ΕΠΛ425
Τεχνολογίες Διαδικτύου
(Internet Technologies)

The Basics of JavaScript (JS)

Διδάσκων
Δρ. Χριστόφορος Χριστοφόρου
christophoros@cs.ucy.ac.cy

Goals

Introduction to Front-End Development:

- HTML to create the document structure and content
- CSS to control its visual/stylist aspect
- JavaScript for interactivity

What we studied so far

HTML + **CSS** **produces**

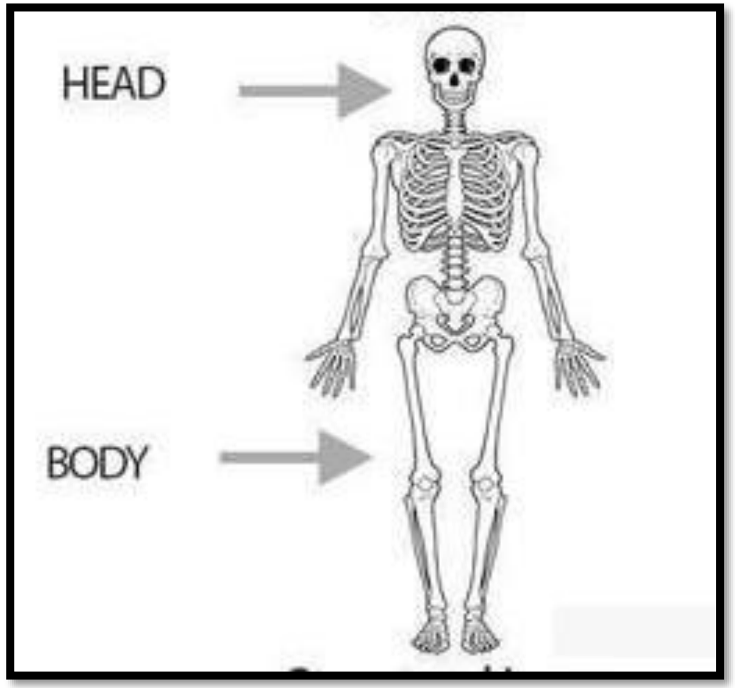
Describes the content and structure of the page

Describes the appearance and style of the page

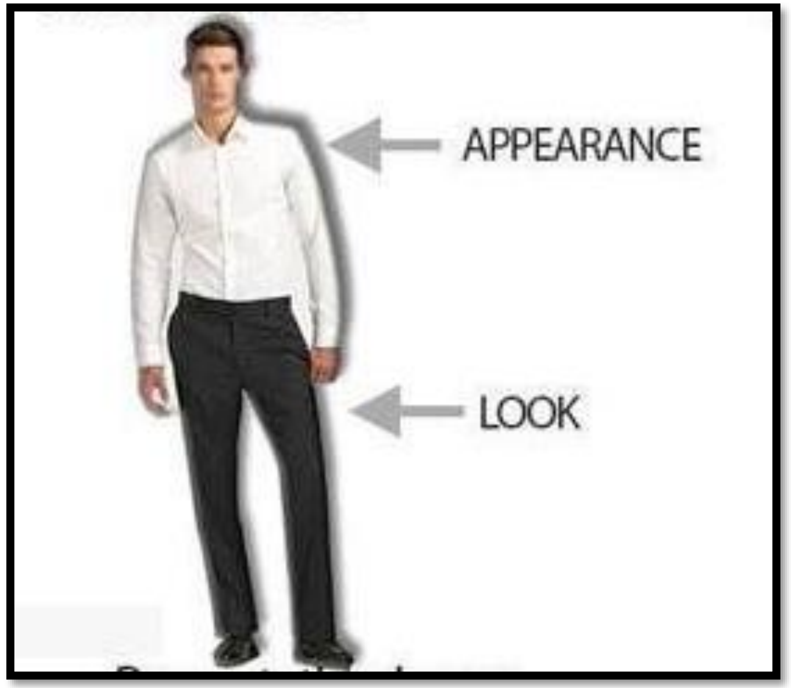
A web page… that doesn't do anything

JavaScript (JS)

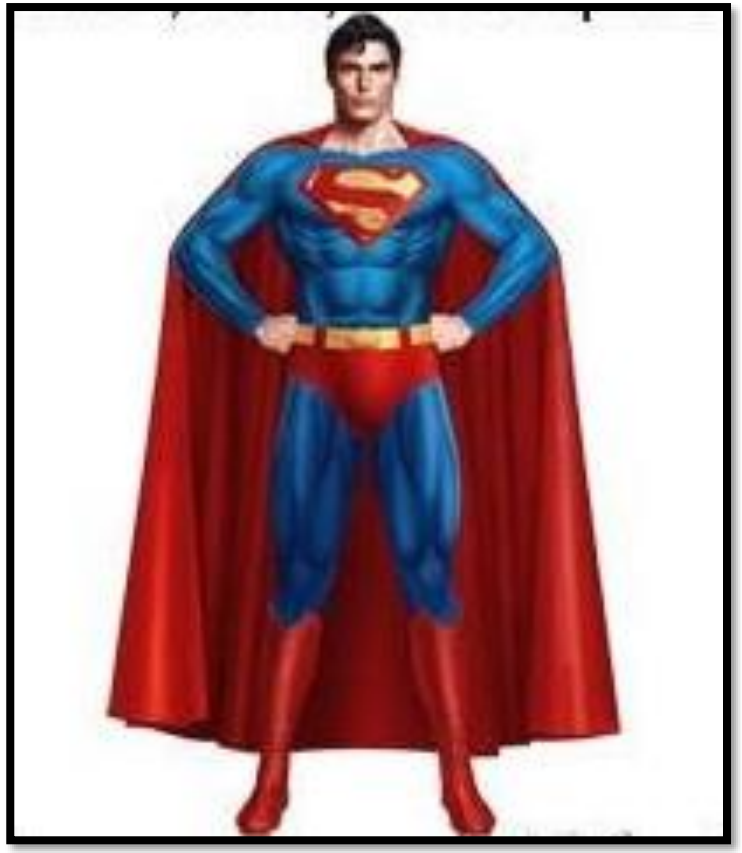
- JavaScript is a **light-weight interpreted** programming language with **object-oriented capabilities**, **that adds interactivity to your website!**



