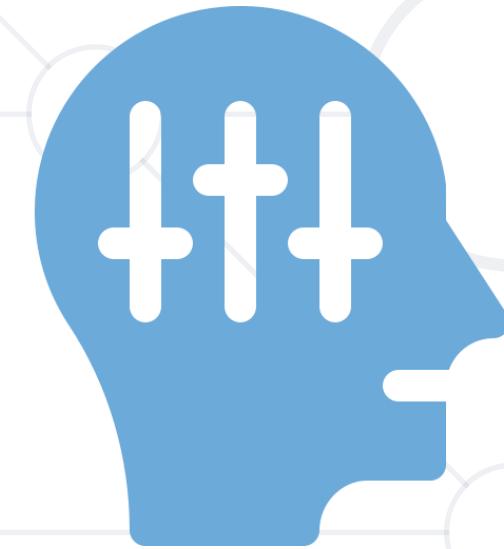
# Attributes and Properties

- Attributes are defined by **HTML**
  - Attributes **initialize** DOM properties
  - **Property** values can **change** via the DOM API
- The HTML **attribute** and the DOM **property** are technically **not the same thing**
- Since the **outcome is the same**, in practice you will **almost never** encounter a difference!



# DOM Manipulations

- The **HTML DOM** allows JavaScript to change the content of **HTML elements**
  - **innerHTML**
  - **textContent**
  - **value**
  - **style**
  - And many others to be discussed in upcoming lessons



# Accessing Element HTML

- To access raw HTML:

```
element.innerHTML = "<p>Welcome to the DOM</p>";
```

```
<html>
  <head></head>
  <body>
    <div id="main">This is JavaScript!</div>
  </body>
</html>
```

```
<html>
  <head></head>
  <body>
    <div id="main">
      <p>Welcome to the DOM</p>
    </div>
  </body>
</html>
```

- This will be **parsed** – beware of **XSS attacks!**
- Changing **textContent** or **innerHTML** removes all child nodes

# Accessing Element Text

- The contents of HTML elements are stored in text nodes
  - To access the contents of an element:

```
let text = element.textContent; //This is JavaScript!
element.textContent = "Welcome to the DOM";
```

```
<html>
  <head></head>
  <body>
    <div id="main">This is JavaScript!</div>
  </body>
</html>
```



```
<html>
  <head></head>
  <body>
    <div id="main">Welcome to the DOM</div>
  </body>
</html>
```

- If the element has children, returns all text **concatenated**

# Accessing Element Values

- The **values** of input elements are **string properties** on them:

```
<html>
  <head></head>
  <body>
    <div id="main">
      <p>Welcome to the DOM</p>
      <input id="num1" type="text">
    </div>
  </body>
</html>
```

```
type: "text"
useMap: ""
validationMessage: ""
▶ validity: ValidityState
value: "56"
valueAsNumber: NaN
▶ webkitEntries: Array[0]
webkitdirectory: false
width: 0
```

```
let num = Number(element.value);
element.value = 56;
```

# Problem: Edit Element

- Create function **edit()** that takes **three** parameters:
  - A **reference** to an HTML element
  - Two strings – **match** and **replacer**
- Replace all occurrences of **match** inside the **text content** of the given element with **replacer**

```
▼ <body>  
  <h1>Hello, %insert name here!</h1>  
</body>
```

'%insert name here%',  
'Document Object Model'

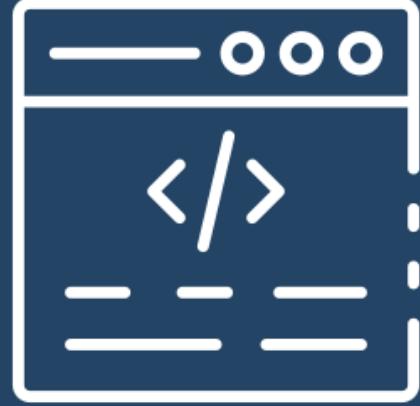


```
▼ <body>  
  <h1>Hello, Document Object Model!</h1>  
</body>
```

# Solution: Edit Element

```
function edit(ref, match, replacer) {  
    const content = ref.textContent;  
    const matcher = new RegExp(match, 'g');  
    const edited = content.replace(matcher, replacer);  
    ref.textContent = edited;  
}
```

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/4364#0>



# Targeting DOM Elements

Obtaining Element References

# Targeting Elements

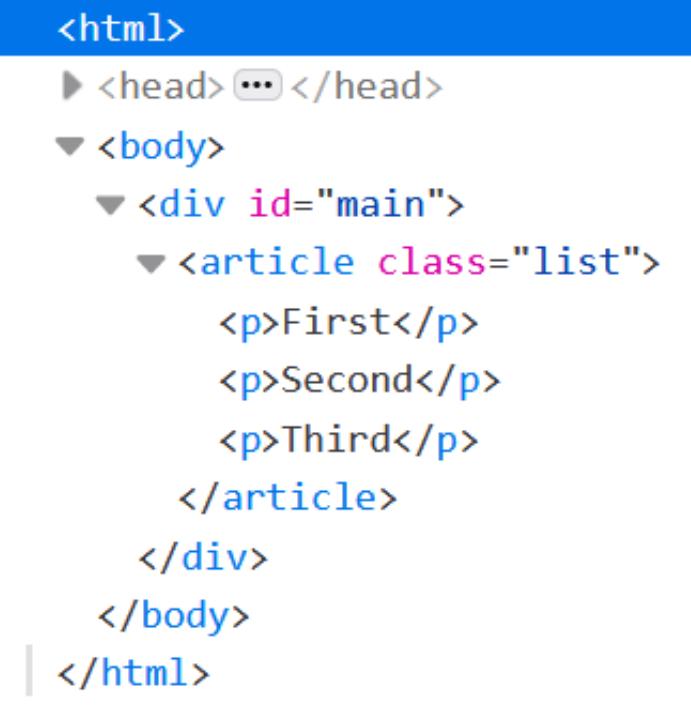
- There are a few ways to **find** a certain **HTML element** in the **DOM**:
  - By ID - **getElementById()**
  - By class name - **getElementsByClassName()**
  - By tag name - **getElementsByTagName()**
  - By CSS selector - **querySelector()**,  
**querySelectorAll()**
- These methods return a **reference** to the element, which can be **manipulated** with JavaScript



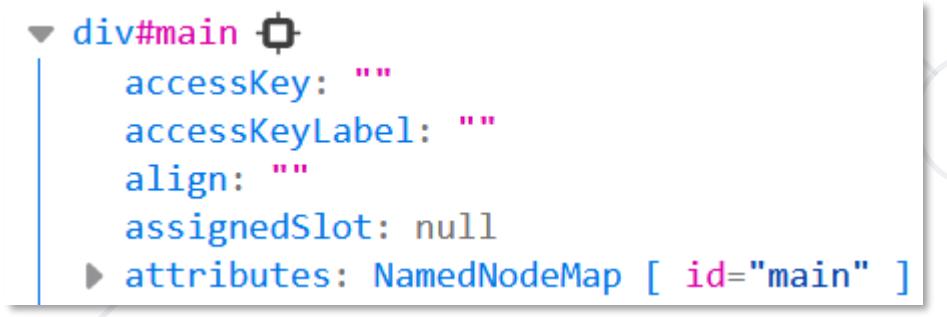
# Targeting by ID - Example

- The **ID attribute** must be **unique** on the page

```
const element = document.getElementById('main');
console.log(element);
```



```
<html>
  <head> ...
  <body>
    <div id="main">
      <article class="list">
        <p>First</p>
        <p>Second</p>
        <p>Third</p>
      </article>
    </div>
  </body>
</html>
```



```
div#main
  accessKey: ""
  accessKeyLabel: ""
  align: ""
  assignedSlot: null
  attributes: NamedNodeMap [ id="main" ]
```

# Targeting by Tag and Class Names – Example

- The **tag name** specifies the **type** of element – **div**, **p**, **ul**, etc.

```
const elements = document.getElementsByTagName('p');  
// Select all paragraphs on the page
```

- Class names** are used for **styling** and easier **selection**

```
const elements = document.getElementsByClassName('list');  
// Select all elements having a class named 'list'
```

- Both methods return a live **HTMLCollection**

- Even if** only **one** element is selected! This is a **common mistake**

- **CSS selectors** are strings that follow CSS syntax for matching
- They allow very fast and powerful element matching, e.g.:
  - "**#main**" - returns the element with ID "main"
  - "**#content div**" - selects all **<div>**s inside **#content**
  - "**.note, .alert**" - all elements with class "note" or "alert"
  - "**input[name='login']**" - **<input>** with name "login"

# CSS Selectors - Example

- Select the **first matching** element

```
const mainDiv = document.querySelector('#main');
// Select the element with ID 'main'

const element = document.querySelector('p');
// Select the first paragraph on the page
```

- Select **all** matching elements
  - Returns a **static NodeList**

```
const elements = document.querySelectorAll('article.list');
// Select all <article> elements having a class named 'list'
```

# NodeList vs. HTMLCollection

- Both interfaces are **collections** of **DOM nodes**
- **NodeList** can contain **any** node type, including **text** and **whitespace**
- **HTMLCollection** contains only **Element nodes**
- Both have **iteration** methods, **HTMLCollection** has an extra **namedItem** method
- **HTMLCollection** is **live**, while **NodeList** can be either **live** or **static**



# Iterating Element Collections

- NodeList and HTMLCollection are NOT arrays but can be indexed and iterated

```
const elements = document.querySelectorAll('p');
const first = elements[0];
// Select the first paragraph on the page
for (let p of elements) { /* ... */ }
// Iterate over all entries
```

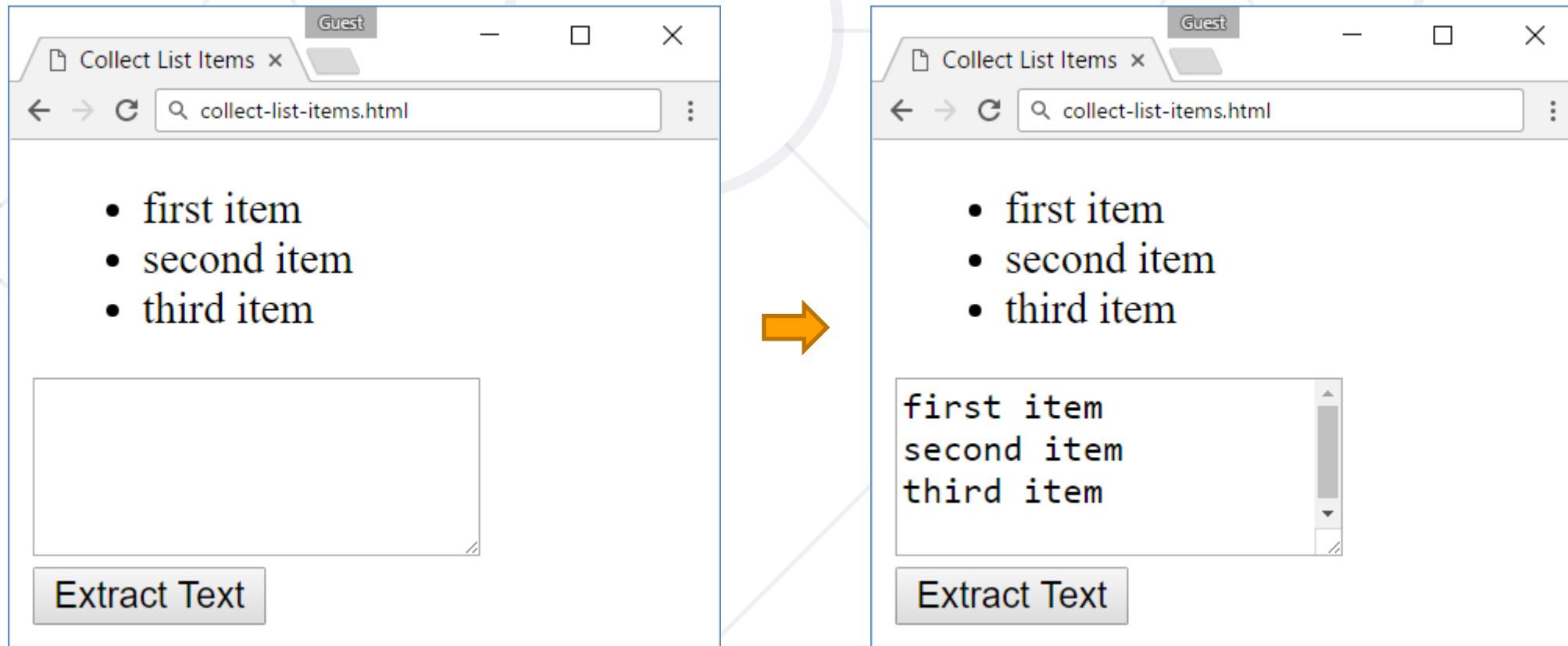
- Both can be explicitly converted to an array

```
const elementArray = Array.from(elements);
const elementArr2 = [...elements]; // Spread syntax
```



# Problem: Collect List Items

- Collect the **list items** from given HTML list and append their **text** to given **text area**



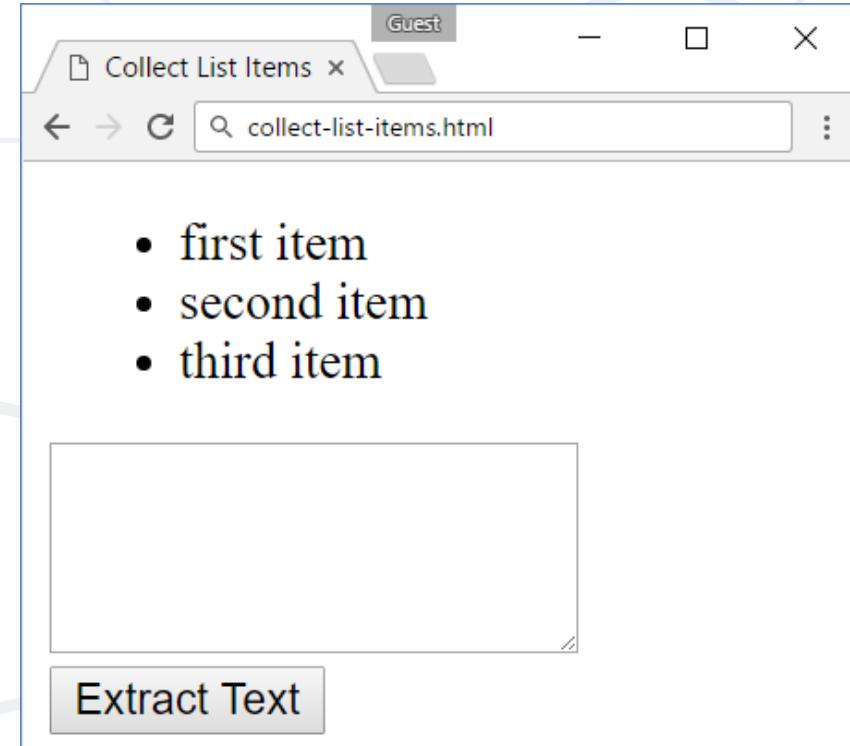
# Problem: Collect List Items – HTML

```
<ul id="items">
  <li>first item</li>
  <li>second item</li>
  <li>third item</li>
</ul>

<textarea id="result">
</textarea>

<br>

<button onclick="extractText()">
Extract Text</button>
```



# Solution: Collect List Items

```
function extractText() {  
    let itemNodes =  
        document.querySelectorAll("ul#items li");  
    let textarea =  
        document.querySelector("#result");  
    for (let node of itemNodes) {  
        textarea.value += node.textContent + "\n";  
    }  
}
```

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/4364#1>

# Parents and Child Elements

- Every DOM Element has a **parent**
  - Parents can be accessed by property **parentElement** or **parentNode**

```
▼<div>
  <p>This is a paragraph.</p>
  <p>This is another paragraph.</p>
</div>
```

Accessing the  
first child

```
let firstP = document.getElementsByTagName('p')[0];
console.log(firstP.parentElement);
```

Accessing the  
child's parent

```
► <div>...</div>
```

# Parents and Child Elements

- When some element contains other elements, that means he is **parent** of those elements
- They are **children** to the **parent**. They can be accessed by property **children**

```
▼<div>
  <p>This is a paragraph.</p>
  <p>This is another paragraph.</p>
</div>
```

```
▼HTMLCollection(2) [p, p]
  ► 0: p
  ► 1: p
    length: 2
```

```
let pElements = document.getElementsByTagName('div')[0].children;
```

