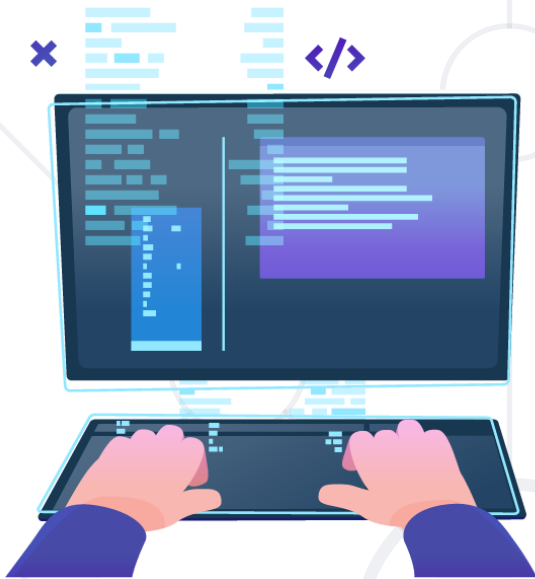


# Syntax, Functions and Statements

Operators, Parameters, Return Value, Arrow Functions



**SoftUni Team**

**Technical Trainers**



**SoftUni**



**Software University**

<https://softuni.bg>

# Table of Contents

1. Introduction to JavaScript
2. Data Types and Variables
3. Operators and Statements
4. Debugging Techniques
5. Functions & Hoisting



**sli.do**

**#js-advanced**



# 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**

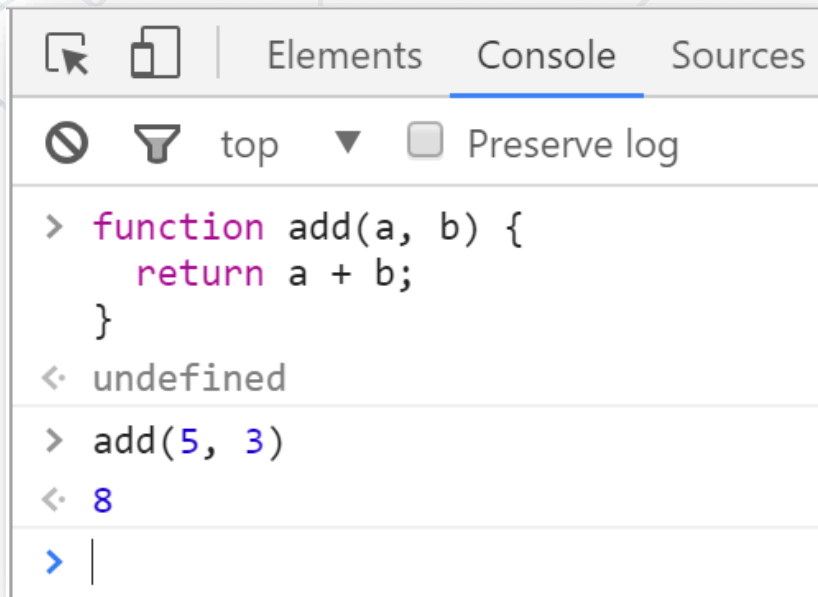
- JavaScript is a **dynamic programming language**
  - Operations otherwise done at **compile-time** can be done at **run-time**
- It is **possible** to change the **type** of a variable or add new properties or methods to an object **while** the program is **running**
- In **static programming languages**,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



# Live Demonstration

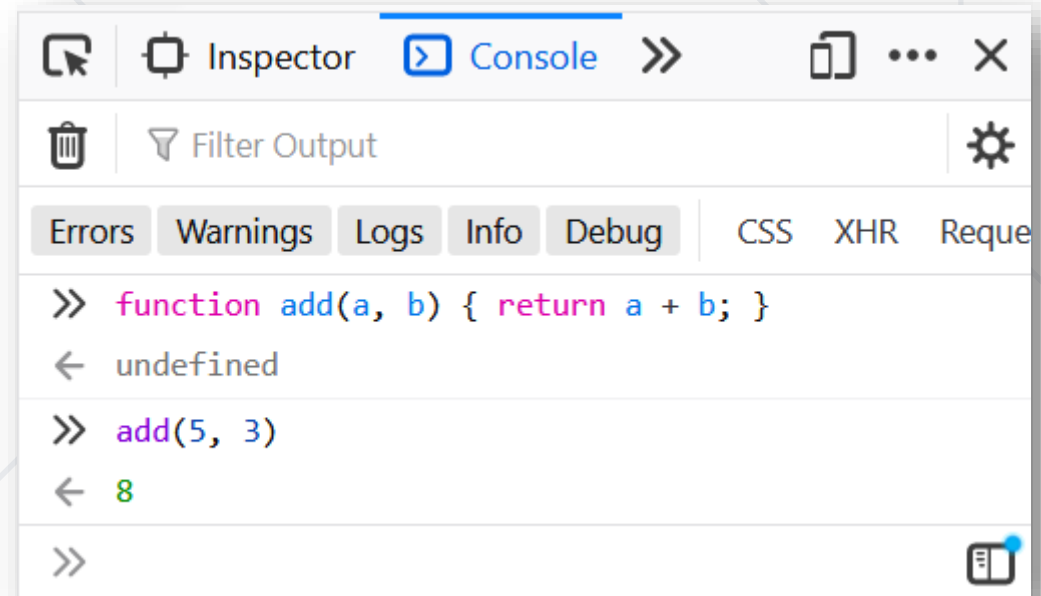
Setting up Node.js + VS Code

Developer Console: **[F12]**



The Chrome Developer Console interface is shown with the 'Console' tab selected. It features a toolbar with a close button, a filter icon, a dropdown menu set to 'top', and a 'Preserve log' checkbox. The console contains the following code and output:

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



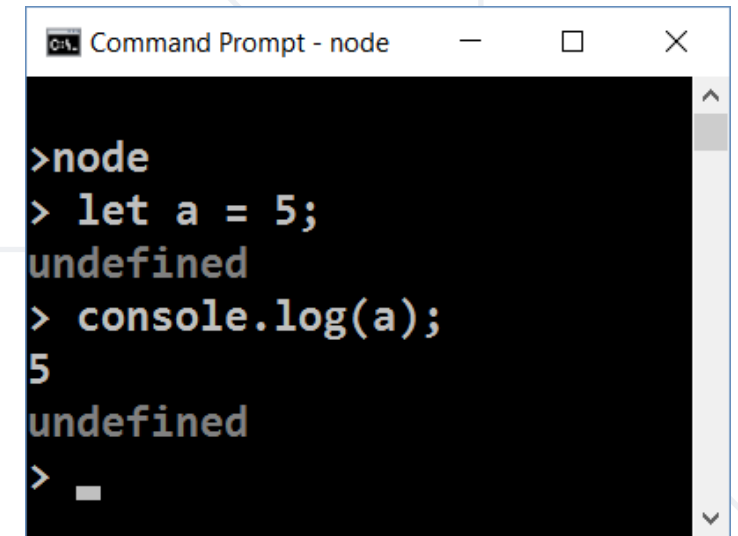
The Firefox Developer Console interface is shown with the 'Console' tab selected. It features a toolbar with a close button, a filter icon, and a 'Filter Output' dropdown menu. The console contains the following code and output:

```
>> 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



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


# Install the Latest Node.js


Latest LTS Version: 18.12.1 (includes npm 8.19.2)


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

  
Windows Installer  
node-v18.12.1-x64.msi

  
macOS Installer  
node-v18.12.1.pkg

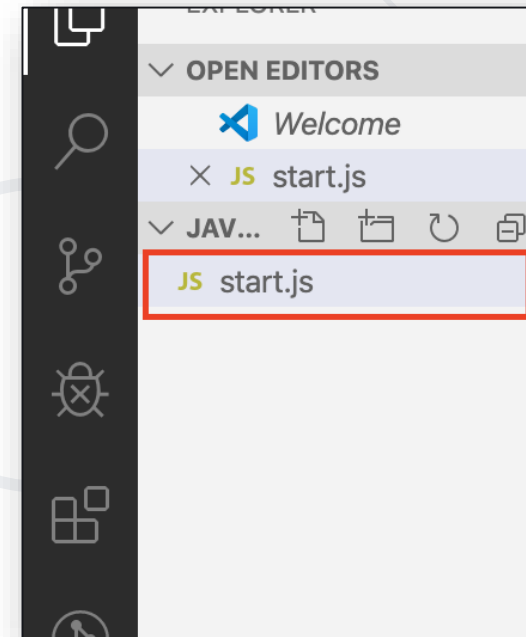
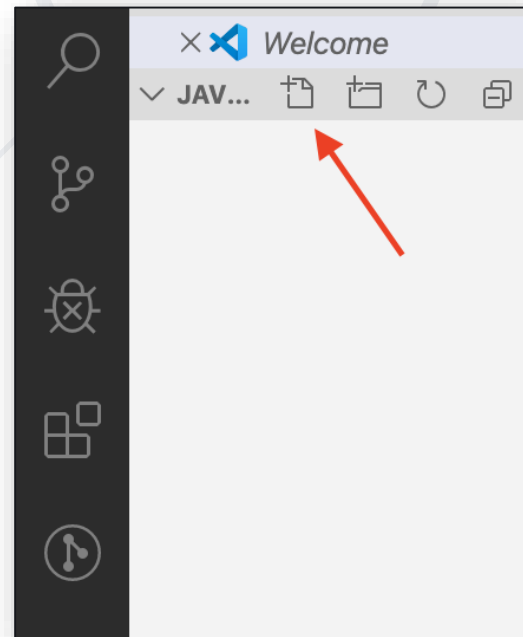
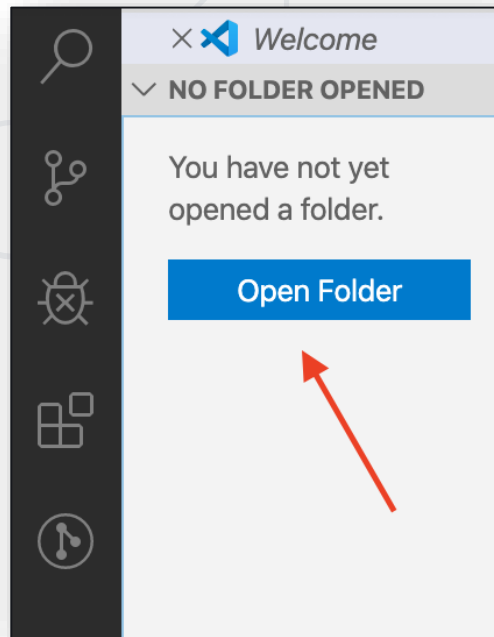
  
Source Code  
node-v18.12.1.tar.gz

- 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
32-bit	64-bit
64-bit / ARM64	
64-bit	ARM64
64-bit	
ARMv7	ARMv8
node-v18.12.1.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**:





# **Data Types & Variables**

Identifiers, Declaring Variables, Variable Scope

- Seven **data types** that are **primitives**
  - **String** - used to represent textual data
  - **Number** - a numeric data type
  - **Boolean** - a logical data type
  - **Undefined** - automatically assigned to variables
  - **Null** - represents the **intentional absence** of any object value
  - **BigInt** - represent integers with **arbitrary precision**
  - **Symbol** - **unique** and **immutable** primitive value
- **Reference types – Object**

- An **identifier** is a sequence of characters in the code that identifies a **variable**, **function**, or **property**
- In JavaScript, identifiers are **case-sensitive** and can contain Unicode **letters**, **\$**, **\_**, and **digits** (0-9), but may **not** start with a digit