HTML



HTML + CSS



HTML + CSS + JS

JavaScript (JS)

- Javascript helps you developing great **front-end (client-side)** as well as **back-end (server-side)** **softwares** using different **server-side** Javascript based **frameworks** like **jQuery** **and** **Node.js**.
  - **Client-Side** (**Front-end**) **JavaScript,** allows **interaction with the user**, **control the browser** and **dynamically create HTML content.**→ You can **change the content of the HTML** or the **CSS** applied to an element **without reloading** **the page**.
  - **Server Side** **(Back-end)** JavaScript (referred as **Node.js**) is an **extended version** of **JavaScript** that **runs on the server** and **enables** **back-end** **access to databases**, **file systems**, and **servers.**

JavaScript (JS)

- ❑ JavaScript is the **ONLY** **programming language** **native** to the **web browser** and one of the **most widely used** **programming languages**, both for Front-end and Back-end!!!
- ❑ **It comes** **installed (built into) on every modern web browser** and so you really **do not need** any **special environment setup** to use **JavaScript**. Chrome, Mozilla Firefox, Safari and every browser you know as of today, **supports Javascript**.

History: JavaScript made its **first appearance** in Netscape 2.0 in **1995** with the name **LiveScript**. However, **Netscape** **changed** **its name** to **JavaScript**, **possibly because** of the **excitement being generated by Java**.

JavaScript (JS) Syntax

- JavaScript can be **implemented** using **JavaScript statements** that are placed within the **<script>... </script> HTML tags** in a web page.
- The **<script>** tag **alerts the browser** to **start interpreting** all the text between these tags **as a script**. A simple syntax of your JavaScript will appear as follows.

```
    <script type = "text/javascript">
        JavaScript code
    </script>
```

Note: Old JavaScript examples may use a **type attribute** like:

<script **type** = **"text/javascript"**>

The **type attribute** is **NO LONGER REQUIRED!!!**

JavaScript is the **DEFAULT scripting language in HTML!!!**

JavaScript (JS) Syntax

- JavaScript **ignores spaces**, **tabs**, and **newlines** that appear in JavaScript programs.
- Thus, **you can use spaces**, **tabs**, and **newlines freely** in your program and **you are free to format** and **indent your programs** in a **neat** and **consistent way** that makes the **code easy to read** and **understand**.
- **Simple statements** in JavaScript are generally **followed** by a **semicolon ; character**, just as they are in C, C++, and Java.
- JavaScript, however, **allows you** to **omit this semicolon if each of your statements** are **placed** on **a separate line**.

```
    <script>
        name = 'Chris'
        epitheto = 'Christophorou'
        age = 44; phone = '99887766';
    </script>
```

No need of a semicolon here.

JavaScript (JS) Syntax

- JavaScript is a **case-sensitive language**. This means that JavaScript can **differentiate** and **treat differently** between **upper case** and **lower case** characters (i.e., in code **FirstName** and **firstname** are **two different variables**).

- JavaScript supports both C-style and C++-style **comments**:

  - Any text **after** a // is treated as a comment and is ignored by JavaScript.

  - Any **text between** the characters /* and */ is treated as a comment. This may span multiple lines.

```
<script>

 // This is a comment.

 /*
  * This is a multi-line comment in JavaScript
  * Very similar to comments in C Programming
  */

</script>
```

JavaScript (JS) Syntax

- Now, **if** the user's **browser does not support JavaScript** or JavaScript is **not enabled**, you can **display** a **warning message** to the user using **<noscript> tags**.

- To do that you can **add** a **<noscript> block immediately after** the <script> block, to display the message on the screen.

```
<!DOCTYPE html>
<html>

<head>
    <title>JavaScript</title>
</head>

<body>
    <script>
         document.write("Hello World!")
    </script>

    <noscript>
        Sorry...JavaScript is needed.
    </noscript>
</body>

</html>
```

JavaScript (JS) Placement in HTML File

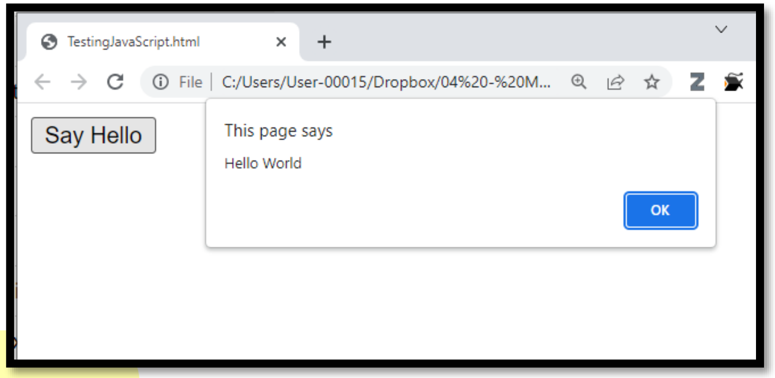
- There is a **flexibility** given **to include JavaScript** code **anywhere** in an **HTML document**. Some **ways** to **include JavaScript** in an **HTML file** are as follows:
  - **Script** in <head>...</head> section.
  - **Script** in <body>...</body> section.
  - **Script** in <body>...</body> and <head>...</head> sections.

  - **Script** in an **external file (.js)** and **then include it** in <head>...</head> section → **PREFERED WAY!!!**

Script in <head>...</head> section

- If you want to have a **script** **run on some event**, such as when a **user clicks a button**, then you can place that script in the <head> as follows!

```
<!DOCTYPE html>
<html>

<head>
    <title>Page Title</title>

    <script>
        function sayHello() {
            window.alert("Hello World")
        }
    </script>
</head>

<body>
    <button type="button" onclick="sayHello()">Say Hello</button>
</body>

</html>
```



This page says
Hello World

*In JavaScript, the **window** **object** is the **global scope** **object**. This means that **variables**, **properties**, and **methods** **by default** **belong** to the **window** **object**.*

*This means that **specifying** the **window** keyword before the alert method is **optional!***

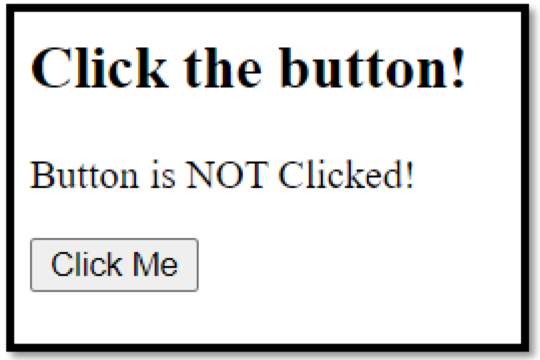
Thus:

alert("Hello World")** is **also correct.