Returns live  
HTMLCollection



# Using the DOM API

Common Techniques and Scenarios

# External Page Scripts

- Page scripts can be **loaded** from an external file

- Use the **src** attribute of the **script element**

```
<script src="app.js"></script>
```

- **Functions** from script files are in the **global scope**
    - Can be referenced and **executed** from **events** and **inline** scripts
    - **Multiple** script files in a page can see **each other**
  - Pay attention to **load order!**

# Problem: Sum Numbers

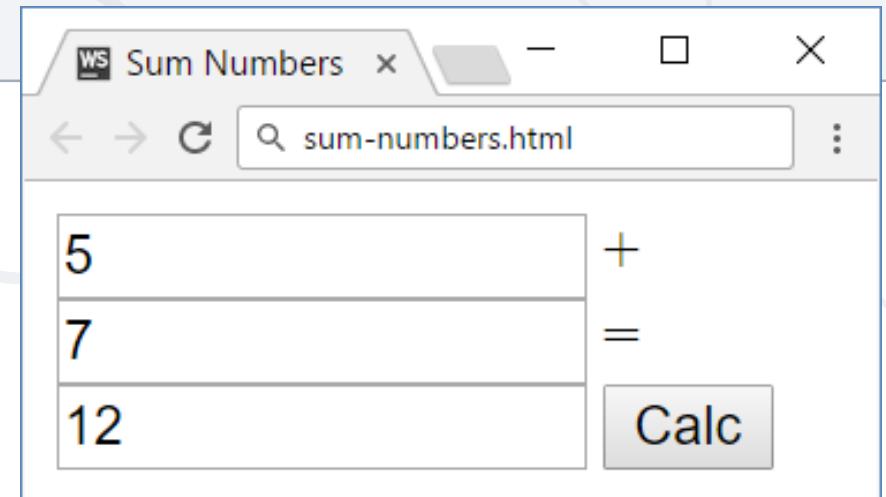
- Write a JS function to sum two numbers (fill the missing code)

```
<input type="text" id="num1" /> +
<input type="text" id="num2" /> =
<input type="text" id="sum" readonly="readonly" />
<input type="button" value="Calc" onclick="calc()" />

<script src="calc.js"></script>
```

calc.js

```
function calc() {
    // TODO
}
```



# Solution: Sum Numbers

```
function calc() {  
    let num1 = document.getElementById('num1').value;  
    let num2 = document.getElementById('num2').value;  
  
    let sum = Number(num1) + Number(num2);  
  
    document.getElementById('sum').value = sum;  
}
```

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/4364#2>

# Control Content via Visibility

- Content can be **hidden** or **revealed** by changing its **display** style
  - This is a **common technique** to display content dynamically
- To **hide** an element:

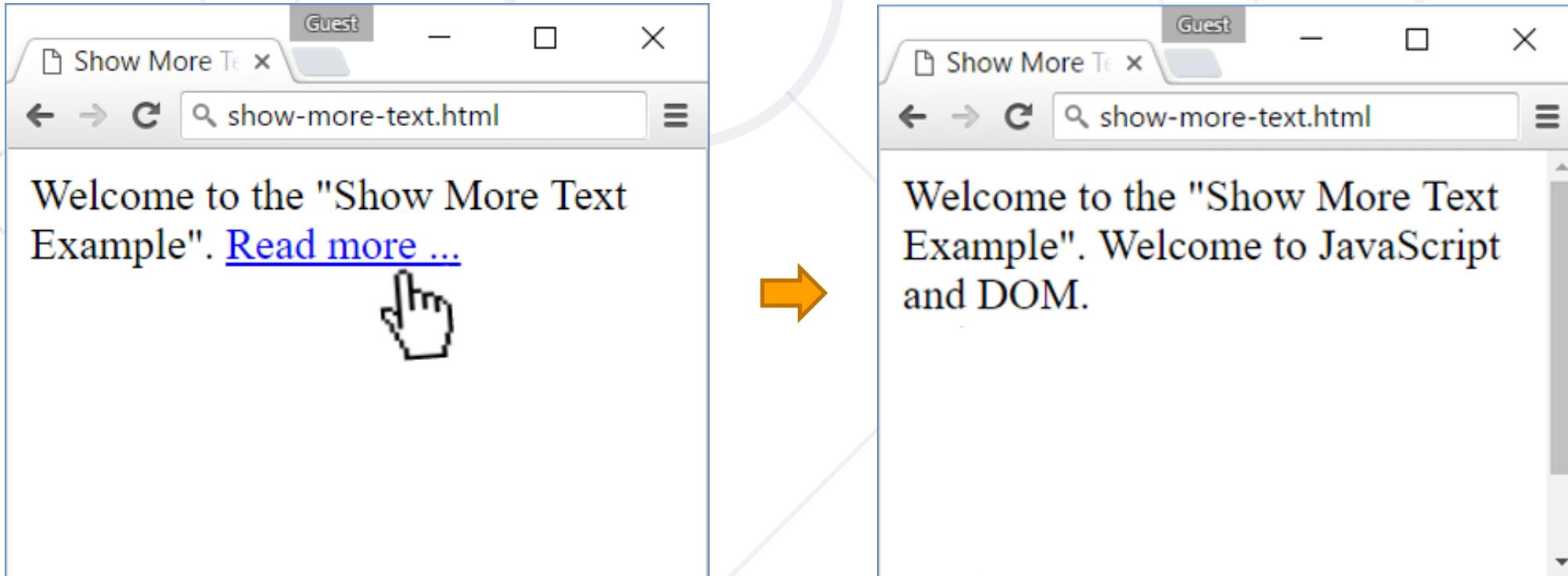
```
const element = document.getElementById('main');
element.style.display = 'none';
```

- To **reveal** an element, set **display** to anything that isn't '**none**' (including **empty string**)

```
element.style.display = ''; // Can be 'inline', 'block', etc.
```

# Problem: Show More Text

- A HTML page holds a short text + link "*Read more ...*"
  - Clicking on the link shows more text and hides the link



# Problem: Show More Text – HTML

Welcome to the "Show More Text Example".

```
<a href="#" id="more" onclick="showText()">Read more ...</a>  
  
<span id="text" style="display:none">Welcome to ...</span>  
  
<script>  
  function showText() {  
    // TODO  
  }  
</script>
```

- See the DOM tree here:  
<http://software.hixie.ch/utilities/js/live-dom-viewer/?saved=4275>

# Solution: Show More Text

```
Welcome to the "Show More Text Example". <a href="#" id="more" onclick="showText()">Read more ...</a>
<span id="text" style="display:none">Welcome to ...</span>
<script>
  function showText() {
    document.getElementById('text')
      .style.display = 'inline';
    document.getElementById('more')
      .style.display = 'none';
  }
</script>
```



Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/4364#3>

# Match n-th Child

- Sometimes we need to target an element based on its **relation** to other **similar elements**
  - E.g., **row** or **column** in a table, **list item**, etc.
- Can be done either by **index** or with a **CSS selector**

```
const list = document.getElementsByTagName('ul')[0];
// First <ul> on the page
```

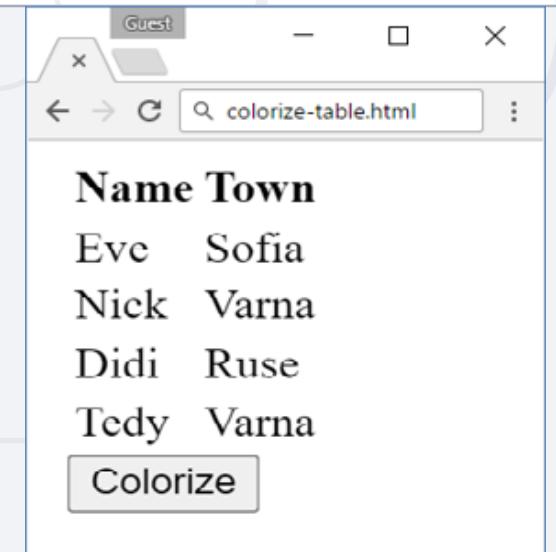
```
const thirdLi = list.getElementsByTagName('li')[2];
// Third <li> inside the selected <ul>
```

```
const thirdLi = document.querySelector('ul li:nth-child(3)');
// Third <li> inside the first <ul> on the page
```

# Problem: Colorize Table Rows

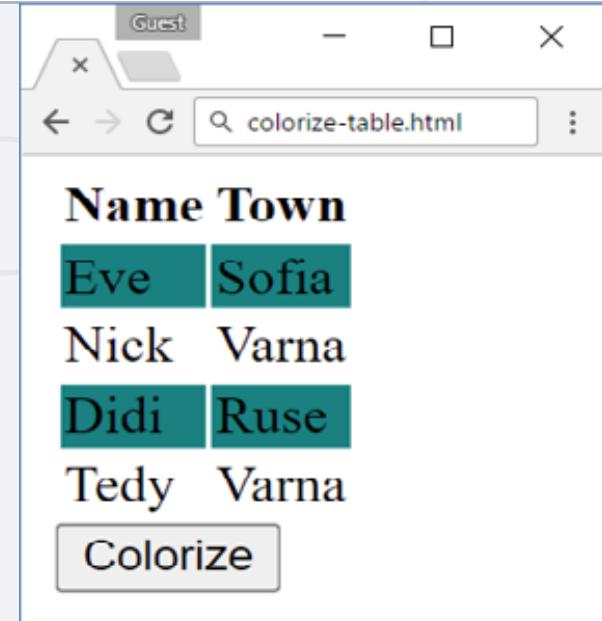
- A HTML page holds a **table** with **rows**
  - On button click, colorize in color "**teal**" all even rows

```
<table border="1">
  <tr><th>Name</th><th>Town</th></tr>
  <tr><td>Eve</td><td>Sofia</td></tr>
  <tr><td>Nick</td><td>Varna</td></tr>
  <tr><td>Didi</td><td>Ruse</td></tr>
  <tr><td>Tedy</td><td>Varna</td></tr>
</table>
<button onclick="colorizeRows()">Colorize</button>
```



# Solution: Colorize Table Rows

```
function colorizeRows() {  
    let rows = document.  
        querySelectorAll("table tr");  
    let index = 0;  
    for (let row of rows) {  
        index++;  
        if (index % 2 == 0)  
            row.style.background = "teal";  
    }  
}
```



Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/4364#4>

# Problem: Sum Table

- Find the **first table** and sum all values in the **last column**
- Display the result inside element with **ID "sum"**



Product Cost	
Beer	2.88
Fries	2.15
Burger	4.59
Total:	
	<b>Sum</b>

Product Cost	
Beer	2.88
Fries	2.15
Burger	4.59
Total:	9.62
	<b>Sum</b>

# Problem: Sum Table (2)

- Sample HTML

```
<table>
  <tbody>
    <tr><th>Product</th><th>Cost</th></tr>
    <tr><td>Beer</td> <td>2.88</td></tr>
    <tr><td>Fries</td> <td>2.15</td></tr>
    <tr><td>Burger</td> <td>4.59</td></tr>
    <tr><td>Total:</td> <td id="sum"></td></tr>
  </tbody>
</table>
<button onclick="sum()">Sum</button>
```

# Solution: Sum Table

```
function sum() {  
    let table = document.querySelectorAll("table tr");  
    let total = 0;  
    for (let i = 1; i < table.length; i++) {  
        let cols = table[i].children;  
        let cost = cols[cols.length - 1].textContent;  
        total += Number(cost);  
    }  
    document.getElementById("sum").textContent = total;  
}
```

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/4364#5>



# Live Demonstration

## Lab Problems

# Problem: Extract Parenthesis

- Extract all **parenthesized text** from a **target** paragraph
  - Your function will receive an element ID to parse
  - Return the result as string, joined by " ; "

```
...<!DOCTYPE html> == $0
<html lang="en">
  <head>...</head>
  <body>
    <p id="content">
      "
      The Rose Valley (Bulgaria) is located just south of the Balkan Mountains
      (Kazanlak).The most common oil-bearing rose found in the valley is the pink-
      petaled Damask rose (Rosa damascena Mill).
      "
    </p>
  </body>
</html>
```

```
>>>let text = extract("content")|
```



Bulgaria;  
Kazanlak;  
Rosa demascena Mill;

# Problem: Extract Parenthesis (2)

## ■ Sample HTML

```
<p id="content">
    The Rose Valley (Bulgaria) is located just south of the
    Balkan Mountains(Kazanlak).The most common oil-bearing rose
    found in the valley is the pink-petaled Damask rose (Rosa
    damascena Mill).
</p>
<p id="holder">
    Lorem ipsum dolor sit amet, (consectetur adipiscing elit),
    sed do eiusmod (tempor) incididunt ut labore (et dolore
    magna) aliqua.
</p>
```

# Solution: Extract Parenthesis

```
function extract(elementId) {  
    let para = document.getElementById(elementId).textContent;  
    let pattern = /\((([^)]+))\)/g;  
    let result = [];  
  
    let match = pattern.exec(para);  
    while(match) {  
        result.push(match[1]);  
        match = pattern.exec(para);  
    }  
  
    return result.join('; ');  
}
```

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/4364#6>

# Summary

- BOM – Browser API
- DOM
  - DOM is a programming API for HTML and XML documents
  - Selecting DOM elements
    - By Id
    - By Class Name
    - Query Selectors
  - DOM Properties & HTML Attributes



# Questions?



Software  
University



SoftUni  
Creative



SoftUni  
Digital



SoftUni  
Foundation



SoftUni  
Kids



Finance  
Academy



SoftUni

# SoftUni Diamond Partners



**SUPER  
HOSTING  
.BG**

**INDEAVR**  
Serving the high achievers

 **SOFTWARE  
GROUP**

 **PHAR  
VISION**



**Coca-Cola HBC  
Bulgaria**

 **AMBITIONED**



**BOSCH**

 **SmartIT**

 **DXC  
TECHNOLOGY**

 **Flutter  
International**™

 **DRAFT  
KINGS**

 **Postbank**  
*Решения за твоето утре*

 **createX**

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)

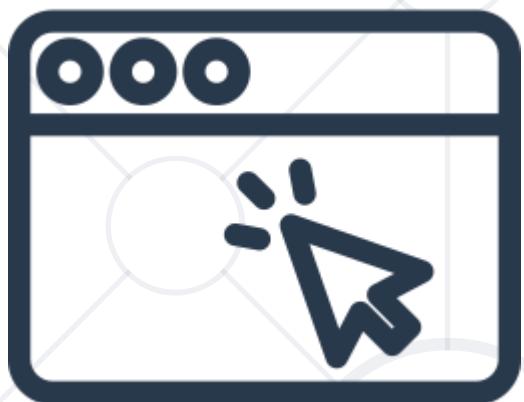


Software  
University



# DOM Events

Handling DOM Events, Propagation & Delegation



SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.bg>

# Table of Contents

1. Creating DOM Elements
2. Browser Events
3. Handling Events
4. Event Propagation

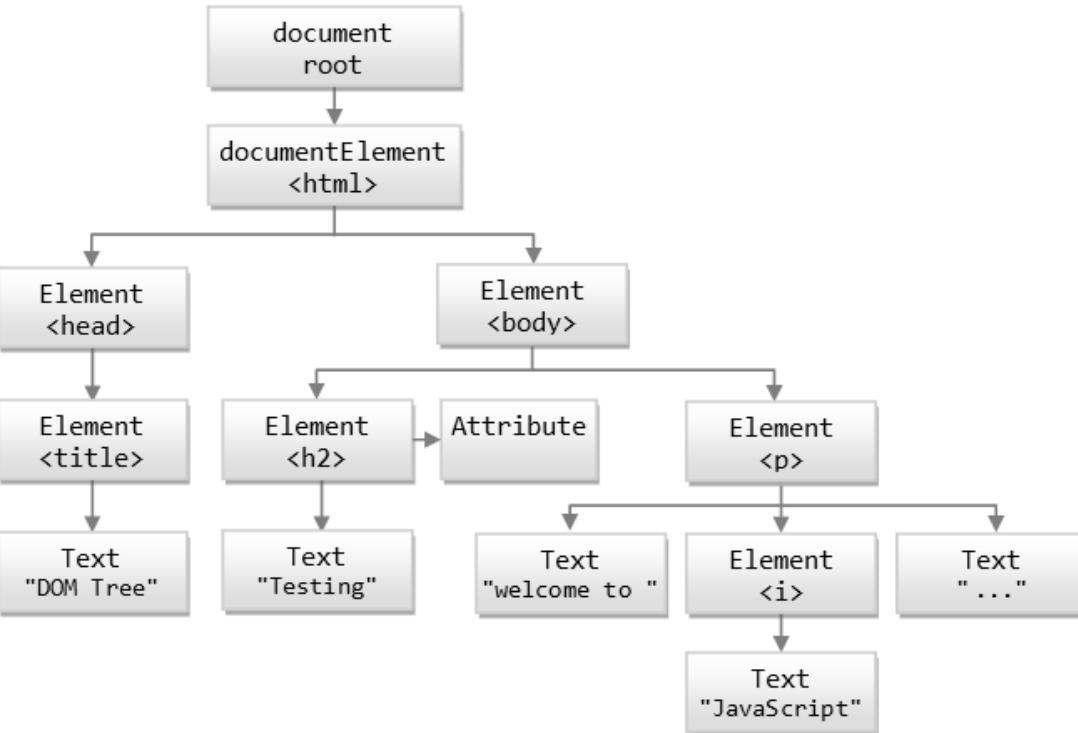


Have a Question?