```
let _name = "John";
```

```
function $sum(x, y) {  
    return x + y;  
}
```

```
let 9 = 'nine'; //SyntaxError: Unexpected number
```

# Variable Values

- Used to **store** data values
- Variables that are assigned a **non-primitive** value are given a **reference** to that value
- **Undefined** - a variable that has been declared with a keyword, but not given a value

```
let a;  
console.log(a) //undefined
```

- **Undeclared** - a **variable** that hasn't been declared at all

```
console.log(undeclaredVariable);  
//ReferenceError: undeclaredVariable is not defined
```



# Variable Values (2)

- **let**, **const** and **var** are used to declare variables
  - **let** - allows **reassignment**

```
let name = "George";  
name = "Maria";
```

- **const** - once assigned it **cannot** be modified

```
const name = "George";  
name = "Maria"; // TypeError
```

- **var** - defines a variable in the function scope **regardless** of block scope

```
var name = "George";  
name = "Maria";
```





# Legacy Variable Declaration

- You will see **var** used in old examples
- Using **var** to declare variables is a **legacy** technique
- Since **ES2015** keywords **let** and **const** are available
- **var** introduces function scope **hoisting**
  - Will be discussed later in the lesson
- There is no good reason to **ever** use **var**!




# Variable Scopes

- **Global scope** – Any variable that's **NOT** inside any **function** or **block** (a pair of curly braces);
- **Functional scope** – Variable declared **inside a function** is inside the local scope;
- **Block scope** – **let** and **const** declares **block** scoped variables



# Dynamic Typing

- 
- Variables in JavaScript are **not** directly **associated** with any particular **value type**
  - Any variable **can** be assigned (and re-assigned) values of all types

```
let foo = 42;      // foo is now a number  
foo = 'bar';      // foo is now a string  
foo = true;       // foo is now a boolean
```


- **NOTE: The use of dynamic typing is considered a bad practice!**



# Functions

Declaring and Invoking

# Functions

- 
- **Function** - named list of instructions (statements and expressions)
  - Can take **parameters** and return **result**
    - Function names and parameters use **camel case**
    - The **{** stays at the same line

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

- **Invoke** the function

```
printStars(10);
```

# Declaring Functions

- Function declaration

```
function walk() {  
  console.log("walking");  
}
```

- Function expression

```
const walk = function () {  
  console.log("walking");  
}
```


- Arrow functions

```
const walk = () => {  
  console.log("walking");  
}
```



# Parameters and Returned Value

- You can receive parameters with **no value**
- The **unused parameters** are ignored



```
function foo(a,b,c){  
  console.log(a);  
  console.log(b);  
  console.log(c); //undefined  
}  
foo(1,2)
```

```
function foo(a,b,c){  
  console.log(a);  
  console.log(b);  
  console.log(c);  
}  
foo(1,2,3,6,7)
```

- Functions can yield a value with the **return** operator

```
function identity(param){  
  return param;  
}  
console.log(identity(5)) // 5
```

# Object Methods and Standard Library

- Any object may have **methods**
  - **Functions** that operate from the **context** of the object
  - Accessed as a **property** using the **dot-notation**

```
let myString = 'Hello, JavaScript!';  
console.log(myString.toLowerCase());  
// hello, javascript!
```

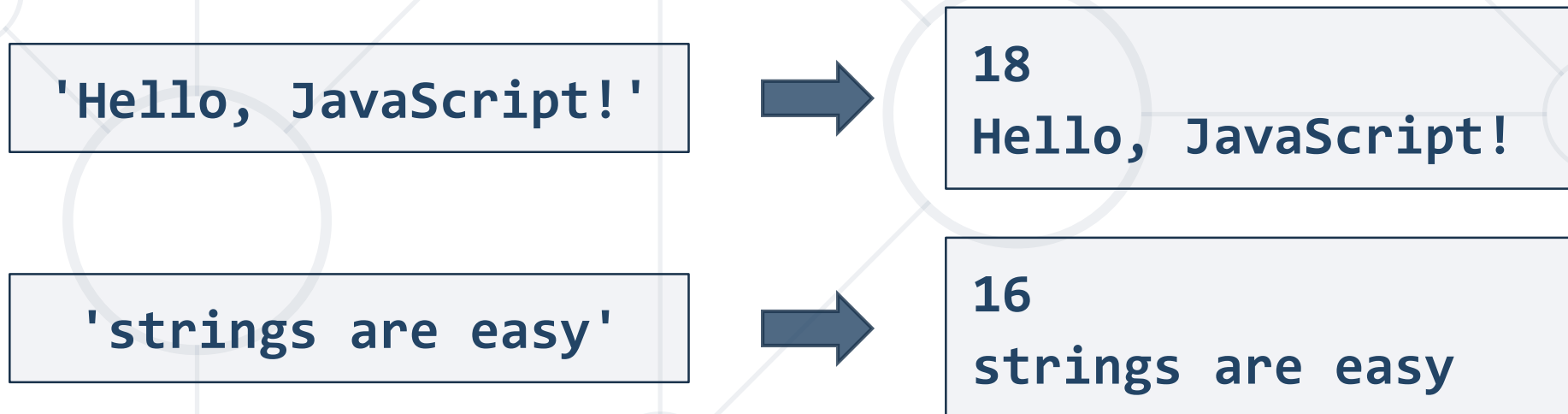
- JavaScript has a large **standard library**
  - **Math, Number, Date, RegExp, JSON** and more
  - For more information, [visit MDN](#)





# Problem: Echo Function

- A **string argument** is passed to your function
- **Print** on separate lines:
  - The **length** of the input parameter (number of characters)
  - The **unchanged parameter** itself



# Solution: Echo Function