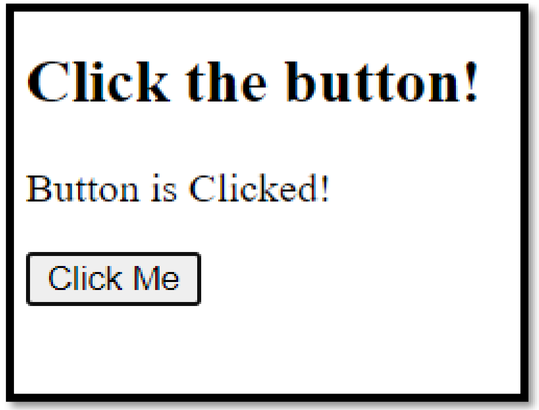
Script in <body>...</body> section

- Note that, if you want to have a **script run on some event**, such as when a **user clicks a button**, then you can **also place** that **script** in the <body> part.

```
<!DOCTYPE html>
<html>

<body>
   <h2>Click the button!</h2>
   <p id="par1">Button is NOT Clicked!</p>
   <button type="button" onclick="myFunction()">Click Me</button>

   <script>
      function myFunction() {
         document.getElementById("par1").innerHTML = "Button is Clicked!";
      }
   </script>


</body>

</html>
```


Click the button!

Button is NOT Clicked!

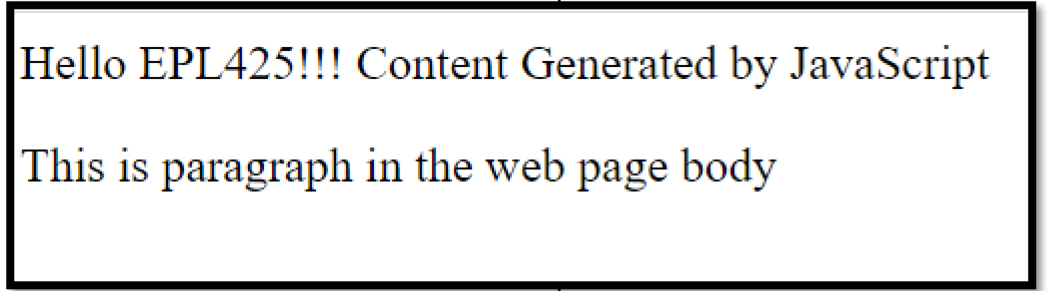

Click the button!

Button is Clicked!

Script in <body>...</body> section

- Also, if you need a **script to run as the page loads,** so that the **script generates content** in the page, then the script goes in the <body> portion of the document. In this case, you would not have any function defined using JavaScript.

```
<!DOCTYPE html>
<html>

<head>
    <title>Page Title</title>
</head>

<body>
    <script>
        document.write("Hello EPL425!!! Content Generated by JavaScript")
    </script>

    <p>This is paragraph in the web page body </p>

</body>

</html>
```

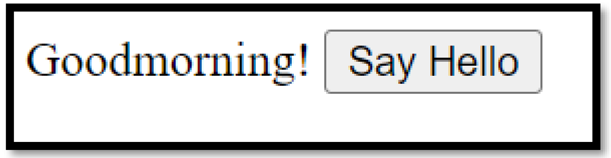


```
Hello EPL425!!! Content Generated by JavaScript

This is paragraph in the web page body
```

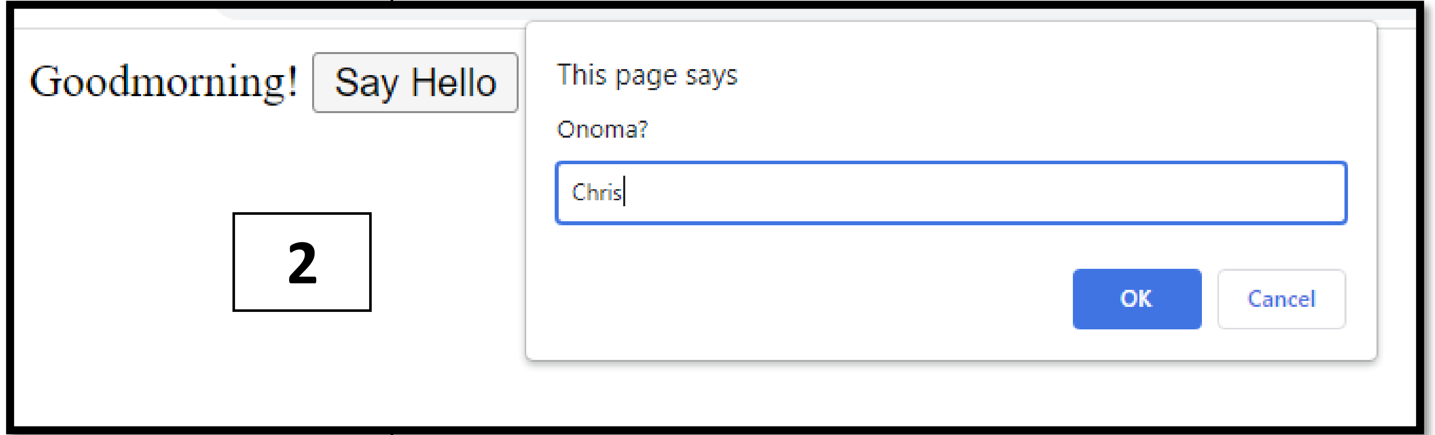
Script in <body>...</body> and <head>...</head> sections

- In case you need both of the aforesaid to occur…

```
<!DOCTYPE html>
<html>

<head>
    <script>
        function sayHello() {
            let name = prompt("Onoma?")
            document.write("Hello " + name + "!")
        }
    </script>
</head>

<body>
    <script>
        document.write("Goodmorning!")
    </script>

    <input type="button" onclick="sayHello()" value="Say Hello" />
</body>
```


Goodmorning! Say Hello

1

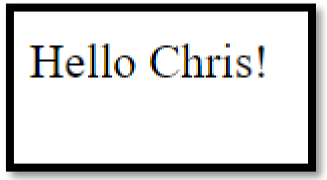

Goodmorning! Say Hello

This page says

Onoma?