**sli.do**

**#js-front-end**

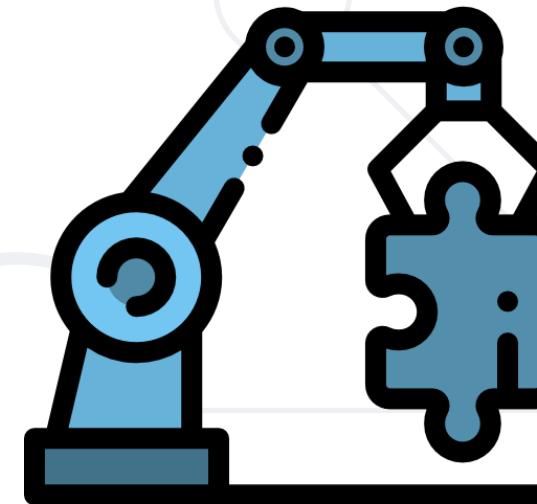


# DOM Manipulation

## Modify the DOM Tree

# DOM Manipulations

- We can **create**, **append** and **remove** HTML elements dynamically
  - **appendChild()**
  - **removeChild()**
  - **replaceChild()**



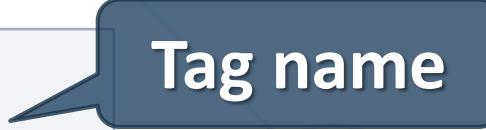
# Creating New DOM Elements

- HTML elements are created with `document.createElement`
  - This is called a **Factory Pattern**
- Variables holding HTML elements are **live**:
  - If you **modify** the contents of the variable, the DOM is **updated**
  - If you **insert** it somewhere in the DOM, the original is **moved**
- Text added to **textContent** will be **escaped**
- Text added to **innerHTML** will be **parsed** and turned into actual HTML elements → beware of **XSS attacks!**

# Creating DOM Elements

- Creating a new DOM element

```
let p = document.createElement("p");
let li = document.createElement("li");
```



Tag name

- Create a copy / cloning DOM element

```
let li = document.getElementById("my-list");
let newLi = li.cloneNode(true);
```

- Elements are created **in memory** – they don't exist on the page
- To become visible, they must be **appended** to the DOM tree

# Manipulating Node Hierarchy

- **appendChild** - Adds a new child, as the **last child**

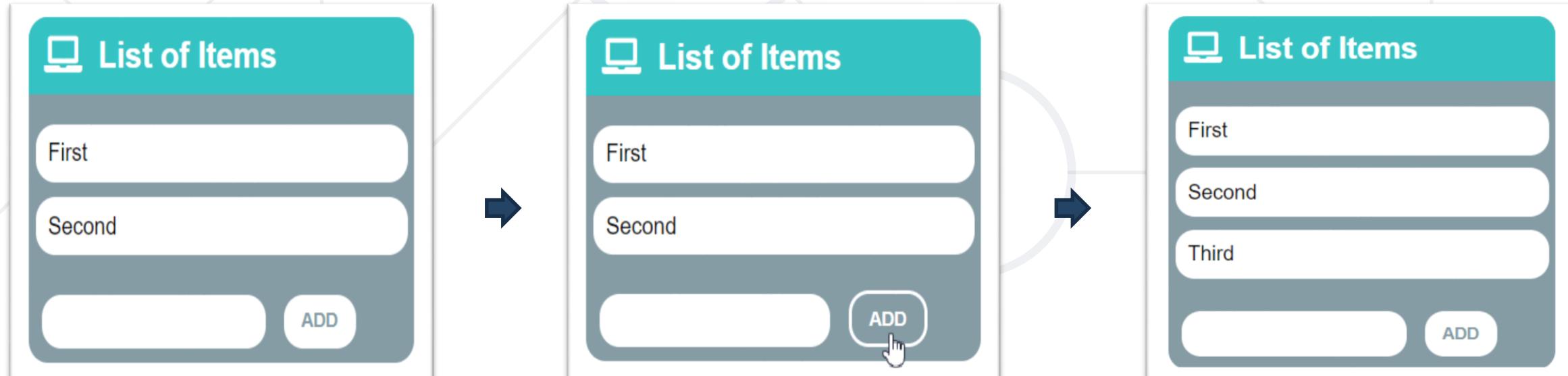
```
let p = document.createElement("p");
let li = document.createElement("li");
li.appendChild(p);
```

- **prepend** - Adds a new child, as the **first child**

```
let ul = document.getElementById("my-list");
let li = document.createElement("li");
ul.prepend(li);
```

# Problem: List of Items

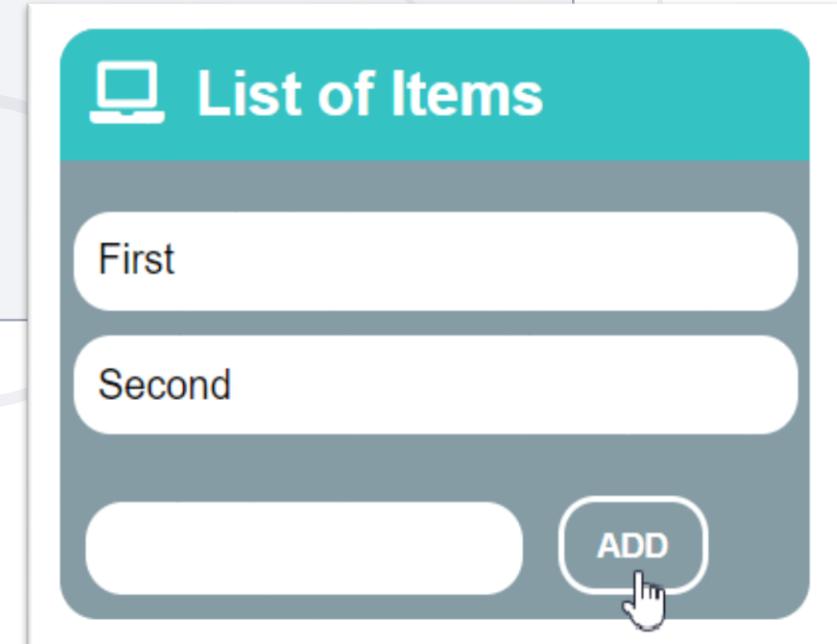
- Create a HTML page holding a **list of items** + **text box** + **button** for adding more items to the list
  - Write a function to **append** the specified text to the list



Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/4367#0>

# Problem: List of Items – HTML

```
<h1>List of Items</h1>
<ul id="items"><li>First</li><li>Second</li></ul>
<input type="text" id="newItemText" />
<input type="button" value="Add" onclick="addItem()">
<script>
function addItem() {
    // TODO: Add new item to the list
}
</script>
```



# Solution: List of Items

```
function addItem() {  
    let text = document.getElementById('newItemText').value;  
    let li = document.createElement("li");  
    li.appendChild(document.createTextNode(text));  
    document.getElementById("items").appendChild(li);  
    //Clearing the input:  
    document.getElementById('newItemText').value = '';  
}
```

# Deleting DOM Elements

```
<ul id="items">
  <li class="red">Red</li>
  <li class="blue">Blue</li>
</ul>
```

```
▼<body>
  ▼<ul id="items">
    <li class="red">Red</li>
    <li class="blue">Blue</li>
  </ul>
</body>
```

```
let redElements =
  document.querySelectorAll("#items li.red");
redElements.forEach(li => {
  li.parentNode.removeChild(li);
});
```

```
▼<body>
  ▼<ul id="items">
    <li class="blue">Blue</li>
  </ul>
</body>
```

# Problem: Delete from Table

```
<table border="1" id="customers">
<tr><th>Name</th><th>Email</th></tr>
<tr><td>Eve</td><td>eve@gmail.com</td></tr>
<tr><td>Nick</td><td>nick@yahoo.com</td></tr>
<tr><td>Didi</td><td>didi@didi.net</td></tr>
<tr><td>Tedy</td><td>tedy@tedy.com</td></tr>
</table>
Email: <input type="text" name="email" />
<button onclick="deleteByEmail()">Delete</button>
<div id="result" />
```

Name	Email
Eve	eve@gmail.com
Nick	nick@yahoo.com
Didi	didi@didi.net
Tedy	tedy@tedy.com

Email:  DELETE

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/4367#1>

# Solution: Delete from Table

```
function deleteByEmail() {  
    let email = document.getElementsByName("email")[0].value;  
    let secondColumn = document.querySelectorAll(  
        "#customers tr td:nth-child(2)");  
    for (let td of secondColumn)  
        if (td.textContent == email) {  
            let row = td.parentNode;  
            row.parentNode.removeChild(row);  
            document.getElementById('result').  
               .textContent = "Deleted.";  
            return;  
        }  
    document.getElementById('result').textContent = "Not found.";  
}
```

Name	Email
Nick	nick@yahoo.com
Didi	didi@didi.net
Tedy	tedy@tedy.com

Email:  DELETE

Deleted.



# The DOM Event

## Event Object and Types

# Event Object

- Calls its **associated function**
- Passes a **single argument** to the function - a **reference** to the event object
- Contains **properties** that describe the event
  - Which **element** triggered the event
  - Screen **coordinates** where it occurred
  - What is the **type** of the event
  - And more



# Event Types in DOM API

- **Mouse** events

- `click`
- `mouseover`
- `mouseout`
- `mousedown`
- `mouseup`

- **Keyboard** events

- `keydown`
- `Keypress`
- `keyup`

- **Touch** events

- `touchstart`
- `touchend`
- `touchmove`
- `touchcancel`

- **DOM / UI** events

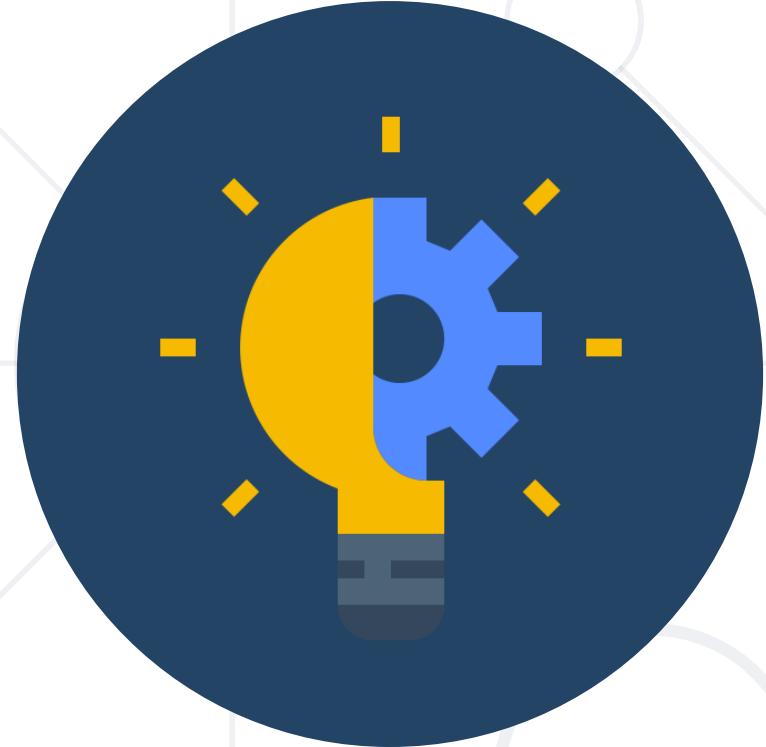
- `load`
- `unload`
- `resize`
- `dragstart / drop`

- **Focus** events

- `focus (got focus)`
- `blur (lost focus)`

- **Form** events

- `input`
- `change`
- `submit`
- `reset`



# Event Handling

# Event Handler

- Event registration is done by providing a **callback function**
- Three ways to register for an event:
  - With **HTML Attributes**
  - Using **DOM element properties**
  - Using **DOM event handler** – preferred method



```
function handler(event){  
    // this --> object, html reference  
    // event --> object, event configuration  
}
```

# Event Listener

- `addEventListener();`

```
htmlRef.addEventListener( 'click' , handler , false );
```

- `removeEventListener();`

```
htmlRef.removeEventListener( 'click' , handler );
```



# Attaching Click Handler

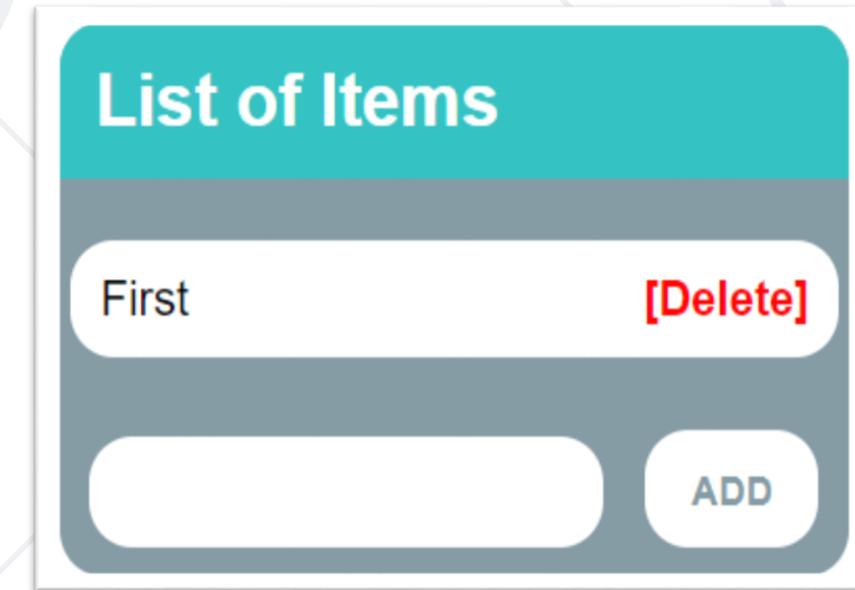
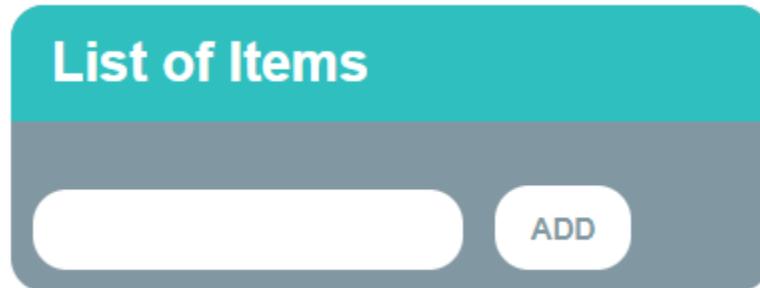
```
const button = document.getElementsByTagName('button')[0];  
  
button.addEventListener('click', clickMe);  
  
function clickMe(e) {  
    const target = e.currentTarget;  
    const targetText = target.textContent;  
    target.textContent = Number(targetText) + 1;  
}
```

Just click the button

0

# Problem: Add / Delete Items

- Extend the previous problem
  - Implement **[Delete]** action as link after each list item