```
function echo(inputAsString) {  
  let stringLength = inputAsString.length;  
  console.log(stringLength);  
  console.log(inputAsString);  
}
```

```
echo('Hello, JavaScript!');  
// 18  
// Hello, JavaScript!
```

# Default Function Parameter Values

- Functions can have **default parameter** values

```
function printStars(count = 5) {  
    console.log("*".repeat(count));  
}
```

```
printStars(); // *****
```

```
printStars(2); // **
```

```
printStars(3, 5, 8); // ***
```





# Operators and Statements

# Assignment, Arithmetic, Comparison, Logical

# 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 = 759375
```



# Assignment Operators

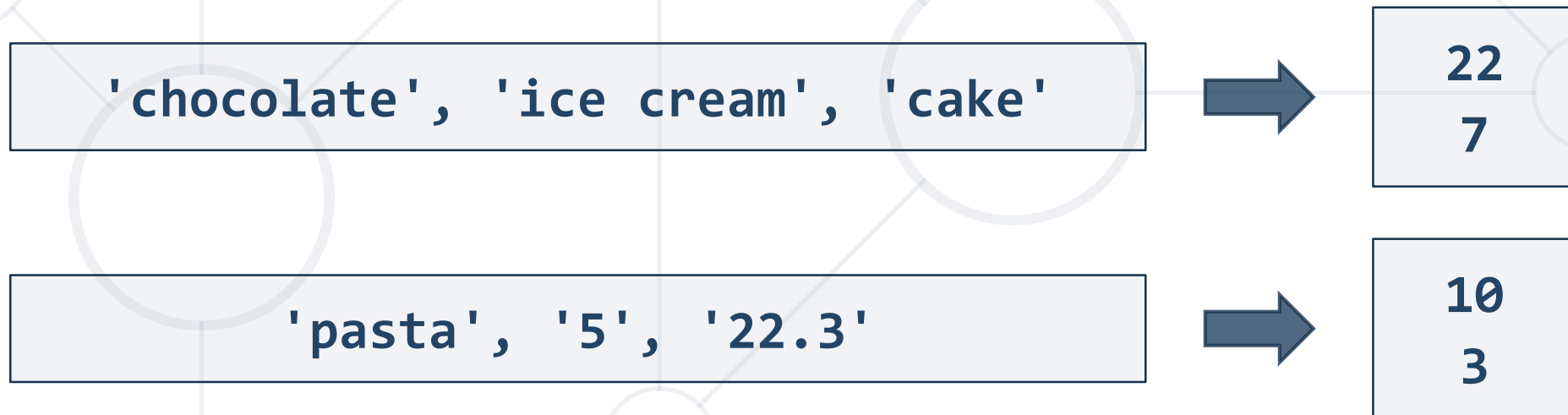
- **Assignment operators** - **assign** a value to its left operand based on the value of the right operand



Name	Shorthand operator	Basic usage
Assignment	$x = y$	$x = y$
Addition assignment	$x += y$	$x = x + y$
Subtraction assignment	$x -= y$	$x = x - y$
Multiplication assignment	$x *= y$	$x = x * y$
Division assignment	$x /= y$	$x = x / y$
Remainder assignment	$x \% = y$	$x = x \% y$
Exponentiation assignment	$x ** = y$	$x = x ** y$

# Problem: String Length

- Receive **three string arguments** as input
- Calculate the **total length** of all strings
- Calculate the **average length, rounded down**
- **Print** the result on the console



# Solution: String Length

```
function solve(str1, str2, str3) {  
  let len1 = str1.length;  
  let len2 = str2.length;  
  let len3 = str3.length;  
  
  let sumLength = len1 + len2 + len3;  
  let averageLength = Math.floor(sumLength / 3);  
  
  console.log(sumLength);  
  console.log(averageLength);  
}
```



# Comparison Operators



Operator	Notation in JS
EQUAL value	<code>==</code>
EQUAL value and type	<code>===</code>
NOT EQUAL value	<code>!=</code>
NOT EQUAL value or type	<code>!==</code>
GREATER than	<code>&gt;</code>
GREATER than OR EQUAL	<code>&gt;=</code>
LESS than	<code>&lt;</code>
LESS than OR EQUAL	<code>&lt;=</code>

# 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

# Conditional Statements

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**



# 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**, **0n** 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.
```

# Logical Operators

- **&& (logical AND)** - returns the leftmost **"false"** value or the **last truthy** value, if all are true.