Chris

2


Hello Chris!

3

You can declare a button like this as well. We will see this when we discuss about HTML forms.

Q: Where is the button and the Good morning! Why is only text???

Script in <body>...</body> and <head>...</head> sections

A: **document.write()** method **when executed**, **clears first any content previously loaded in the browser** (like the **button** and the "**Goodmorning!" text**) before displaying the "Hello Chris!" on the browsers.

So...**you should be carefull** when you use the **document.write()** method.

JavaScript in External File

- As you begin to **work more extensively** with JavaScript, you will be likely to find that **there are cases** where you are **reusing identical JavaScript code** on multiple pages of a site

- In this case, is better to **write** the JavaScript code in an **external file,** with extension "**.js**".

JavaScript in External File

- To **use** an **external script**, **put the URL of the script file** in the src (source) **attribute** of a <script> tag and **place** anywhere between <head> … </head> .

jsCode.js file in JS folder

```
function sayHello() {
    let name = prompt("Onoma?")
    document.write("Hello " + name + "!")
}
```

```
<!DOCTYPE html>
<html>

<head>
    <script src="JS/jsCode.js" defer></script>
</head>

<body>
    <input type="button" onclick="sayHello()" value="Say Hello" />
</body>
```

JavaScript in External File

Placing scripts in **external files** has some **advantages**:

- It **separates HTML** and **JavaScript** code
- It **makes** HTML and JavaScript **easier to read** and **maintain!**
- **Cached JavaScript** files can **speed up page loads,** in case you include it from a **Content Delivery Network (CDN)**

JavaScript in External File

- To **add several script files** to one page - **use several script tags**:

```
<script src="myScript1.js" defer></script>
<script src="myScript2.js" defer></script>
```

Note: You can **place an external script reference** in <head> or <body> as you like. The **script will behave** as **if it was located exactly where the** <script> tag **is located**.

However, is **better to place** it in <head> and use the **defer** attribute.

Always Use defer ! ! !

- You add the **defer** **attribute** onto the <script> tag **so that the JavaScript doesn't execute UNTIL AFTER the DOM is loaded** (see mdn for more details):

```
<script src="JS/script.js" defer></script>
```

- Other **old-school** way of doing this (but **DON'T DO THIS**) is to **put the** <script> tag **at the bottom** of the **page (**exactly before the </body> tag**)**.
- You will see tons of examples on the internet that do this. They are out of date. **defer** is widely supported and **better**.

Some Examples of what JavaScript can do

Change the HTML content and hide HTML elements:

- One of many JavaScript HTML methods is **getElementById()**
- The example below "**finds**" an **HTML element** (with id="paris"), and **changes** the **element content** (**.innerHTML**) to **provide more information** about the City.
- Also, **when** the **extra info appears**, the **button** is **hidden**.

```
function moreInfo() {
    document.getElementById("paris").innerHTML = "Paris is
    the capital and most populous city of France. Also the
    Eifel Tower is located there!!!";
    document.getElementById("btn1").style.display = "none";
    }
```

Some Examples of what JavaScript can do

```html
<!DOCTYPE html>
<html>
<head>
    <title>HTML Web Page With CSS and JavaScript</title>
    <link rel="stylesheet" href="CSS/style1.css">
    <script src="JS/jsCode.js"></script>
    <meta name="viewport" content="width=device-width, initial-scale=1">
</head>

<body>
    <h1>Great Cities!</h1>
    <img id="globe" src="images/Earth_globe.png" alt="The Globe" width="200" height="auto"/>

    <div class="cities">
        <h2>London</h2>
        <p id="london">London is the capital city of England.</p>
    </div>
    <div class="cities">
        <h2>Paris</h2>
        <p id="paris">Paris is the capital and most populous city of France.</p>
        <button type="button" id="btn1" onclick="moreInfo()">More Info</button>
    </div>
</body>
</html>
```

Do you remember what is the purpose of this?

Some Examples of what JavaScript can do

*This is how we can **define** a **function with JavaScript**.*

*The **function** keyword is placed first, then the **name** of the function with **parameters** in parenthesis (if any), followed by the code in **{ }***

jsCode.js file
in JS folder

```
function moreInfo() {
    document.getElementById("paris").innerHTML = "Paris is the
    capital and most populous city of France. Also the Eifel
    Tower is located there!!!";
    document.getElementById("btn1").style.display = "none";
    }
```

Showing hidden HTML elements can also be done by changing the **style display property**:

```
document.getElementById("btn1").style.display = "block";
```

Some Examples of what JavaScript can do

- **Note1:** The **document** object **represents your web page**. If you want to **access any element** in an **HTML page**, you **always start with accessing** the **document object**.

- **Note2:** A **JavaScript function** is a **block of JavaScript code**, that can be **executed** when **"called" for**. For example, a function can be called when **an event occurs**, like when the **user clicks** a **button** (i.e., onclick="moreInfo()")

We will see more about functions later in this Lecture!!!

Let see now an Example!

With this HTML Code .....

```
<!DOCTYPE html>
<html>

<head>
    <title>HTML Web Page With CSS and JavaScript</title>
    <link rel="stylesheet" href="CSS/style1.css">
    <script src="JS/jsCode.js"></script>
    <meta name="viewport" content="width=device-width, initial-scale=1">
</head>

<body>
    <h1>Great Cities!</h1>
    <img id="globe" src="images/Earth_globe.png" alt="The Globe" width="200" height="auto" />
    <div class="cities">
        <h2>London</h2>
        <p id="london">London is the capital city of England.</p>
    </div>
    <div class="cities">
        <h2>Paris</h2>
        <p id="paris">Paris is the capital and most populous city of France.</p>
        <button type="button" id="btn1" onclick="moreInfo()">More Info</button>
    </div>
</body>

</html>
```