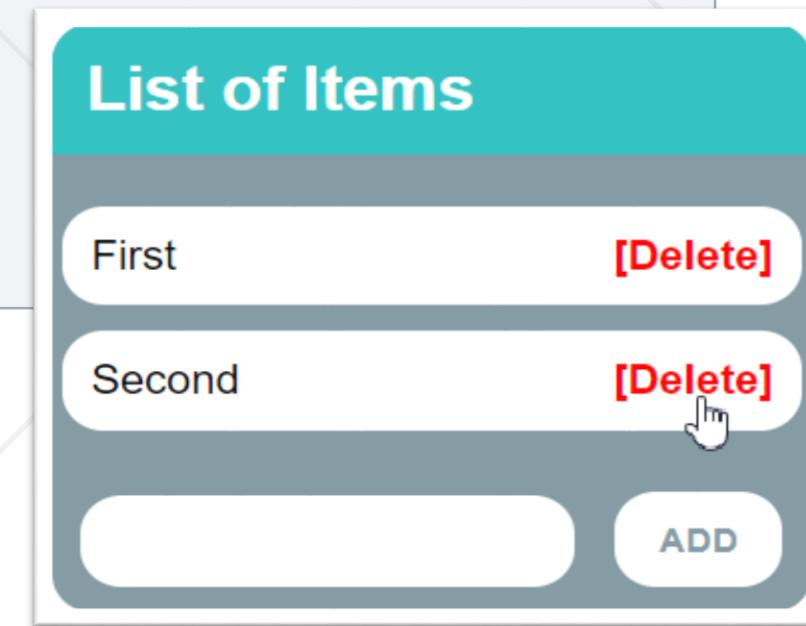


Check your solution here:

<https://judge.softuni.org/Contests/Practice/Index/4367#2>

# Problem: Add / Delete Items – HTML

```
<h1>List of Items</h1>
<ul id="items"></ul>
<input type="text" id="newText" />
<input type="button" value="Add" onclick="solve()">
<script>
function solve() {
    // TODO...
}
</script>
```



# Solution: Add / Delete Items

```
function solve() {  
    let newElement = document.getElementById("newText").value;  
    let list = document.getElementById("items");  
  
    if (newElement.length === 0) return;  
  
    let listItem = document.createElement("li");  
    listItem.textContent = newElement;  
  
    let remove = document.createElement("a");  
    let linkText = document.createTextNode("[Delete]");  
    // Continued on the next slide ...
```

# Solution: Add / Delete Items

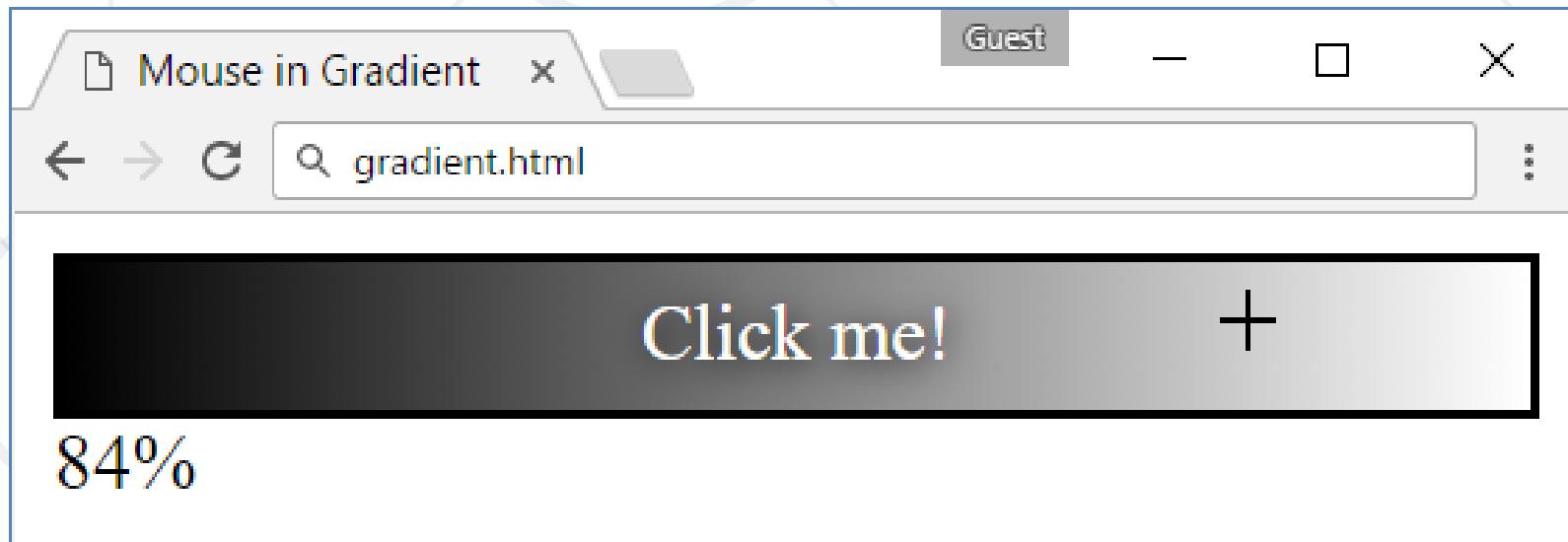
```
remove.appendChild(linkText);
remove.href = "#";
remove.addEventListener("click", deleteItem);

listItem.appendChild(remove);
list.appendChild(listItem);

function deleteItem() {
    listItem.remove();
}
```

# Problem: Mouse in Gradient

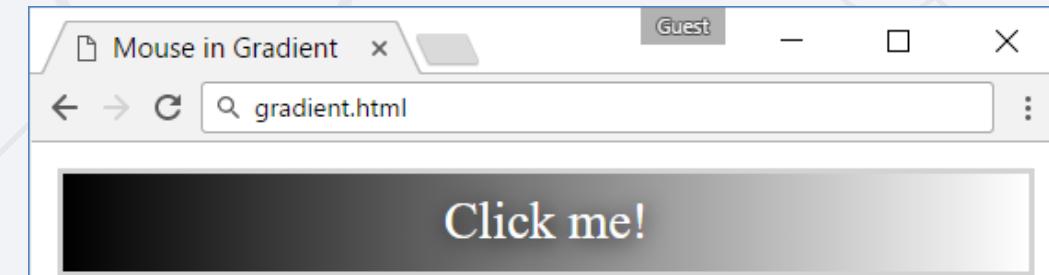
- A HTML page holds **linear gradient box**
  - Moving the mouse should show **percentage** [0% ... 100%], depending on the **location of mouse**
  - Left side → **0%**; middle → **50%**; right side → **100%**



Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/4367#3>

# Problem: Mouse in Gradient – HTML

```
<html>
<head>
  <title>Mouse in Gradient</title>
  <link rel="stylesheet" href="gradient.css" />
  <script src="gradient.js"></script>
</head>
<body onload="attachGradientEvents()">
  <div id="gradient-box">
    <div id="gradient">Click me!</div>
  </div>
  <div id="result"></div>
</body>
</html>
```



# Problem: Mouse in Gradient – CSS

```
#gradient-box {  
    width: 300px;  
    border: 2px solid lightgrey;  
}  
  
#gradient-box:hover {  
    border: 2px solid black;  
}  
  
#gradient {  
    height: 30px;  
    color: white;  
    text-shadow:  
        1px 1px 10px black;
```

```
        text-align: center;  
        line-height: 30px;  
        background:  
            linear-gradient(  
                to right, black, white);  
        cursor: crosshair;  
    }
```



# Solution: Mouse in Gradient

```
function attachGradientEvents() {  
    let gradient = document.getElementById('gradient');  
    gradient.addEventListener('mousemove', gradientMove);  
    gradient.addEventListener('mouseout', gradientOut);  
  
    function gradientMove(event) {  
        let power = event.offsetX / (event.target.clientWidth - 1);  
        power = Math.trunc(power * 100);  
        document.getElementById('result').textContent = power + "%";  
    }  
  
    function gradientOut(event) {  
        document.getElementById('result').textContent = "";  
    }  
};
```



# Live Demonstration

## Lab Problems 5 and 6

# Events Handler Execution Context

- In event handlers, **this** refers to the event **source element**

```
element.addEventListener("click", function(e) {  
  console.log(this === e.currentTarget); // Always true  
});
```

- Pay attention when using **object methods** as event listeners!
  - **this** may not behave as you expect with objects

# Attaching Hover Handler

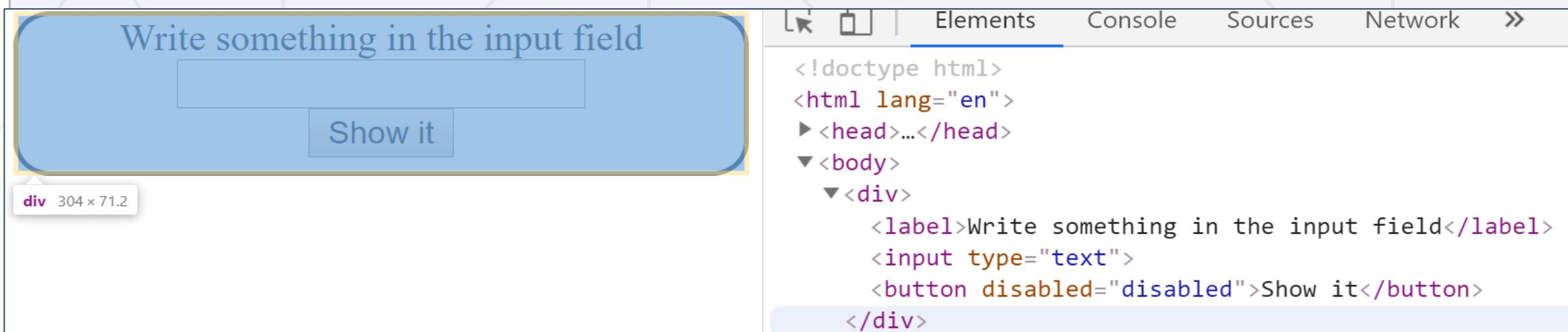
```
const button = document.getElementsByTagName('div')[0];
button.addEventListener('mouseover', function (e) {
  const style = e.currentTarget;
  const { backgroundColor } = style;

  if(backgroundColor === 'white'){
    targetStyles.backgroundColor = '#234465';
    targetStyles.color = 'white';
  } else {
    targetStyles.backgroundColor = 'white';
    targetStyles.color = '#234465';
  }
});
```

# Attaching Input Handler

```
const inputField = document.getElementsByTagName('input')[0];
const button = document.getElementsByTagName('button')[0];

inputField.addEventListener('input', function () {
  button.setAttribute('disabled', 'false')
});
```



The screenshot shows a browser's developer tools with the 'Elements' tab selected. The DOM tree on the right side of the tool panel displays the following structure:

```
<!doctype html>
<html lang="en">
  <head>...</head>
  <body>
    <div>
      <label>Write something in the input field</label>
      <input type="text">
      <button disabled="disabled">Show it</button>
    </div>
  </body>
</html>
```

To the left of the developer tools, there is a screenshot of a web page. The page has a blue background with white text. It features a label "Write something in the input field" above an input field, and a button labeled "Show it" below it. A tooltip for the input field indicates its dimensions: "div 304 x 71.2".

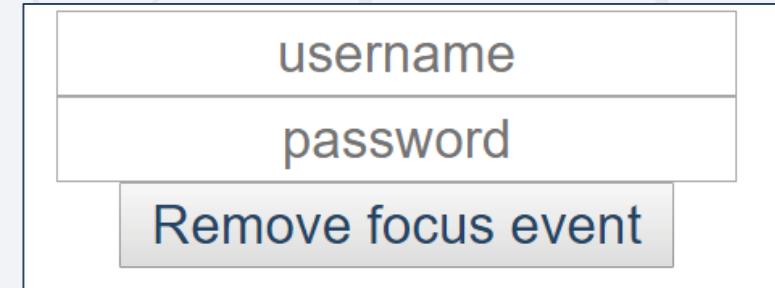
# Remove Listeners

```
const password = document.querySelector('input[type="password"]');
const button = document.querySelector('button');
password.addEventListener('focus', focusEvent);

function focusEvent (){
    event.target.style.background = '#234465';
}

password.addEventListener('blur', (event) => {
    event.target.style.background = '';
});

button.addEventListener('click', () => {
    password.removeEventListener('focus', focusEvent);
});
```



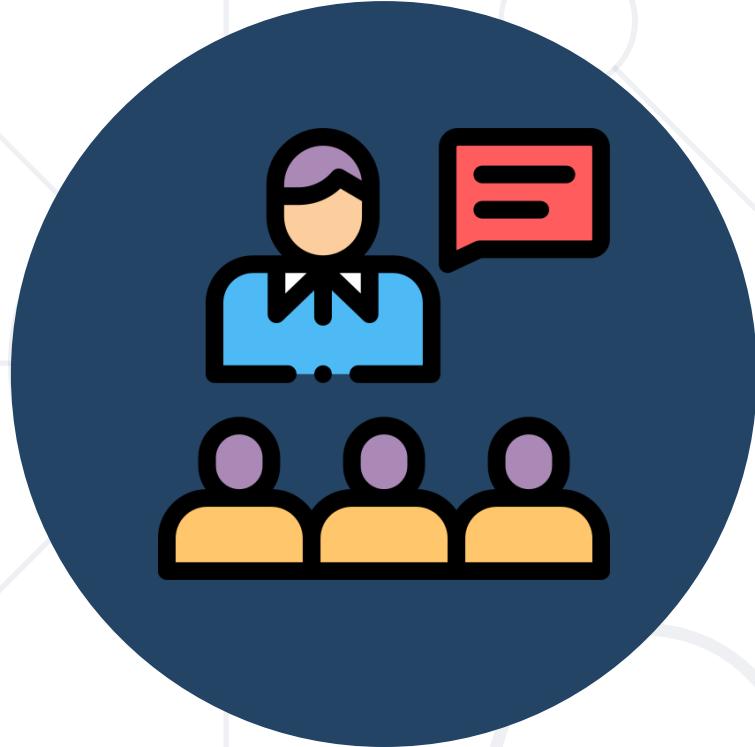
# Multiple Listeners

- The **addEventListener()** method also allows you to add many listeners to the same element, without overwriting existing ones:

```
element.addEventListener("click", myFirstFunction);
element.addEventListener("click", mySecondFunction);
element.addEventListener("mouseover", myThirdFunction);
element.addEventListener("mouseout", myFourthFunction);
```



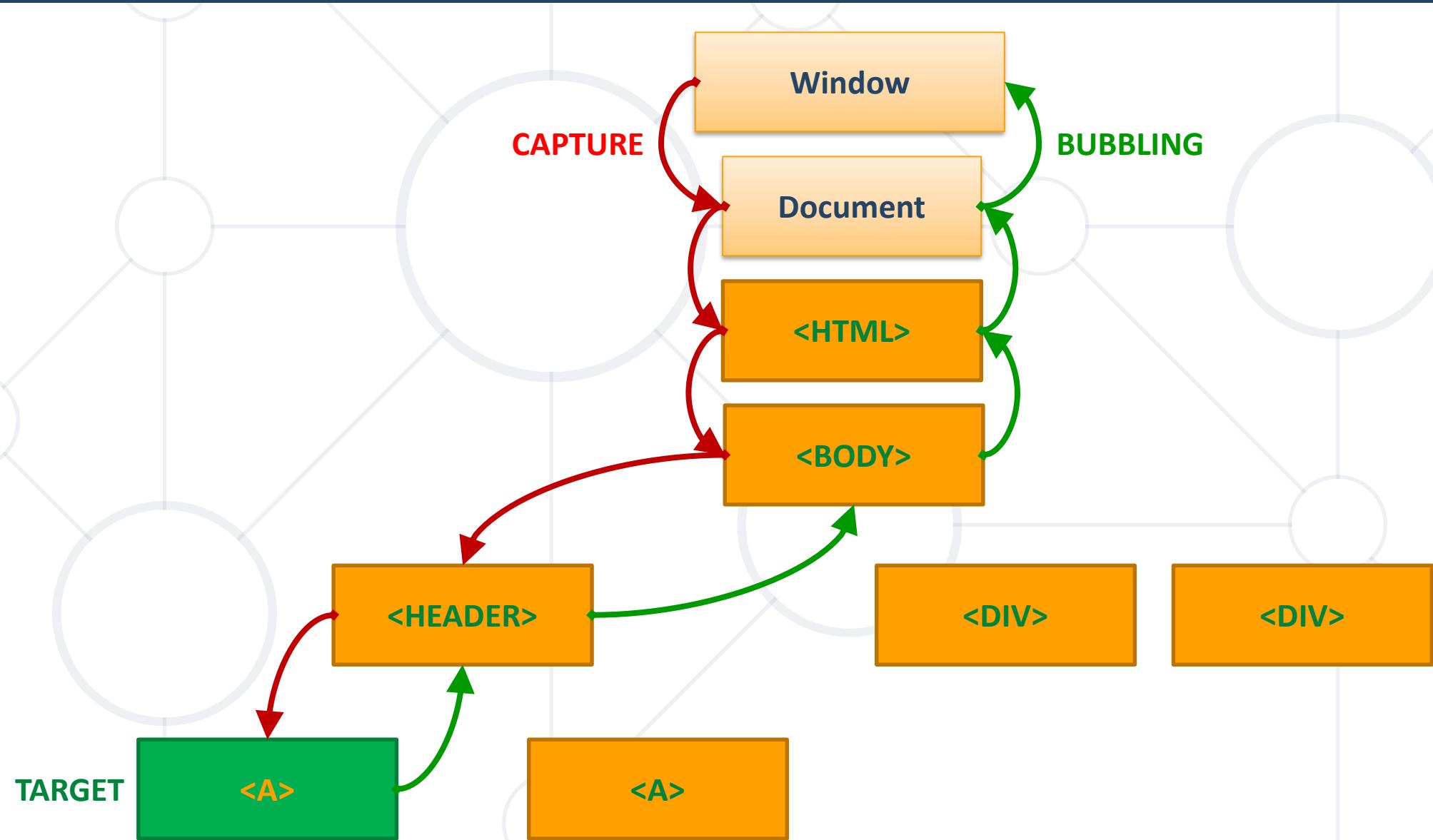
*Note that you don't use the "on" prefix for the event  
use "click" instead of "onclick"*