```
const val = 'yes' && null && false
console.log(val); // null
const val1 = true && 5 && 'yes';
console.log(val1); // 'yes'
```

- **|| (logical OR)** - returns the leftmost **"true"** value or the **last falsy** value, if all are false.

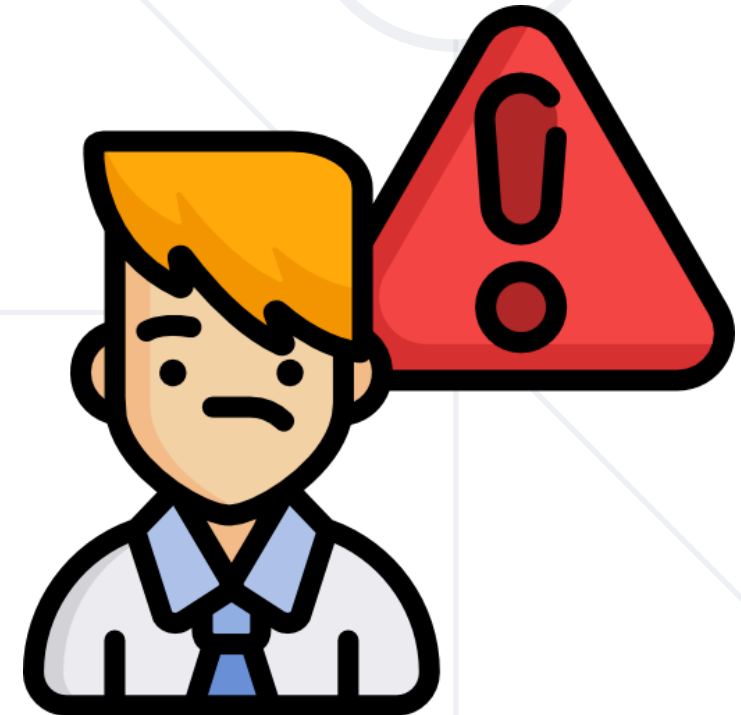
```
const val = false || '' || 5;
console.log(val); // 5
const val1 = null || NaN || undefined;
console.log(val1); // undefined
```



# Logical Operators (2)

- **!** (**logical NOT**) - Returns **false** if its single operand can be converted to **true**; otherwise, returns **true**

```
const val = !true  
console.log(val); // false  
const val = !false;  
console.log(val); // true
```



# Problem: Largest Number

- **Three number arguments** passed to your function as an input
- Find the **largest** of them
- **Print** the result on the **console**

3, 4, 5



The largest number is 5.

7, 11, 2



The largest number is 11.

- **Tip:** Use **interpolated strings** to format the result

```
`3 + 5 = ${3 + 5}` // 3 + 5 = 8
```

# Solution: Largest Number

```
function firstSolution(x, y, z) {  
  let result;  
  if (x >= y && x >= z) {  
    result = x;  
  } else if (y >= x && y >= z) {  
    result = y;  
  } else {  
    result = z;  
  }  
  console.log(`The largest number is ${result}.`);  
}
```

```
function secondSolution(...params) {  
  console.log(`The largest number is ${Math.max(...params)}.`);  
}
```



# 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
```

# Problem: Circle Area

- Write a function that takes a **single parameter** as an input
- Calculate the **area of a circle**, with the parameter as **radius**
  - If the parameter is **not a number**, print an **error** message
  - Include the **type** of parameter in the message
- **Print** the result on the console, **rounded** to the second decimal

5



78.54

'name'



We can not calculate the circle area, because we receive a **string**.

# Solution: Circle Area

```
function solve(radius) {  
  let inputType = typeof(radius);  
  
  if (inputType === 'number') {  
    let area = Math.pow(radius, 2) * Math.PI;  
    console.log(area.toFixed(2));  
  } else {  
    console.log(`We can not calculate the circle  
    area, because we receive a ${inputType}.`);  
  }  
}
```

# Some Interesting Examples

## ■ Data Types

```
console.log(typeof NaN);           //number
console.log(NaN === NaN);          //false
console.log(typeof null);          //object(Legacy reasons)
console.log(new Array() == false); //true
console.log(0.1 + 0.2);             //0.30000000000000004
console.log((0.2 * 10 + 0.1 * 10) / 10); //0.3
```

## ■ Truthy and Falsy values

```
const variable = [];               //empty array
console.log(variable == false);    //evaluates true
if (variable) { console.log('True!') }; // 'True!'
```

# Loops

- The **for** / **while** loops work as in C++, C# and Java
- Classical **for**-loop

```
for (let i = 0; i <= 5; i++) { console.log(i); }  
// 0 1 2 3 4 5
```

- JavaScript supports two more variants of the **for**-loop:
  - **for-of** – used with arrays and iterators
  - **for-in** – used with objects and associative arrays
  - Both will be reviewed in **upcoming lessons**