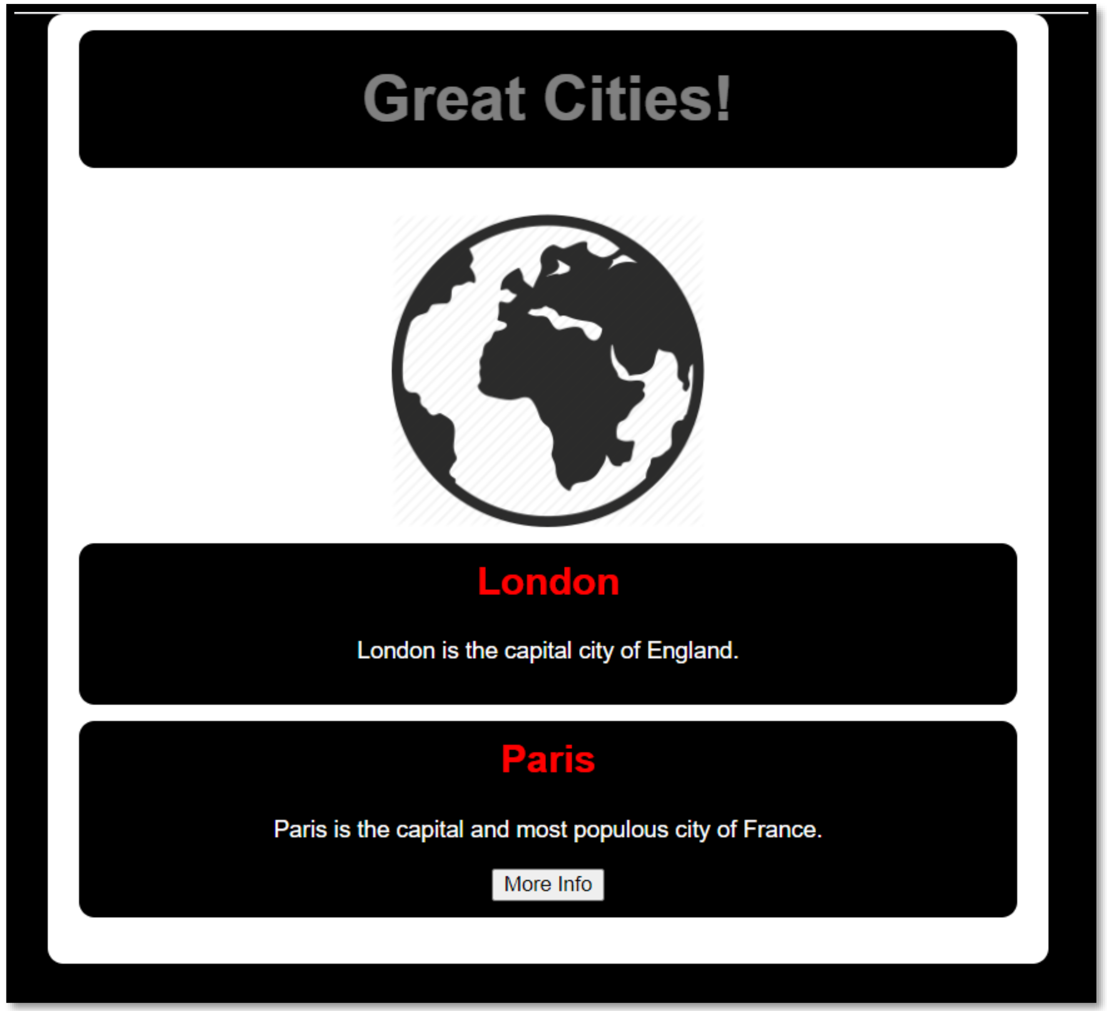
Let see now an Example!

```css
html {
    background-color: black;
}

body {
    width: 600px;
    background-color: white;
    text-align: center;
    font-family: Arial, Helvetica, sans-serif;
    line-height: 1.2;
    margin: 0px auto;
    padding: 10px 20px 20px 20px;
    border-radius: 10px;
}

h1 {
    font-size: 40px;
    background-color: black;
    color: rgba(255, 255, 255, 0.5);
    padding: 20px 20px;
    margin-bottom: 30px;
    margin-top: 0;
    border-radius: 10px;
}

div.cities {
    background-color: black;
    margin: 10px 0;
    padding: 10px;
    border-radius: 10px;
}

h2 {
    margin-top: 0;
    font-size: 25px;
    color: red;
}

p {
    font-size: 15px;
    color: white;
}


img {
    margin: 0px auto;
    display: block;
}
```

And this CSS Code .....

This is the style1.css file included in folder CSS

Let see now an Example!


Great Cities!

London

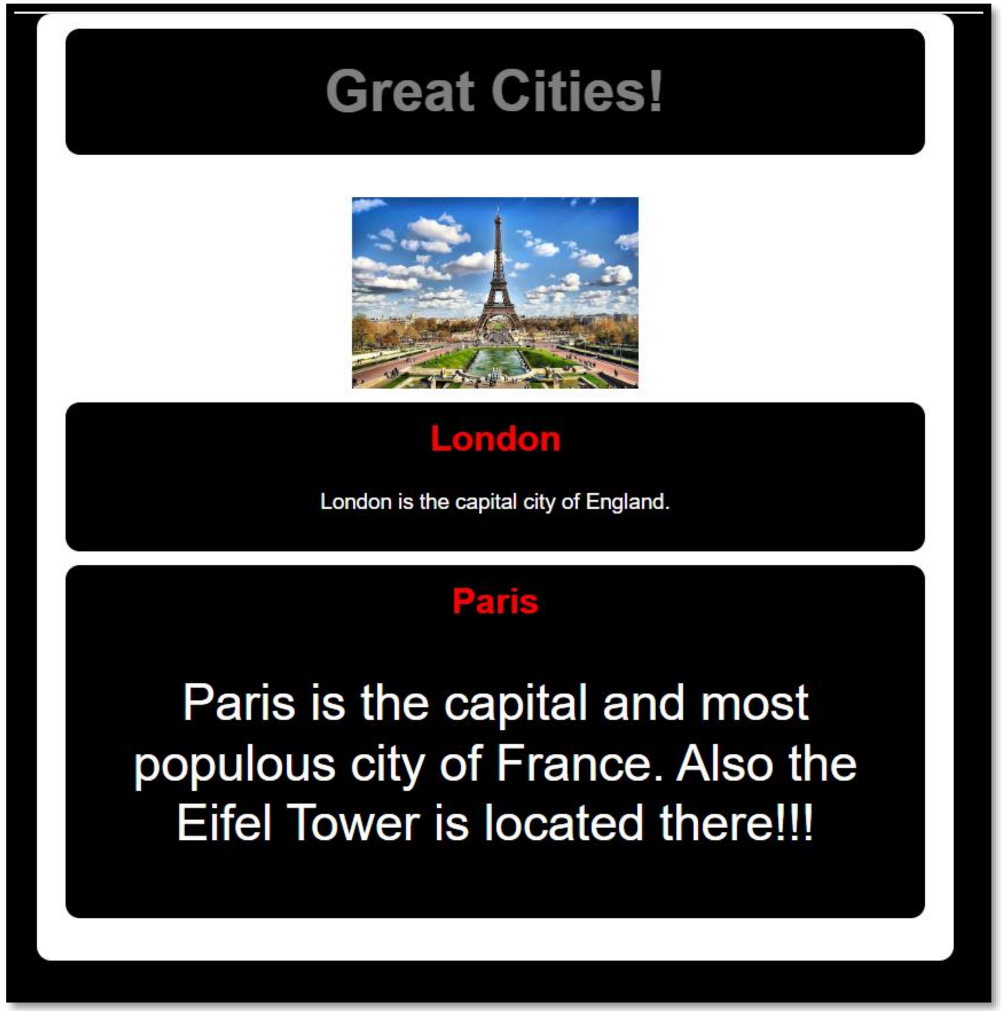
London is the capital city of England.