# Event Propagation

Handling Events Away From Their Source

# Event Propagation



# DOM Event Delegation

- Allows you to **avoid** adding event listeners to specific nodes
- Event listener is assigned to a **single ancestor**

```
<ul id="parent-list">
  <li id="post-1">Item 1</li>
  <li id="post-2">Item 2</li>
</ul>
```

```
document.getElementById("parent-list")
  .addEventListener("click", function(e) {
    if(e.target && e.target.nodeName == "LI") {
      console.log(
        "List item ", e.target.id.replace("post-", ""),
        " was clicked!");
    }
});
```

# Pros and Cons

## ■ Benefits

- Simplifies initialization
- Saves memory
- Less code

## ■ Limitations

- Event must be bubbling
- May add CPU load





# Live Demonstration

## Lab Problem 7

# Controlling Propagation and Behavior

- **stopPropagation** prevents further propagation of the event
  - If there are **multiple handlers** for the same event
- **preventDefault** stop the browser from executing default behavior, for example:
  - **Navigating** to a new page when <a> is clicked
  - Submitting **HTTP requests** via forms
  - Opening **context menus**

# Summary

- The DOM tree can be **manipulated** by:
  - **Creating** and **deleting** elements
  - **Moving** elements between nodes
- User interaction **triggers events**
  - They can be **listened** to and **handled**
  - The handler receives **event details**
  - Events **propagate** through the DOM tree



# Questions?



SoftUni



Software  
University



SoftUni  
Creative



SoftUni  
Digital



SoftUni  
Foundation



SoftUni  
Kids



Finance  
Academy

# SoftUni Diamond Partners



**SUPER  
HOSTING  
.BG**

**INDEAVR**  
Serving the high achievers

 **SOFTWARE  
GROUP**

 **PHAR  
VISION**



**Coca-Cola HBC  
Bulgaria**

 **AMBITIONED**



**BOSCH**

 **SmartIT**

 **DXC  
TECHNOLOGY**

 **Flutter  
International**™

 **DRAFT  
KINGS**

 **Postbank**  
*Решения за твоето утре*

 **createX**

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



# HTTP and REST Services

HTTP, Request Headers, RESTful Web Services



**REST API**

SoftUni Team

Technical Trainers



**SoftUni**



Software University

<https://softuni.bg>

# Table of Contents

1. HTTP Overview & Developer Tools
2. REST & RESTful Services
3. Accessing the GitHub API
4. NPM & package.json
5. CRUD Operations to a NodeJS REST API

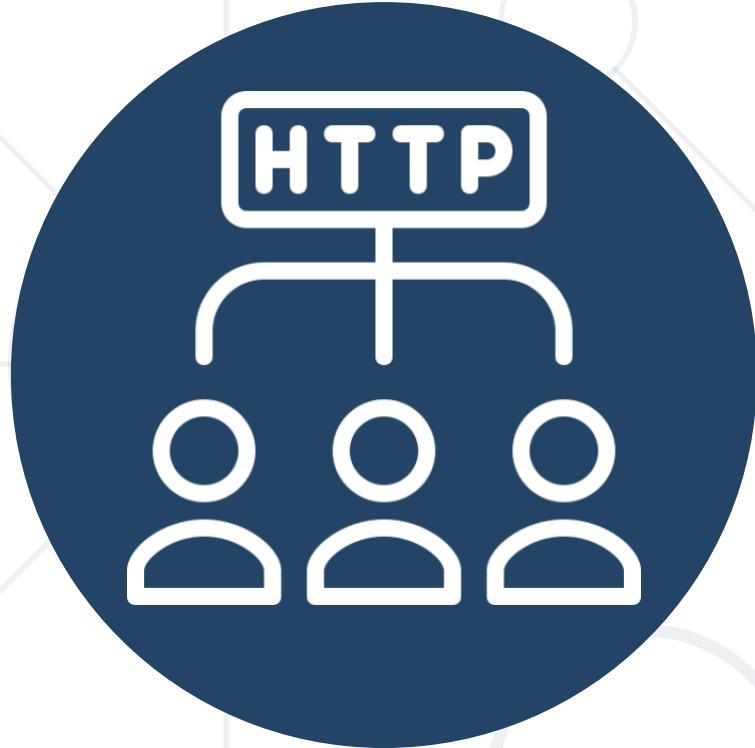


Have a Question?



**sli.do**

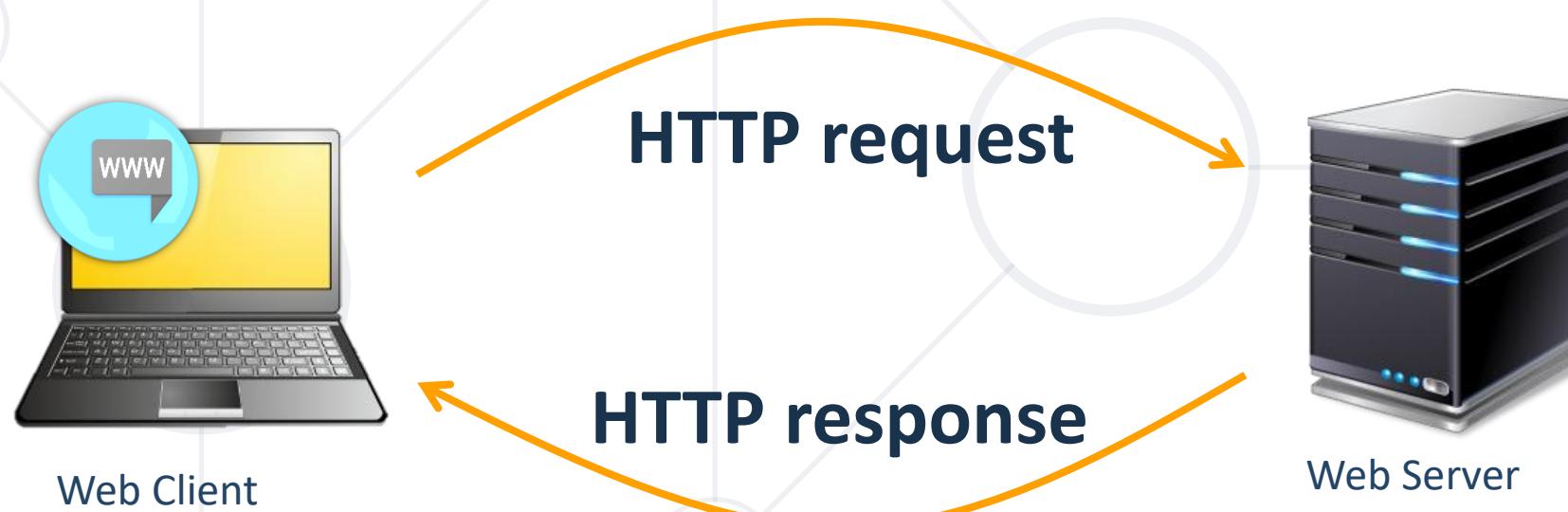
**#js-front-end**



# HTTP Overview

Hypertext Transfer Protocol

- HTTP (Hyper Text Transfer Protocol)
  - Text-based client-server protocol for the Internet
  - For transferring Web resources (HTML files, images, styles, etc.)
  - Request-response based



# HTTP Request Methods

- HTTP defines **methods** to indicate the desired action to be performed on the identified resource

Method	Description
GET	 Retrieve / load a resource
POST	 Create / store a resource
PUT	 Update a resource
DELETE	 Delete (remove) a resource
PATCH	 Update resource partially
HEAD	 Retrieve the resource's headers
OPTIONS	Returns the HTTP methods that the server supports for the specified URL

# HTTP GET Request – Example

GET /users/testnakov/repos HTTP/1.1

Host: api.github.com

Accept: \*/\*

Accept-Language: en

Accept-Encoding: gzip, deflate

User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.71 Safari/537.36

Connection: Keep-Alive

Cache-Control: no-cache

<CRLF>

HTTP request line

HTTP headers

The request body is empty

# HTTP POST Request – Example

POST /repos/testnakov/test-nakov-repo/issues HTTP/1.1

Host: api.github.com

Accept: \*/\*

Accept-Language: en

Accept-Encoding: gzip, deflate

User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)

Connection: Keep-Alive

Cache-Control: no-cache

<CRLF>

{"title": "Found a bug",  
 "body": "I'm having a problem with this.",  
 "labels": ["bug", "minor"]}

<CRLF>

HTTP headers

HTTP request line

The request body holds  
the submitted data

# HTTP Response – Example

HTTP/1.1 200 OK

HTTP response status line

Date: Fri, 11 Nov 2016 16:09:18 GMT+2

Server: Apache/2.2.14 (Linux)

Accept-Ranges: bytes

HTTP response  
headers

Content-Length: 84

Content-Type: text/html

<CRLF>

<html>

  <head><title>Test</title></head>

HTTP response body

  <body>Test HTML page.</body>

</html>

# HTTP Response Status Codes

Status Code	Action	Description
200	OK	Successfully retrieved resource
201	Created	A new resource was created
204	No Content	Request has nothing to return
301 / 302	Moved	Moved to another location (redirect)
400	Bad Request	Invalid request / syntax error
401 / 403	Unauthorized	Authentication failed / Access denied
404	Not Found	Invalid resource
409	Conflict	Conflict was detected, e.g. duplicated email
500 / 503	Server Error	Internal server error / Service unavailable

# Content-Type and Disposition

- The **Content-Type / Content-Disposition** headers specify how the HTTP request / response body should be processed

JSON-encoded data

Content-Type: **application/json**

UTF-8 encoded HTML page.  
Will be shown in the browser

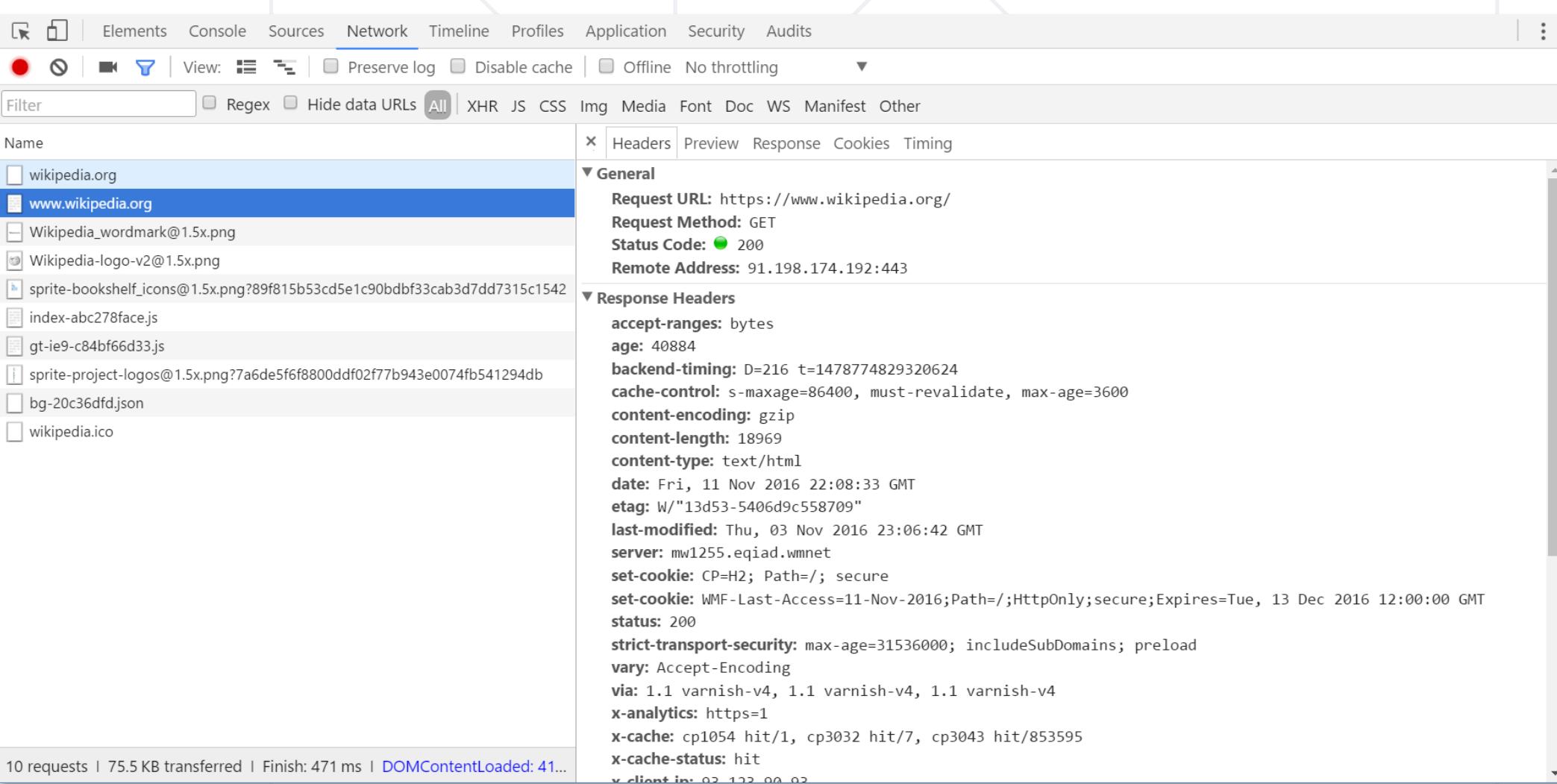
Content-Type: **text/html; charset=utf-8**

Content-Type: **application/pdf**

This will download a PDF file named  
Financial-Report-April-2016.pdf

Content-Disposition: attachment;  
filename="Financial-Report-April-2016.pdf"

# Browser Developer Tools



The screenshot shows the Network tab in the Chrome DevTools developer tools. A request for `https://www.wikipedia.org/` is selected in the list. The Headers panel is expanded, displaying the following details:

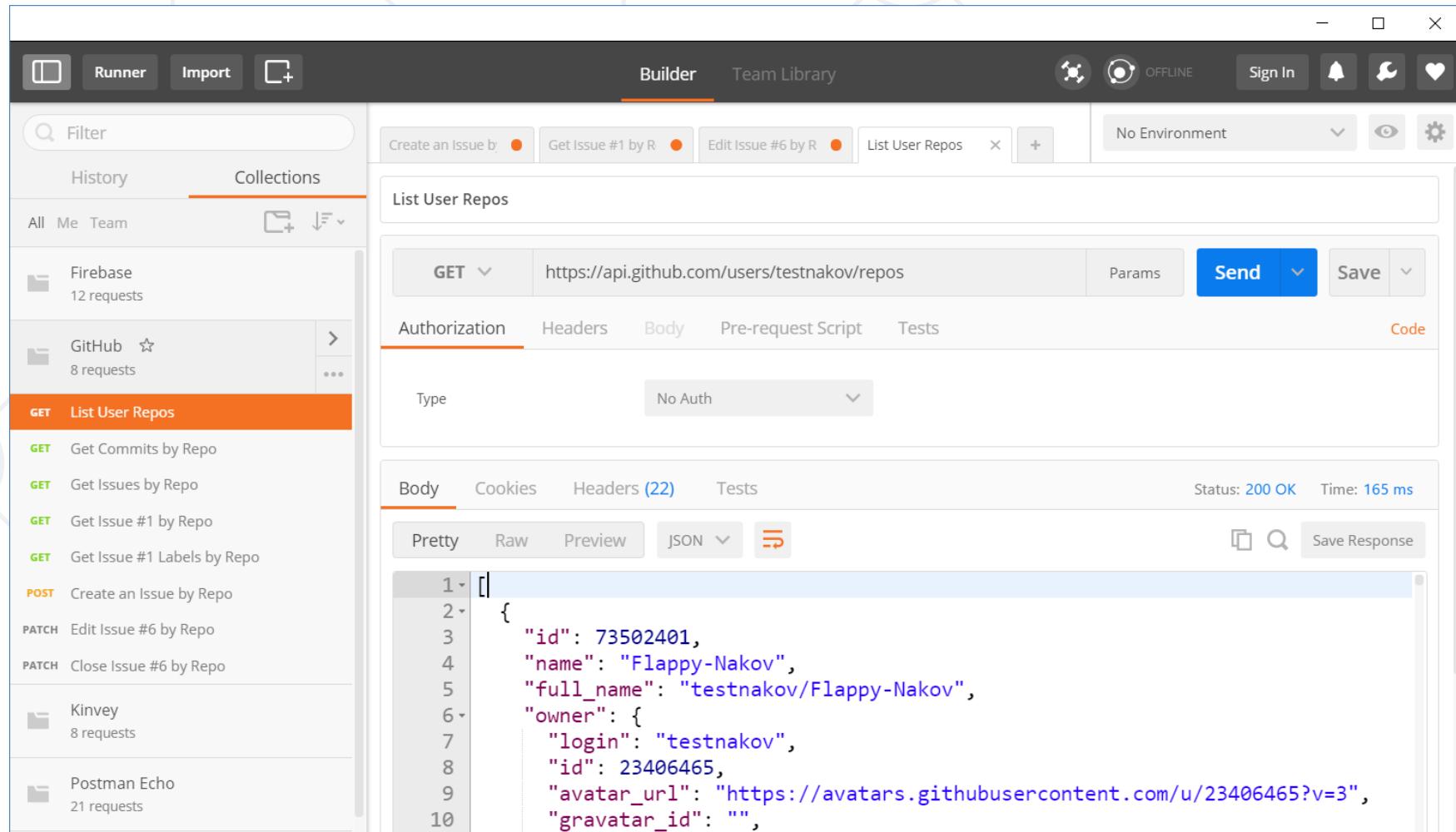
- Request URL:** `https://www.wikipedia.org/`
- Request Method:** GET
- Status Code:** 200
- Remote Address:** 91.198.174.192:443

The Response Headers section lists the following headers:

- accept-ranges: bytes
- age: 4084
- backend-timing: D=216 t=1478774829320624
- cache-control: s-maxage=86400, must-revalidate, max-age=3600
- content-encoding: gzip
- content-length: 18969
- content-type: text/html
- date: Fri, 11 Nov 2016 22:08:33 GMT
- etag: W/"13d53-5406d9c558709"
- last-modified: Thu, 03 Nov 2016 23:06:42 GMT
- server: mw1255.eqiad.wmnet
- set-cookie: CP=H2; Path=/; secure
- set-cookie: WMF-Last-Access=11-Nov-2016;Path=/;HttpOnly;secure;Expires=Tue, 13 Dec 2016 12:00:00 GMT
- status: 200
- strict-transport-security: max-age=31536000; includeSubDomains; preload
- vary: Accept-Encoding
- via: 1.1 varnish-v4, 1.1 varnish-v4, 1.1 varnish-v4
- x-analytics: https=1
- x-cache: cp1054 hit/1, cp3032 hit/7, cp3043 hit/853595
- x-cache-status: hit
- x-client-ip: 93.122.90.92

At the bottom of the Network tab, the status bar shows: 10 requests | 75.5 KB transferred | Finish: 471 ms | DOMContentLoaded: 41...

# Postman



[Read more about Postman REST Client](#)

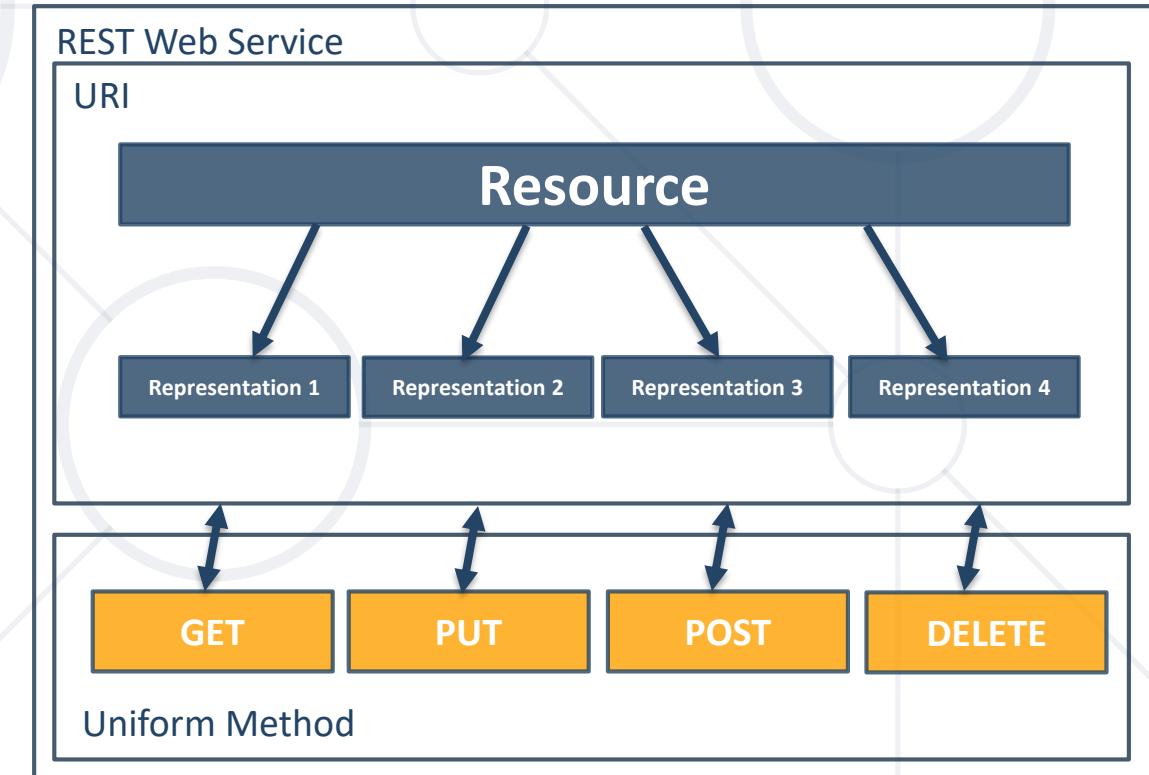


{REST}

**REST and RESTful Services**

# REST and RESTful Services

- **Representational State Transfer (REST)**
  - Architecture for **client-server communication** over HTTP
  - Resources have **URI** (address)
  - Can be **created/retrieved/modified/deleted/etc.**
- RESTful API/RESTful Service
  - Provides access to **server-side resources** via **HTTP** and **REST**



# REST and RESTful Services – Example

- Create a new post

<b>POST</b>	<a href="http://some-service.org/api/posts"><u>http://some-service.org/api/posts</u></a>
-------------	--

- Get all posts / specific post

<b>GET</b>	<a href="http://some-service.org/api/posts"><u>http://some-service.org/api/posts</u></a>
------------	--

<b>GET</b>	<a href="http://some-service.org/api/posts/17"><u>http://some-service.org/api/posts/17</u></a>
------------	--

- Delete existing post

<b>DELETE</b>	<a href="http://some-service.org/api/posts/17"><u>http://some-service.org/api/posts/17</u></a>
---------------	--

- Replace / modify existing post

<b>PUT/PATCH</b>	<a href="http://some-service.org/api/posts/17"><u>http://some-service.org/api/posts/17</u></a>
------------------	--



# Accessing GitHub Through HTTP

GitHub REST API

- List user's all public repositories:

GET

<https://api.github.com/users/testnakov/repos>

- Get all commits from a public repository:

GET

<https://api.github.com/repos/testnakov/softuniada-2016/commits>

- Get all issues/issue #1 from a public repository

GET

</repos/testnakov/test-nakov-repo/issues>

GET

</repos/testnakov/test-nakov-repo/issues/1>

# Github: Labels Issue

- Get the first issue from the "**test-nakov-repo**" repository
- Send a **GET** request to:
  - [https://api.github.com/repos/testnakov/test-nakov-repo/  
issues/:id](https://api.github.com/repos/testnakov/test-nakov-repo/issues/:id)
  - Where **:id** is the current issue



- Get all labels for certain issue from a public repository:

GET

<https://api.github.com/repos/testnakov/test-nakov-repo/issues/1/labels>

- Create a new issue to certain repository (with authentication)

POST

<https://api.github.com/repos/testnakov/test-nakov-repo/issues>

Headers

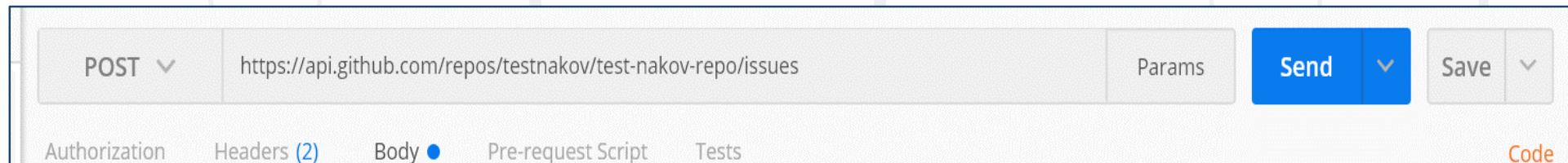
Authorization: Basic base64(user:pass)

Body

```
{"title": "Found a bug",
  "body": "I'm having a problem with this."}
```

# Github: Create Issue

- Create an issue when you send a "**POST**" request
- Use your Github account **credentials** to submit the issue





**NPM**

**Node Package Manager (NPM)**

# NPM: Package Management for JavaScript Projects

- **Node Package Manager (NPM)** is a package manager for JavaScript and Node.js projects
- Simplifies installing, managing, and sharing libraries and tools in web development
- Bundled with Node.js, it offers a command-line interface for various tasks
- Facilitates integrating third-party packages to reuse existing code and speed up development
- Helps manage project dependencies, ensuring required packages are available
- Allows defining and running scripts through the project's package.json file

# package.json

- **package.json** is a metadata file used in Node.js projects to provide information about the project, its dependencies, and various configurations
- It includes details such as the project's name, version, description, author, license, and more, making it a central place to document essential project information
- integral to the Node.js and NPM ecosystems. When you share your project, including the **package.json** file lets others quickly understand and recreate your project's environment. It also makes it easy for others to install the correct dependencies with a single command (**npm install**)

# Package.json – Example

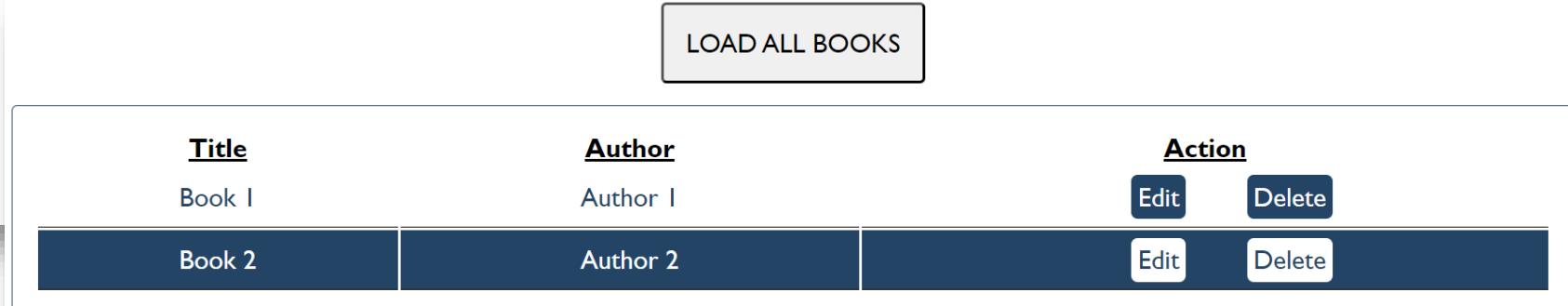
```
{  
  "name": "test",  
  "version": "1.0.0",  
  "description": "",  
  "main": "app.js",  
  "scripts": {  
    "start": "http-server -a localhost -p 3000 -P http://localhost:3000? -c-1"  
  },  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "http-server": "0.12.3"  
  }  
}
```

# Install Packages From The package.json

- **Navigate to Project:** Open your command-line interface and navigate to your project's directory
- Run the following command to install the project's dependencies listed in the **package.json**  
`npm install`
- NPM will read the **package.json** file and install all the dependencies