# Live Demonstration

Lab Problems 5 & 6

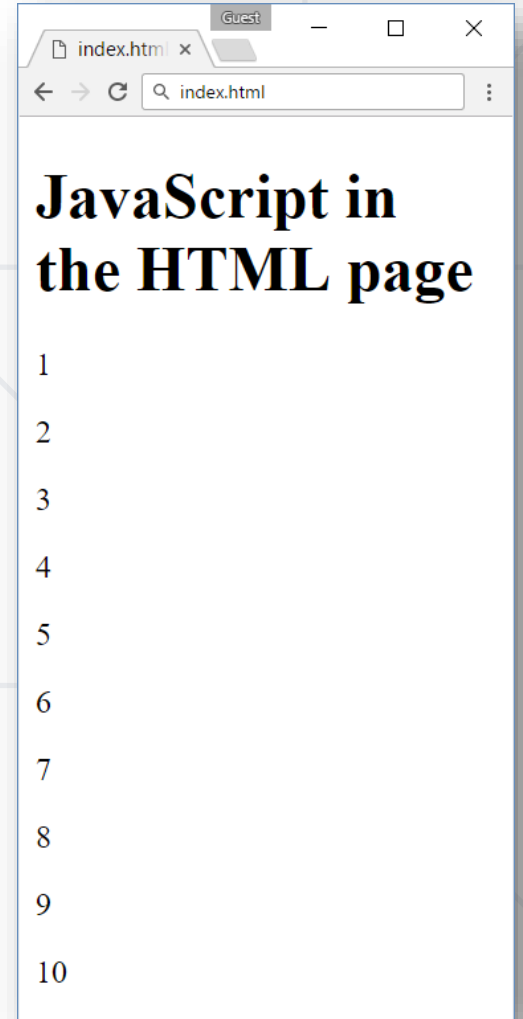


# Mix HTML and JavaScript

## Using JS Code from HTML Page

# Mixing HTML + JavaScript

```
<!DOCTYPE html>
<html>
<body>
  <h1>JavaScript in the HTML page</h1>
  <script>
    for (let i=1; i<=10; i++) {
      document.write(`<p>${i}</p>`);
    }
  </script>
</body>
</html>
```





# Sum Numbers with HTML Form

```
<form>
  num1: <input type="text" name="num1" /> <br>
  num2: <input type="text" name="num2" /> <br>
  sum: <input type="text" name="sum" /> <br>
  <input type="button" value="Sum" onclick="calcSum()" />
</form>
```

```
function calcSum() {
  let num1 = document.getElementsByName('num1')[0].value;
  let num2 = document.getElementsByName('num2')[0].value;
  let sum = Number(num1) + Number(num2);
  document.getElementsByName('sum')[0].value = sum;
}
```

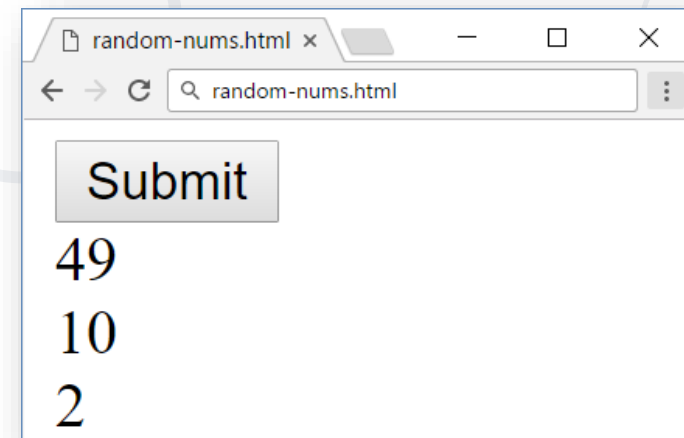
# Load JavaScript File from HTML Document

## random-nums.html

```
<!DOCTYPE html>
<html>
<head>
  <script src="numbers.js">
  </script>
</head>
<body>
  <input type="submit"
onclick="printRandNum()" />
</body>
</html>
```

## numbers.js

```
function printRandNum() {
  let num = Math.round(
    Math.random() * 100);
  document.body.innerHTML +=
    `<div>${num}</div>`;
}
```