Paris

Paris is the capital and most populous city of France.

This is how our web page will be displayed on the browser

Let see now an Example!



Great Cities!

London

London is the capital city of England.

Paris

Paris is the capital and most populous city of France. Also the Eifel Tower is located there!!!

And this is how our web page we want to look like, after the button "More Info" is pressed!

Let see now an Example!

Here we want to change the HTML attribute values, for example **change** the **value** of the **src** (source) **attribute** of an <img> tag:

- The example below will "find" the HTML element with id="globe" (which is the globe image) and change it with a picture of Paris city.

```
document.getElementById('globe').src='images/pic_paris.jpg';
```

Also we want to change the HTML CSS Styles, for example the font size of the text:

- The example below will **"find"** the HTML element with id="paris" (which is the globe image) and change it with a picture of Paris city.

```
document.getElementById('paris').style.fontSize ="35px";
```

Some Examples of what JavaScript can do

And here is the JavaScript code doing that!!!

jsCode.js file in JS folder

```
function moreInfo() {
    document.getElementById("paris").innerHTML = "Paris is the
    capital and most populous city of France. Also the Eifel Tower
    is located there!!!";
    document.getElementById("btn1").style.display = "none";
    document.getElementById('globe').src = 'images/pic_paris.jpg';
    document.getElementById('globe').alt = 'Paris City';
    document.getElementById('paris').style.fontSize = "35px";
}
```

JavaScript Output

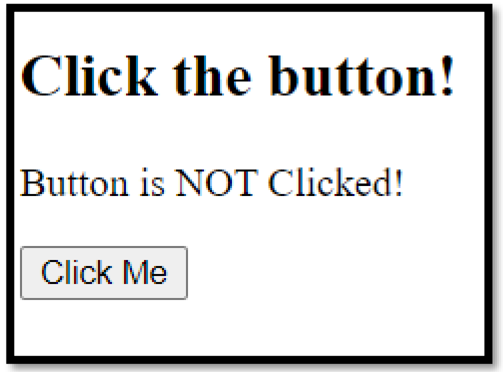
- As we saw in the previews examples, **JavaScript** can **"display"** data in **different ways**:
  - Writing into an **HTML Element**, using **innerHTML property**. Changing the **innerHTML** property of an HTML element is **the most common way** to **display data** in **HTML**.
  - Writing into the **HTML document** using **document.write()**. **Note:** Using **document.write() after** an **HTML document is loaded**, **will delete all existing HTML.** The document.write() method **should only be used for testing.**
  - Writing into an **alert box**, using **window.alert(…), or just alert(…)**

JavaScript Output for Debugging Purposes → console.log()

- The JavaScript programming code **might contain syntax errors**, or **logical errors** which many of these errors are **difficult to diagnose**.
- Often, **when your code contains errors**, **NOTHING will happen**. Usually there are **no error messages**, and you will get **no indications where to search** for errors.
- For **debugging purposes** you **can use** the **console.log()** method to display data in the **browser console** **(debugger window)!!!**

JavaScript Output for Debugging Purposes → console.log()

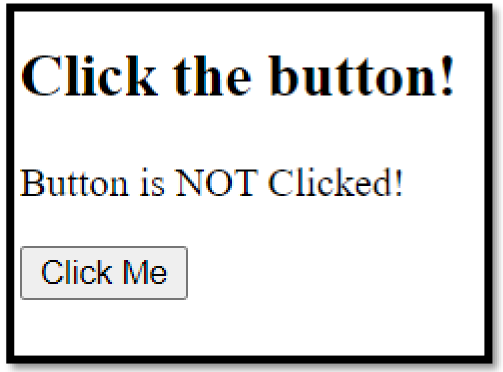
```
<!DOCTYPE html>
<html>

<body>

    <h2>Click the button!</h2>

    <p id="par1">Button is NOT Clicked!</p>

    <button type="button" id="btn1" onclick="myFunction()">Click Me</button>

    <script>
        function myFunction() {
            console.log("Button is Clicked!");
        }
    </script>

</body>

</html>
```



Click the button!

Button is NOT Clicked!