# Problem: Book Library

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <meta http-equiv="X-UA-Compatible" content="ie=edge">
7   <title>Books</title>
8   <link rel="stylesheet" href="./styles.css">
9 </head>
10 <body>
11   <button id="loadBooks">LOAD ALL BOOKS</button>
12   <table>
13     <thead>
14       <tr>
15         <th>Title</th>
16         <th>Author</th>
17         <th>Action</th>
```



A screenshot of a web application interface. At the top right is a button labeled "LOAD ALL BOOKS". Below it is a table with three columns: "Title", "Author", and "Action". The "Action" column contains two buttons each row: "Edit" and "Delete". The table has two rows. The first row contains the text "Book 1" under "Title" and "Author 1" under "Author". The second row contains "Book 2" under "Title" and "Author 2" under "Author".

Title	Author	Action
Book 1	Author 1	<button>Edit</button> <button>Delete</button>
Book 2	Author 2	<button>Edit</button> <button>Delete</button>

## FORM

### TITLE

### AUTHOR

Submit

# Summary

- **HTTP** is text-based request-response protocol
- **RESTful** services address resources by URL
- Provide **CRUD** operations over HTTP
- **RESTful** services address resources by URL
- **NPM** is used for installing, managing, and sharing libraries and tools



# Trainings @ Software University (SoftUni)



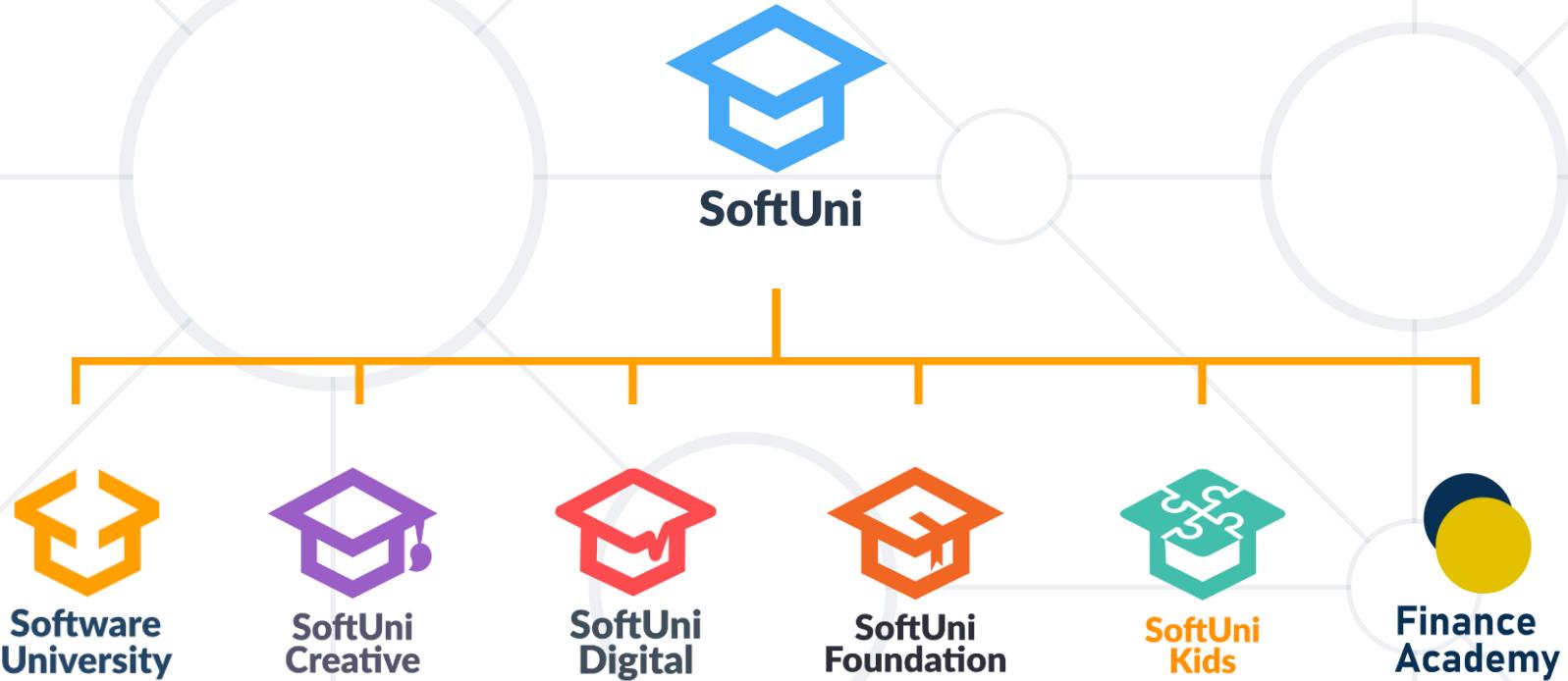
- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [softuni.org](http://softuni.org)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



# Questions?



# SoftUni Diamond Partners



**SUPER  
HOSTING  
.BG**

**INDEAVR**  
Serving the high achievers

 **SOFTWARE  
GROUP**

 **PHAR  
VISION**



**Coca-Cola HBC  
Bulgaria**

 **AMBITIONED**



**BOSCH**

 **SmartIT**

 **DXC  
TECHNOLOGY**

 **Flutter  
International**™

 **DRAFT  
KINGS**

 **Postbank**  
*Решения за твоето утре*

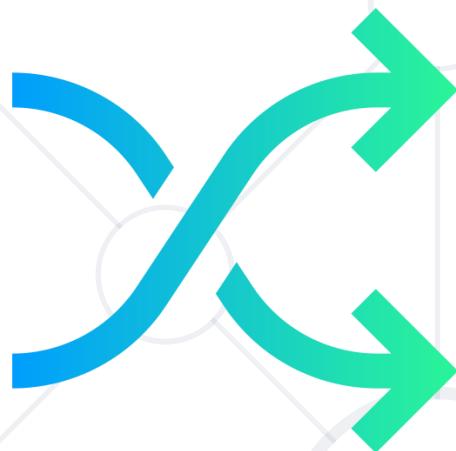
 **createX**

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://softuni.org>
- © Software University – <https://softuni.bg>



# Asynchronous Programming and Promises

Fetch API, Promises, async/await



SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.bg>

# Table of Contents

1. Asynchronous Programming
2. Promises Basics
3. AJAX & Fetch API
4. ES6 Async/Await

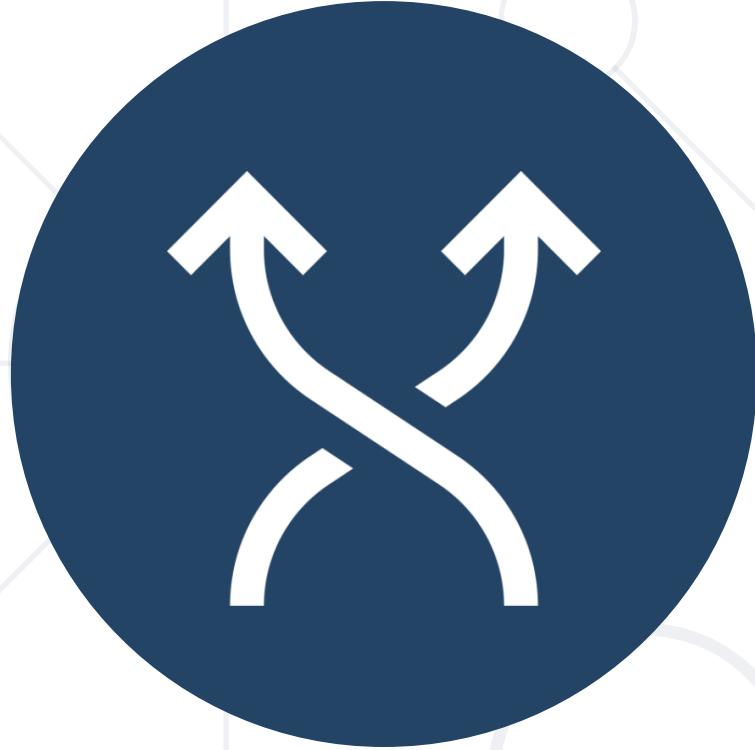


Have a Question?



**sli.do**

**#js-front-end**



# Synchronous vs Asynchronous

## Asynchronous Programming

# Asynchronous Programming in JS

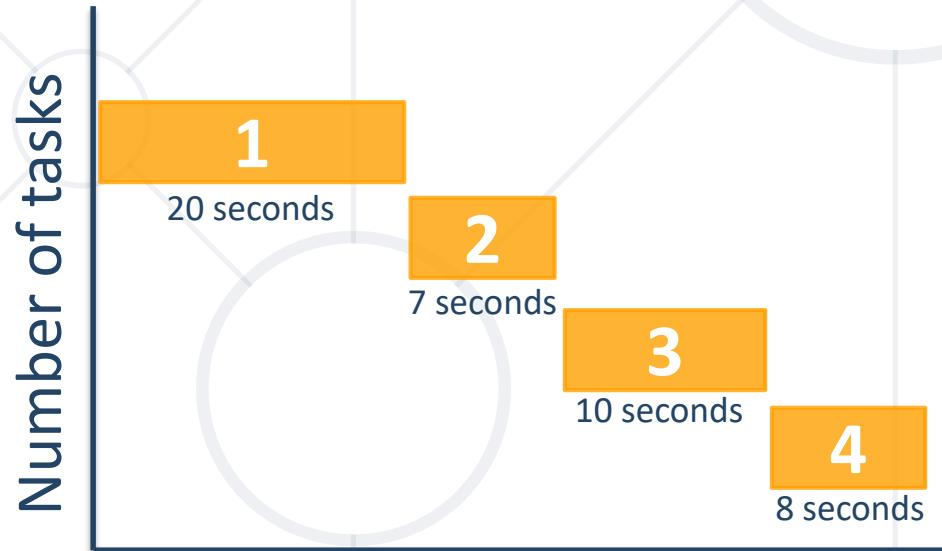
- Structured using **callback functions**
- In current versions of JS there are:
  - **Callbacks**
  - **Promises**
  - **Async Functions**
- Not the same thing as **concurrent** or **multi-threaded**
- **JS code** is generally **single-threaded**



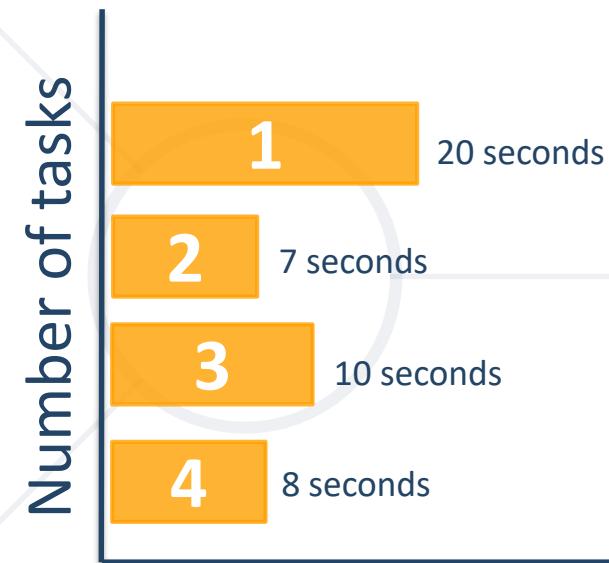
# Asynchronous Programming

- Runs several tasks (pieces of code) in parallel, **at the same time**

## Synchronous



## Asynchronous



# Asynchronous Programming – Example

- The following commands will be executed as follows:

```
console.log("Hello.");
setTimeout(function() {
    console.log("Goodbye!");
}, 2000);
console.log("Hello again!");
```

// Hello.

// Hello again!

// Goodbye!



# Callbacks

- Function **passed** into another function as an **argument**
- Then **invoked** inside the outer function to complete some kind of routine or action



```
function running() {  
    return "Running";  
}  
  
function category(run, type) {  
    console.log(run() + " " + type);  
}  
  
category(running, "sprint"); //Running sprint
```

Callback  
function



# Promises

Objects Holding Asynchronous Operations

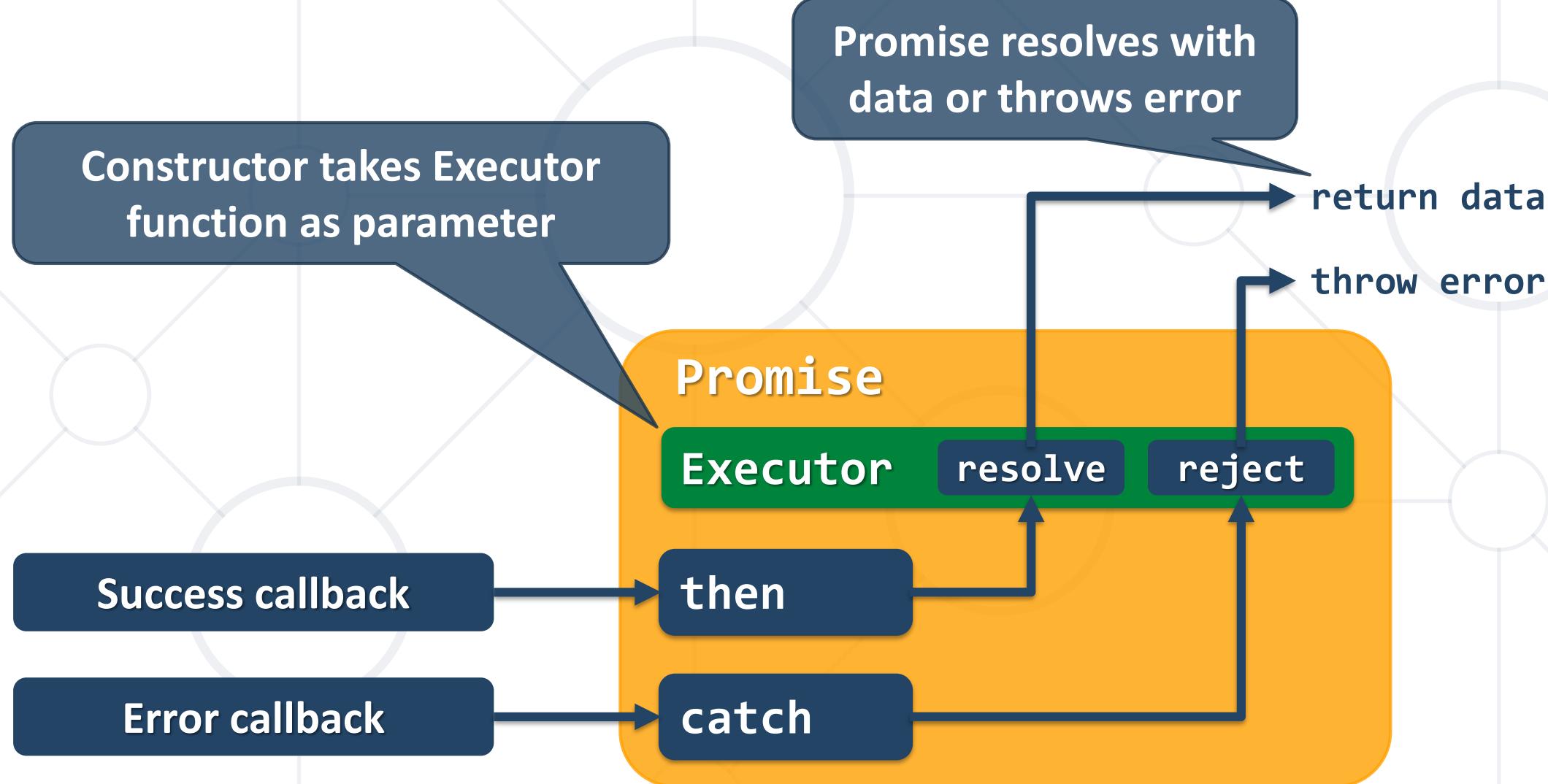
# What is a Promise?

- A promise is an **asynchronous action** that **may complete** at some point and **produce a value**
- States:
  - **Pending** - operation still running (unfinished)
  - **Fulfilled** - operation finished (the result is available)
  - **Failed** - operation failed (an error is present)
- Promises use the **Promise class**

```
new Promise(executor);
```



# Promise Flowchart



# Promise.then() – Example

```
console.log('Before promise');
```

```
new Promise(function(resolve, reject) {  
    setTimeout(function() {  
        resolve('done');  
    }, 500);  
})  
.then(function(res) {  
    console.log('Then returned: ' + res);  
});
```

Resolved after 500 ms

// Before promise

// After promise

// Then returned: done

```
console.log('After promise');
```

# Promise.catch() – Example

```
console.log('Before promise');
```

```
new Promise(function (resolve, reject) {  
    setTimeout(function () {  
        reject('fail');  
    }, 500);  
}).then (function (result) { console.log(result); })  
.catch (function(error) { console.log(error); });
```

Rejected after 500 ms