# Debugging Techniques

Strict Mode, IDE Debugging Tools

# Strict Mode

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



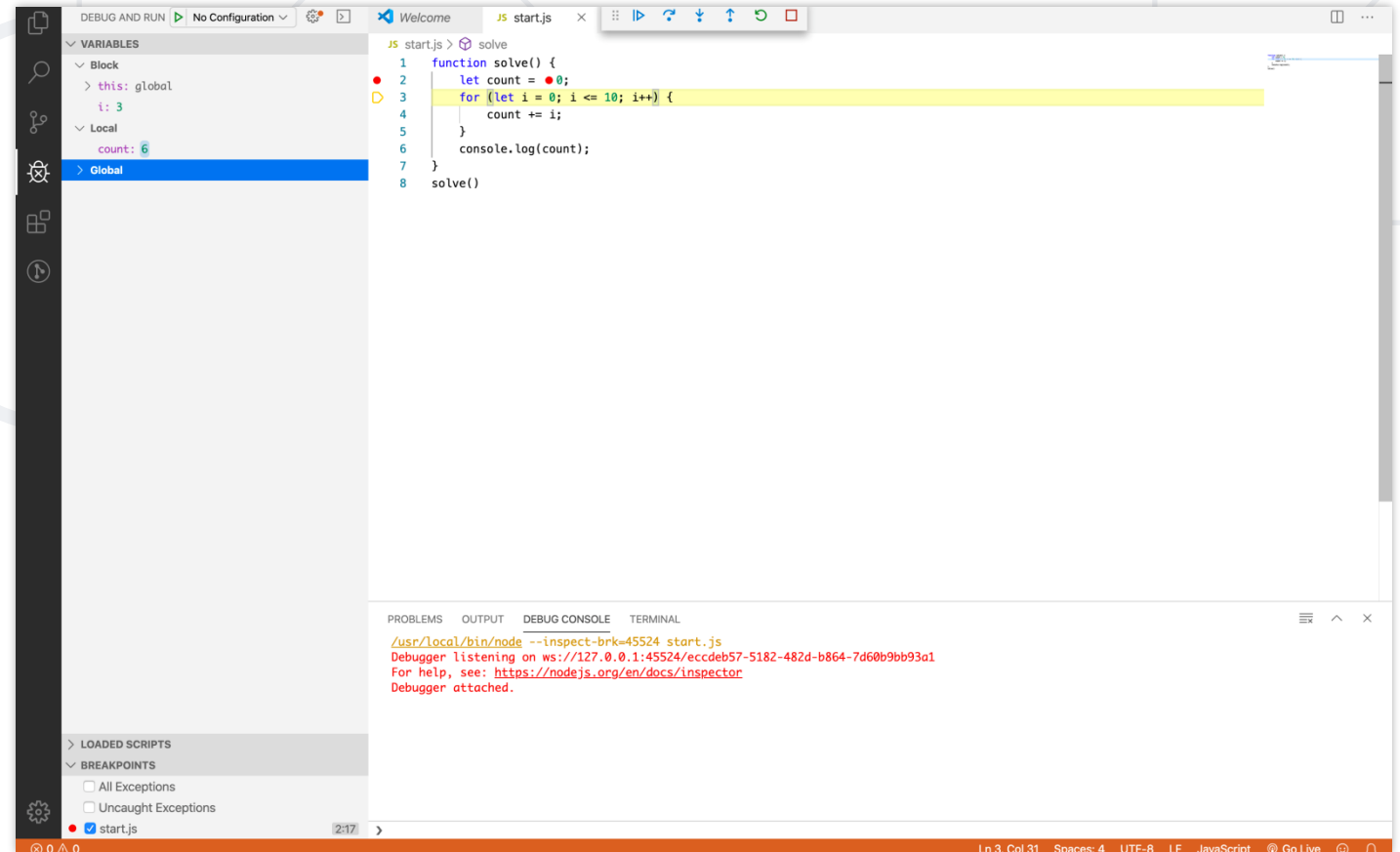
```
'use strict';           // File-level  
mistypeVariable = 17;    // ReferenceError
```

```
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



# 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]**



# Live Demonstration

Lab Problems 7 & 8




# Language Specifics

Type Coercion, Functions and Scope



# First-class Functions

- First-class functions – a function can be passed as an **argument** to other functions
- Can be **returned** by another function and can be **assigned** as a value to a variable




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

Callback function

# Nested Functions

- Functions can be **nested** - hold other functions
  - Inner functions have **access** to **variables** from **their parent**



```
function hypotenuse(m, n) { // outer function
  function square(num) { // inner function
    return num * num;
  }
  return Math.sqrt(square(m) + square(n));
}
```

3, 4



5

# Hoisting



- Variable and function declarations are **put into memory** during the **compile** phase, but stay exactly where you **typed** them in your code
- **Only declarations are hoisted**

# Hoisting Variables



```
console.log(num); // Returns undefined  
var num;  
num = 6;
```

```
num = 6;  
console.log(num); // returns 6  
var num;
```

```
num = 6;  
console.log(num); // ReferenceError: num is not defined  
let num;
```

```
console.log(num); // ReferenceError: num is not defined  
num = 6;
```

# Hoisting Functions



```
run(); // running  
function run() {  
    console.log("running");  
};
```

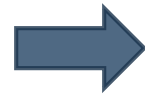
```
walk(); // ReferenceError: walk is not defined  
let walk = function () {  
    console.log("walking");  
};
```

```
console.log(walk); //undefined  
walk(); // TypeError: walk is not a function  
var walk = function () {  
    console.log("walking");  
};
```

# Problem: Aggregate Elements

- Create function that applies **sum**, **inverse sum** and **concatenation**
  - Try to use a **nested aggregating function**
- Input will be an **array** of numbers
- **Print** the result on **separate lines** on the console

[1, 2, 4]



```
7 // sum: 1 + 2 + 4
3.5 // inverse: 1/1 + 1/2 + 1/4
124 // concat: '1' + '2' + '4'
```

# Solution: Aggregate Elements

```
function aggregateElements(elements) {  
    aggregate(elements, 0, (a, b) => a + b);  
    aggregate(elements, 0, (a, b) => a + 1 / b);  
    aggregate(elements, '', (a, b) => a + b);  
  