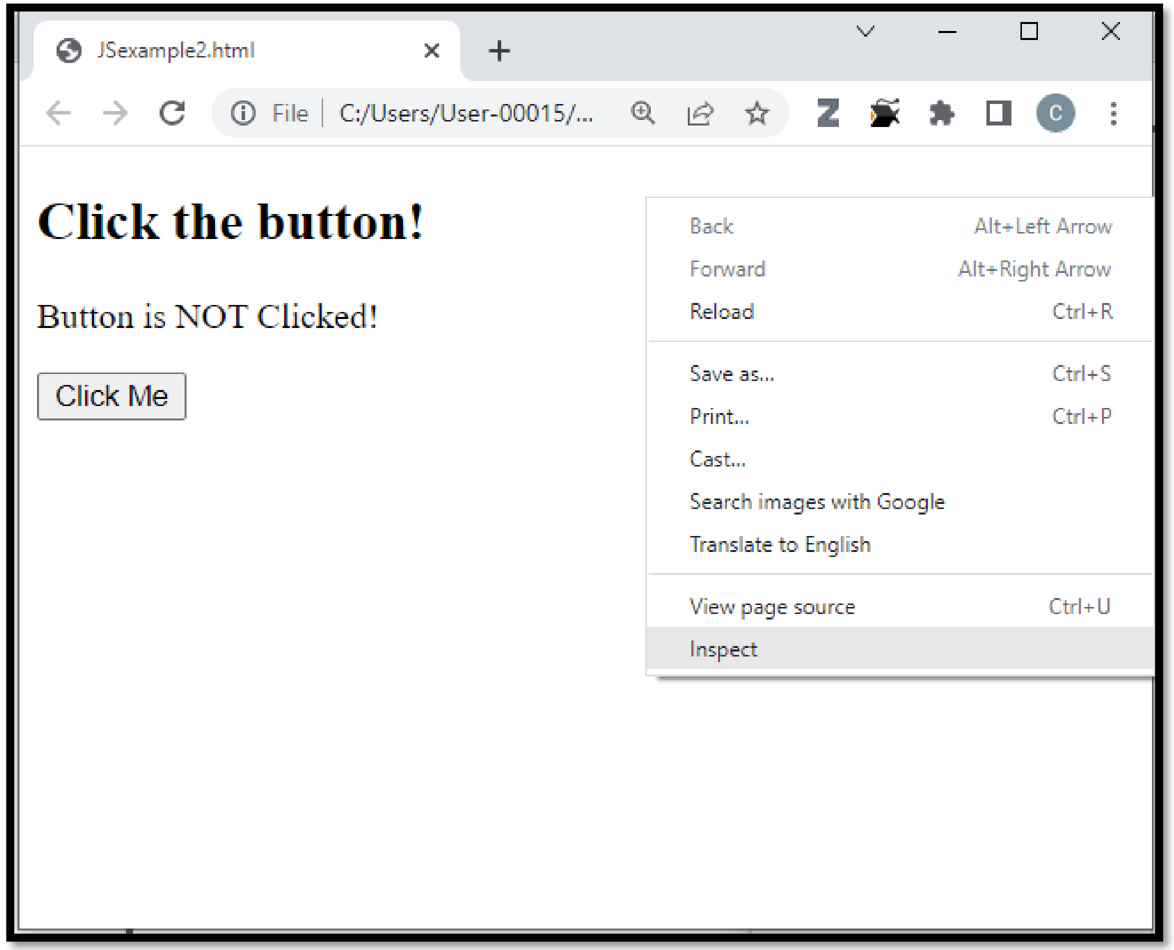
Button was clicked but nothing happened!!!



Click the button!

Button is NOT Clicked!

JavaScript Output for Debugging Purposes → console.log()

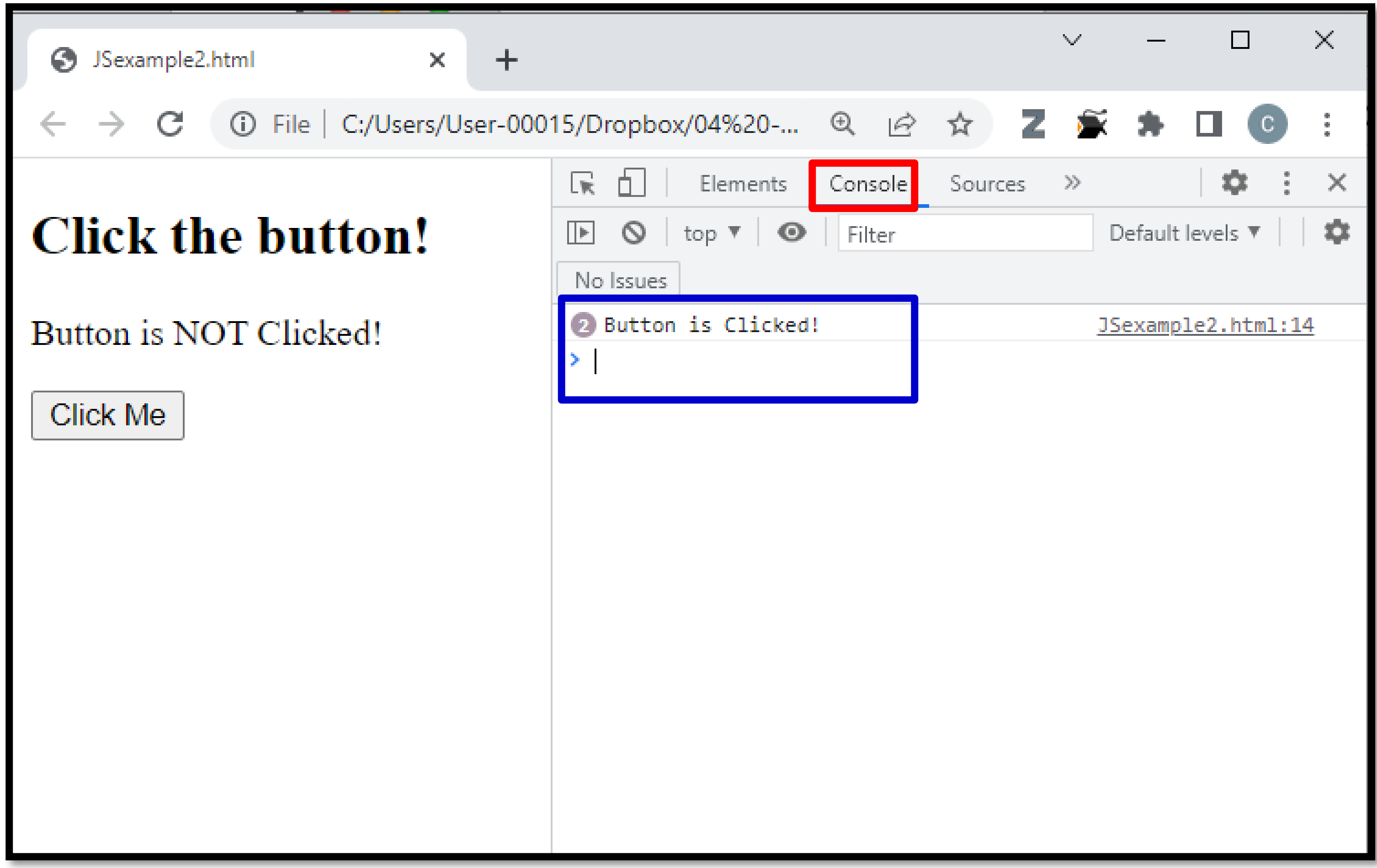


Click the button!