```
console.log('After promise');
```



# Popular Promise Methods

- **Promise.reject(reason)**
  - Returns an **object** that is **rejected** with the given **reason**
- **Promise.resolve(value)**
  - Returns an object that is **resolved** with the given **value**
- **Promise.finally()**
  - The handler is called when the promise is settled
- **Promise.all(iterable)**
  - Returns a **promise**
    - Fulfils when **all** of the promises **have fulfilled**
    - Rejects as soon as **one** of them **rejects**



AJAX

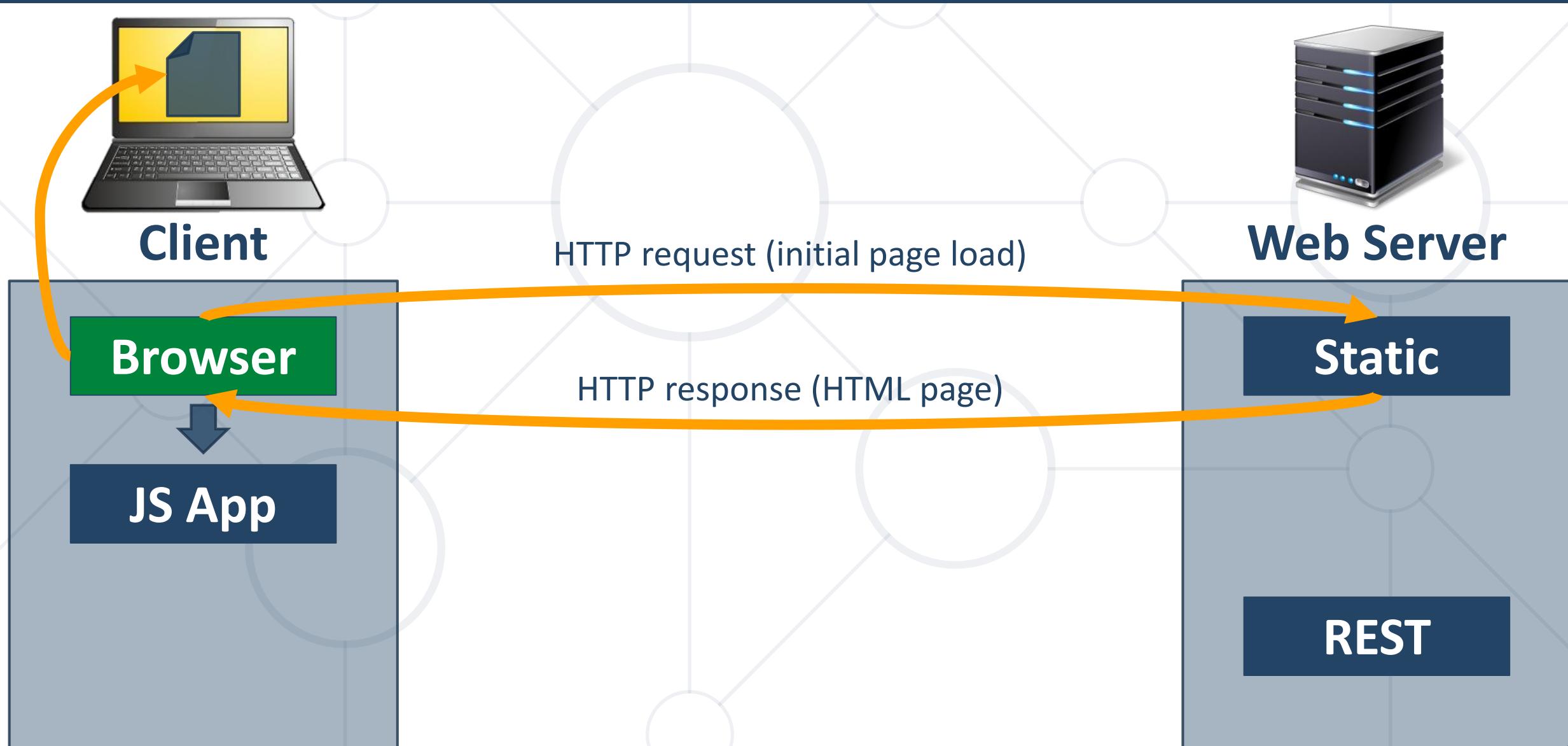
Connecting to a Server via Fetch API

# What is AJAX?

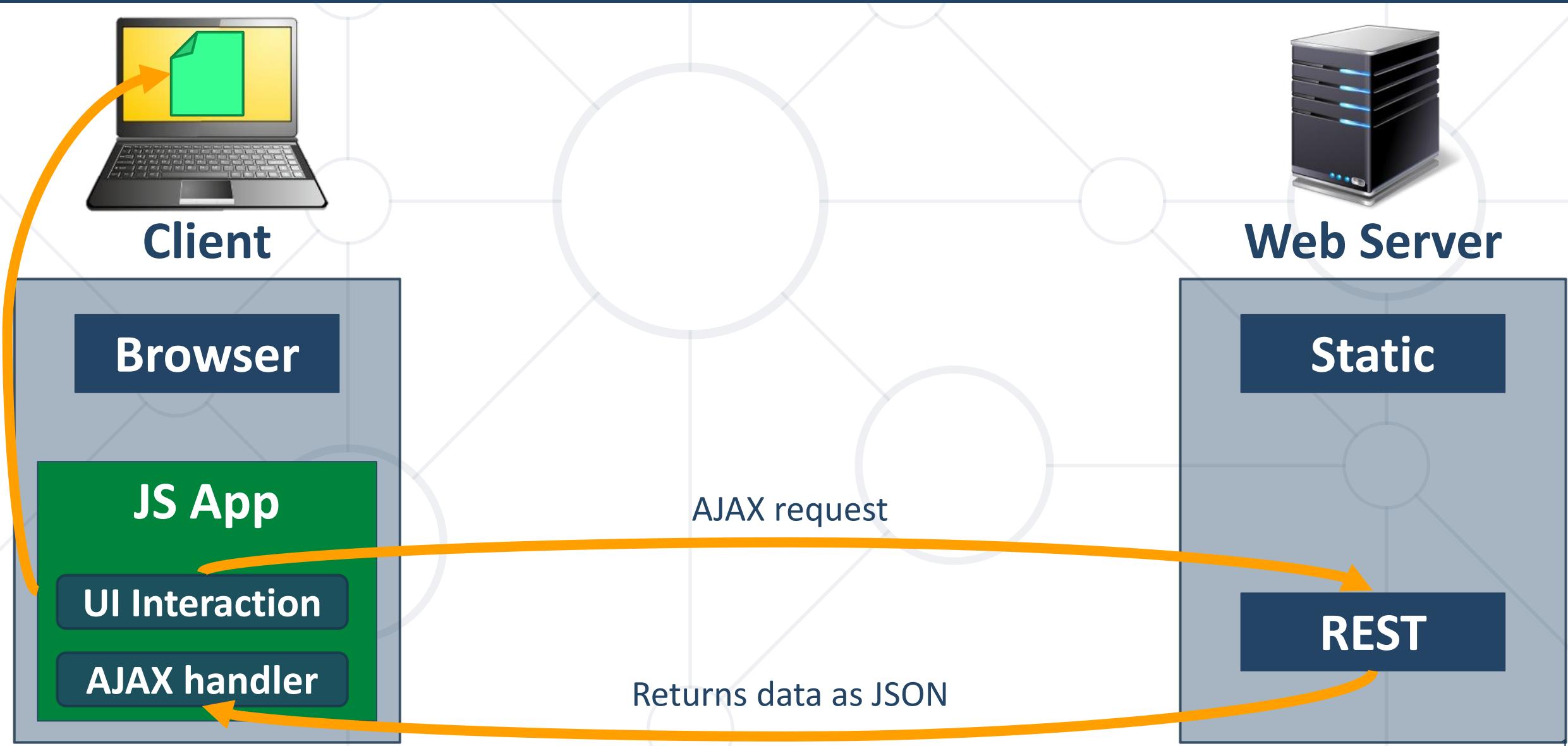
- **Asynchronous JavaScript And XML**
  - Background loading of **dynamic content/data**
  - Load data from the Web server and **render** it
- Some **examples** of AJAX usage:
  - **Partial page rendering**
    - Load HTML fragment + show it in a **<div>**
  - **JSON service**
    - Loads JSON object and displays it



# AJAX: Workflow



# AJAX: Workflow



# What is Fetch?

- The **Fetch API**:
  - Allows making network requests
  - Uses **Promises**
  - Enables a **simpler and cleaner** API
  - Makes code more readable and maintainable

```
fetch('./api/some.json')
  .then(function(response) {...})
  .catch(function(err) {...})
```



# Basic Fetch Request

- The response of a **fetch()** request is a **Stream** object
- The **reading** of the stream happens **asynchronously**
- When the **json()** method is called, a **Promise** is **returned**
  - The **response status** is checked (should be **200**) before **parsing** the response as **JSON**

```
if (response.status !== 200) {  
    // handle error  
}  
response.json()  
.then(function(data) { console.log(data)})
```

# GET Request

- Fetch API uses the GET method so that a direct call would be like this

```
fetch('https://api.github.com/users/testnakov/repos')  
  .then((response) => response.json())  
  .then((data) => console.log (data))  
  .catch((error) => console.error(error))
```



# Problem: GitHub Repos

- Execute an **AJAX GET** Request to load **all repos** of a **user**
- Use the **Fetch API**
- Use the following **URL**:
  - <https://api.github.com/users/testnakov/repos>
- In the **first then()** block map the response to **text**
- In the **second then()** block **display** the content in a **div**

# POST Request

- To make a **POST** request, we can set the **method** and **body** parameters in the **fetch()** options



```
fetch('/url', {  
  method: 'post',  
  headers: { 'Content-type': 'application/json' },  
  body: JSON.stringify(data),  
})
```

# PUT Request

```
fetch('/url/:id', {  
  method: 'put',  
  headers: { 'Content-type': 'application/json' },  
  body: JSON.stringify(data),  
})
```



# PATCH Request

```
fetch('/url/:id', {  
  method: 'patch',  
  headers: { 'Content-type': 'application/json' },  
  body: JSON.stringify(data),  
})
```



# DELETE Request

```
fetch('/url/:id', {  
  method: 'delete',  
})
```



# Problem: Load GitHub Commits

GitHub username:

```
<input type="text" id="username" value="nakov" /> <br>  
Repo: <input type="text" id="repo" value="nakov.io.cin" />
```

```
<button onclick="loadCommits()">Load Commits</button>
```

```
<ul id="commits"></ul>
```

```
<script>
```

```
function loadCommits() {
```

*// Use Fetch API*

```
}
```

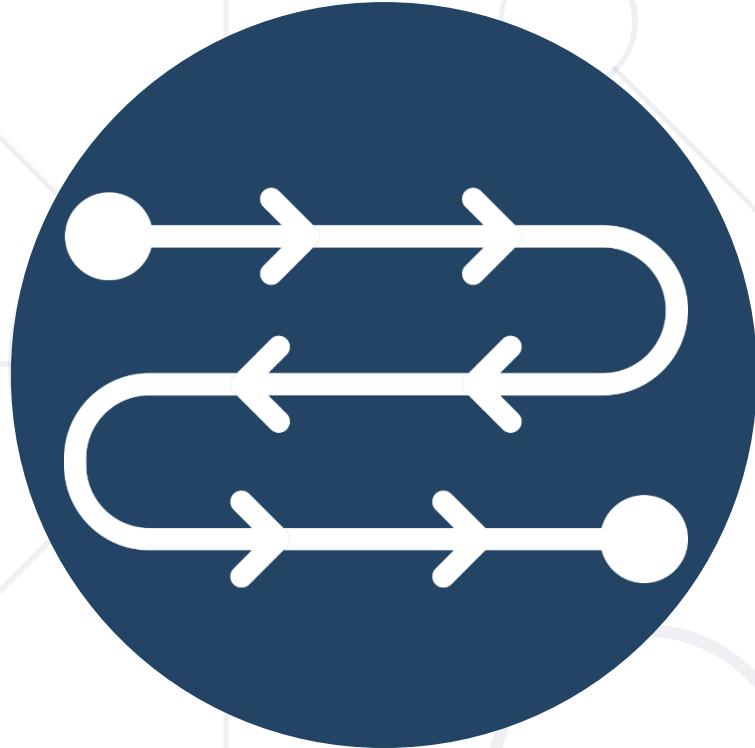
```
</script>
```

GitHub username: nakov

Repo: nakov.io.cin

Load Commits

- Svetlin Nakov: Delete Console.Cin.v11.suo
- Svetlin Nakov: Create LICENSE
- Svetlin Nakov: Update README.md
- Svetlin Nakov: Added better documentation

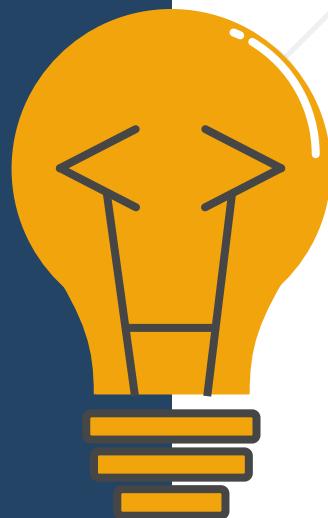


# Async / Await

## ES6 Simplified Promises

# Async Functions

- Returns a **promise**, that can await other promises in a way that **looks synchronous**
- Contains an **await** expression that:
  - Is **only valid** inside **async functions**
  - **Pauses** the execution of that function
  - Waits for the Promise's **resolution**



# Async Functions (2)



```
function resolveAfter2Seconds() {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve('resolved');  
    }, 2000);  
  });  
}
```

Expected output:  
`// calling`  
`// resolved`

```
async function asyncCall() {  
  console.log('calling');  
  let result = await resolveAfter2Seconds();  
  console.log(result);  
}
```

# Error Handling



```
async function f() {
  try {
    let response = await fetch();
    let user = await response.json();
  } catch (err) {
    // catches errors both in fetch and response.json
    alert(err);
  }
}
```

```
async function f() {
  let response = await fetch();
}
// f() becomes a rejected promise
f().catch(alert);
```

# Async/Await vs Promise.then

## Promise.then

```
function logFetch(url) {  
  return fetch(url)  
    .then(response => {  
      return response.text()  
    })  
    .then(text => {  
      console.log(text);  
    })  
    .catch(err => {  
      console.error(err);  
    });  
}
```

## Async/Await

```
async function logFetch(url) {  
  try {  
    const response =  
      await fetch(url);  
    console.log(  
      await response.text()  
    );  
  } catch (err) {  
    console.log(err);  
  }  
}
```



# Summary

- **HTTP** is text-based request-response protocol
- **RESTful** services address resources by URL
  - Provide **CRUD** operations over HTTP
- **Asynchronous** programming
- **Promises** hold operations – **resolve** & **reject**
- **AJAX** & **Fetch** API – connect to a server
- ES6 **Async/Await** Expression



# Trainings @ Software University (SoftUni)



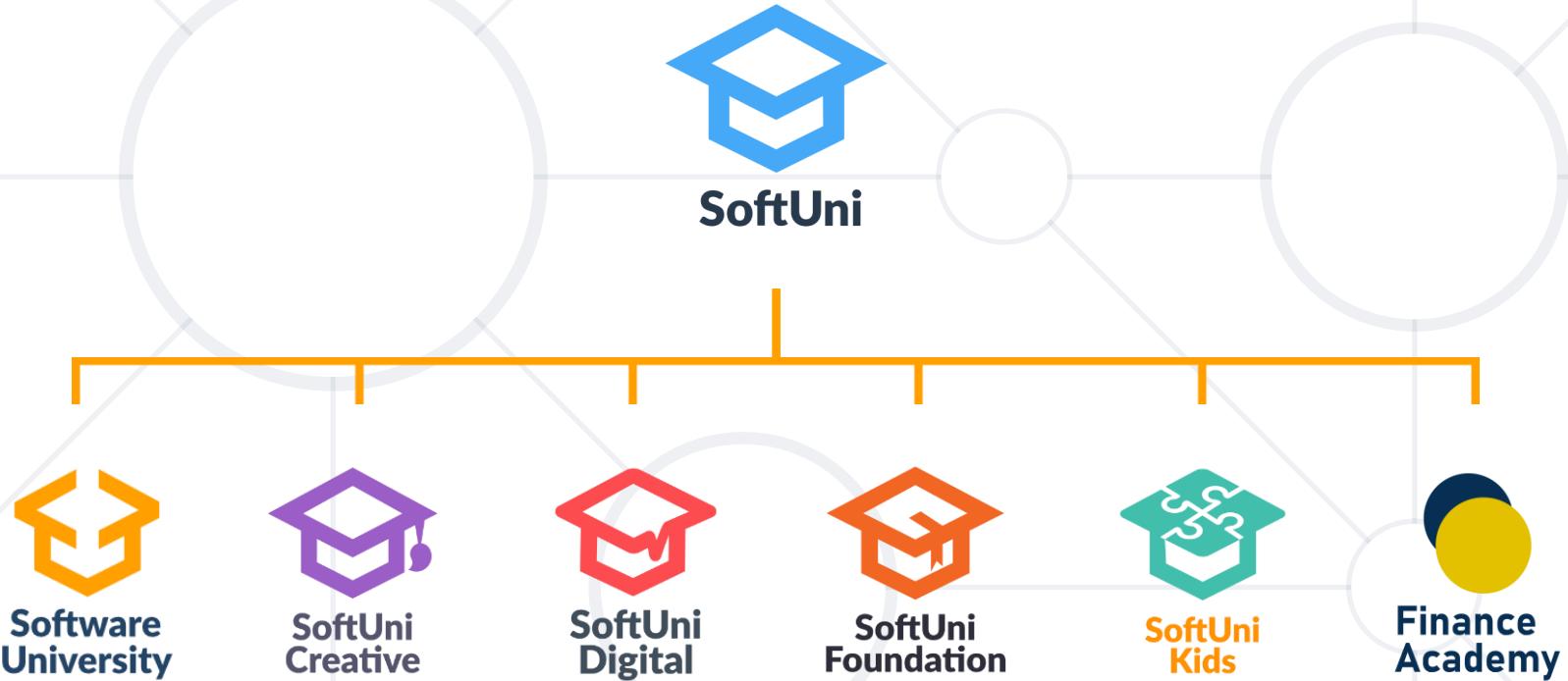
- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [softuni.org](http://softuni.org)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



# Questions?



# SoftUni Diamond Partners



**SUPER  
HOSTING  
.BG**

**INDEAVR**  
Serving the high achievers

 **SOFTWARE  
GROUP**

 **PHAR  
VISION**



**Coca-Cola HBC  
Bulgaria**

 **AMBITIONED**



**BOSCH**

 **SmartIT**

 **DXC  
TECHNOLOGY**

 **Flutter  
International**™

 **DRAFT  
KINGS**

 **Postbank**  
*Решения за твоето утре*

 **createX**

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://softuni.org>
- © Software University – <https://softuni.bg>