    function aggregate(arr, initVal, func) {  
        let val = initVal;  
        for (let i = 0; i < arr.length; i++)  
            val = func(val, arr[i]);  
        console.log(val);  
    }  
}
```

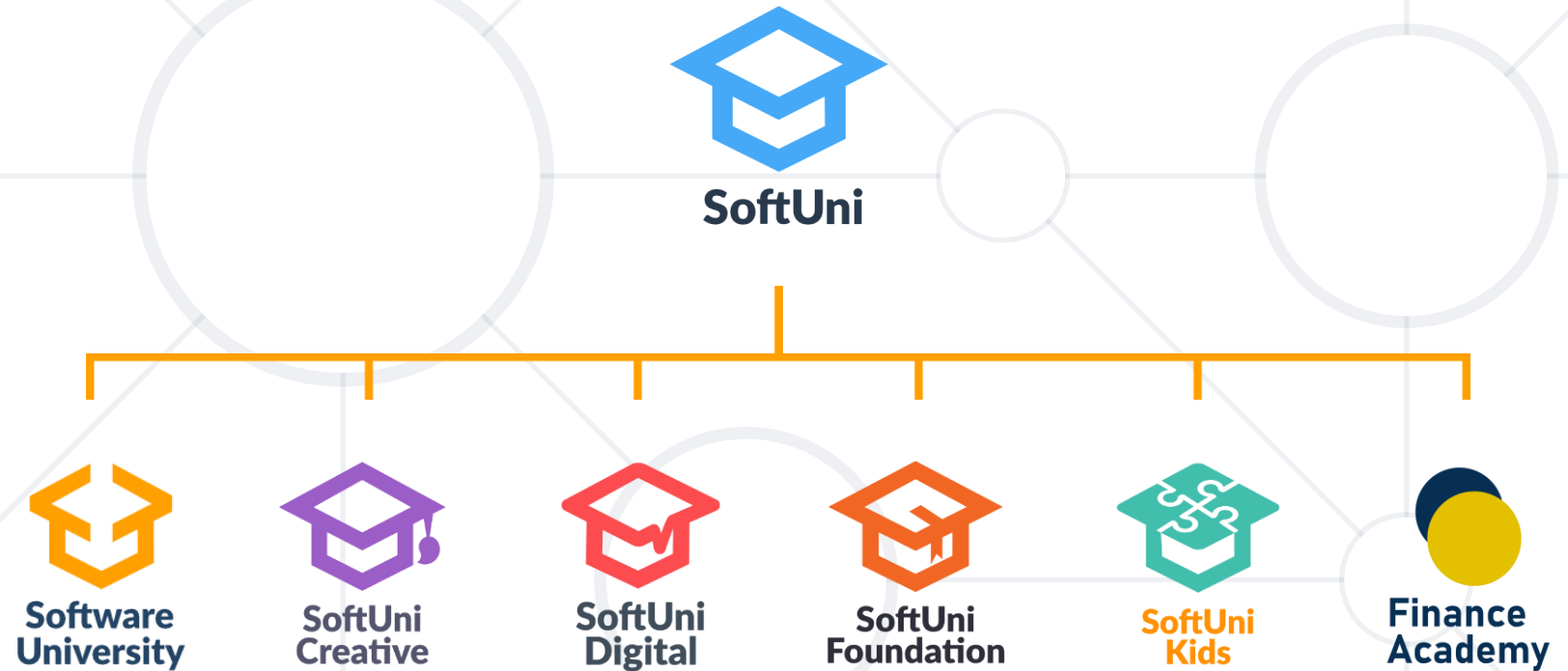


- JavaScript is a **multi-paradigm** language
- Variables are used to **store** data **references**
  - **let**, **const** and **var** are used to **declare variables**
- Arithmetic operators take **numerical values** as their operands
- Functions can:
  - **Take parameters** and **return result**
  - **Hold other functions** inside them





# Questions?



# SoftUni Diamond Partners

**SUPER  
HOSTING  
.BG**



**Coca-Cola HBC  
Bulgaria**



**POKERSTARS**  
POKER | CASINO | SPORTS  
a Flutter International brand

**INDEAVR**  
Serving the high achievers



**AMBITIONED**

 **DRAFT  
KINGS**



**SOFTWARE  
GROUP**

createX



**Postbank**

Решения за твоето утре

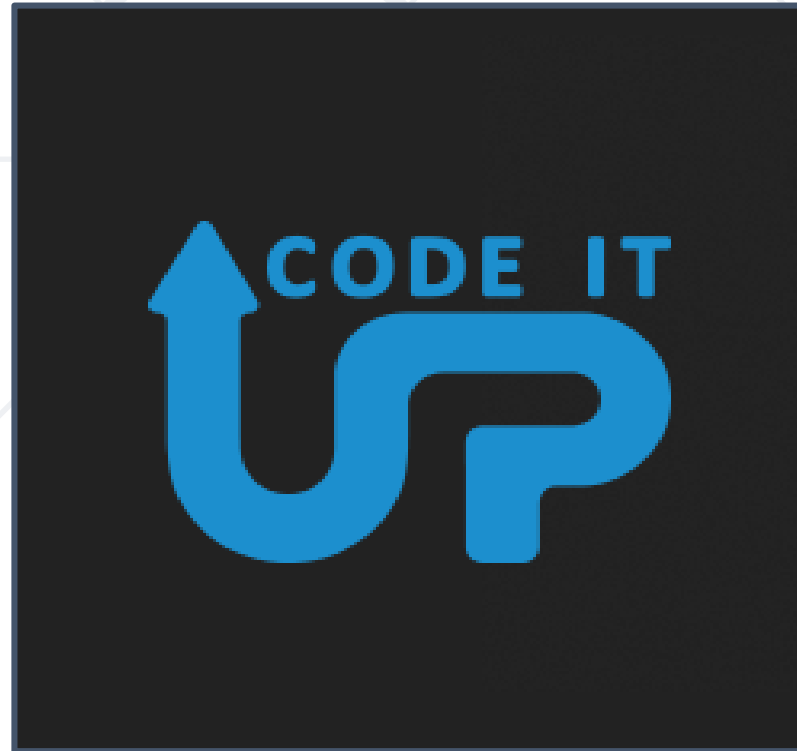


**BOSCH**

**DXC**  
TECHNOLOGY



**SmartIT**



- 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>



- 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)