Button is NOT Clicked!

Click Me

Right-click (or control-click on Mac) and choose "Inspect"



Click the button!

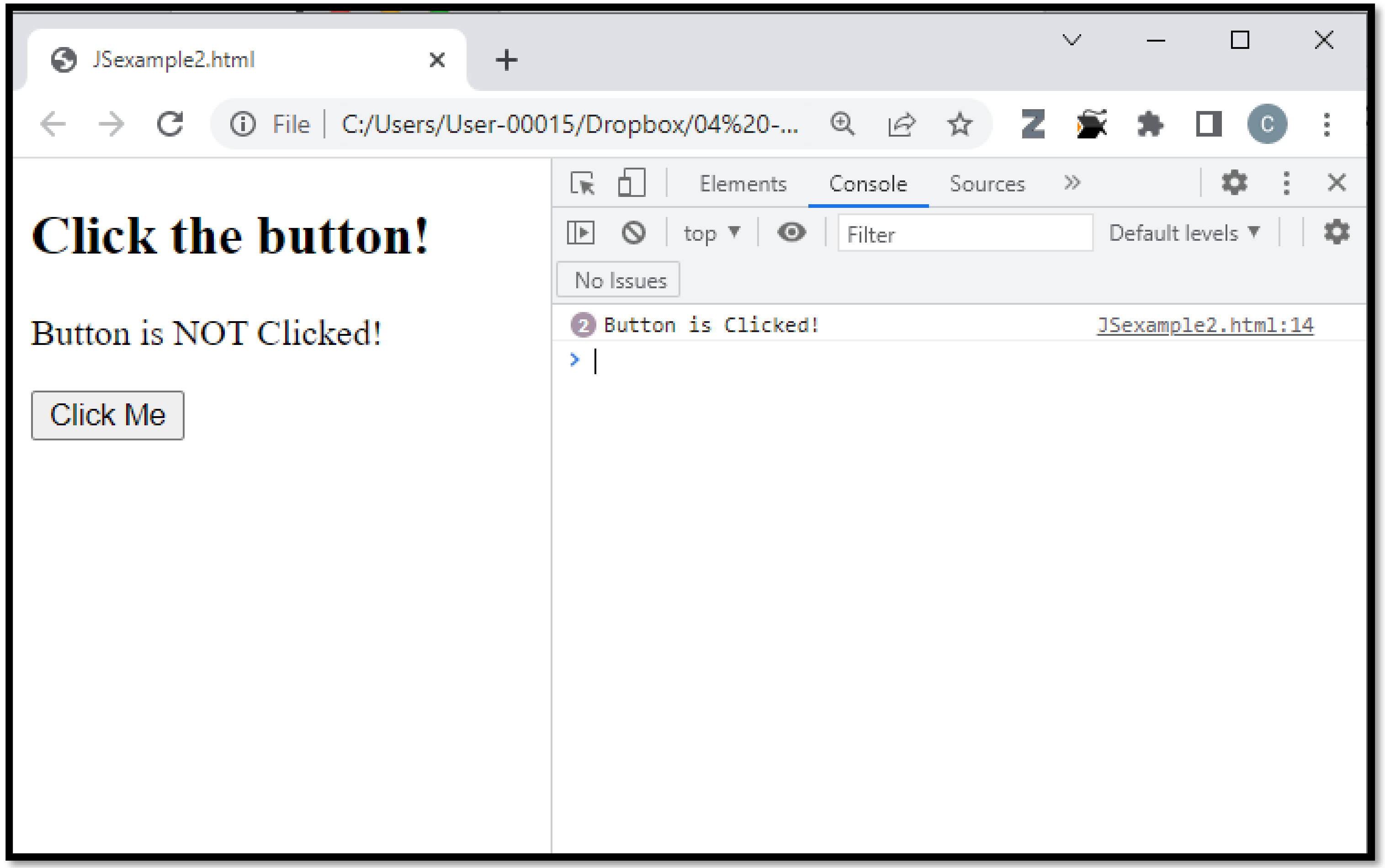
Button is NOT Clicked!

Click Me

```
2 Button is Clicked!                    JSexample2.html:14
>
```

Click "Console" tab

JavaScript Output for Debugging (console.log())


JSexample2.html

C:/Users/User-00015/Dropbox/04%20-...

Click the button!

Button is NOT Clicked!

```
2 Button is Clicked!                    JSexample2.html:14
>
```

The "Console" tab is a **read–eval–print loop (REPL)**, also termed an interactive toplevel or language shell, that takes **single user inputs** (i.e., JavaScript expressions), **executes them**, and **returns the result** to the user;

We will use this Console to test and debug our JavaScript code!

JavaScript language features

Note: In HTML, JavaScript programs are executed by the web browser!!! Thus, to test and debug your JavaScript programs must be included and executed THROUGH AN HTML PAGE.

Same as Java/C++/C-style langs

for loop:

```
for (let i = 0; i < 5; i++) { … }
```

while loop:

```
while (Condition) { … }
```

Comment:

```
// This is a comment
/* This is a comment */
```

Same as Java/C++/C-style langs

Semicolons ;

Add a semicolon at the end of each executable statement.

```
let a, b, c;  // Declare 3 variables
a = 5;        // Assign the value 5 to a
b = 6;        // Assign the value 6 to b
c = a + b;    // Assign the sum of a and b to c
```

Same as Java/C++/C-style langs

if- else if -else statements:

```
if (Condition1) {
     ...
}
else if (Condition2) {
     ...
}
else{
   ...
}
```

Same as Java/C++/C-style langs

Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (ES2016)
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

Assignment Operators

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

Comparison Operators

Operator	Description
==	equal to
===	**equal value and equal type**
!=	not equal
!==	**not equal value or not equal type**
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

Logical Operators

Operator	Description
&&	logical and
||	logical or
!	logical not

Type Operators

Operator	Description
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

Equality

JavaScript's == and != are basically "broken": they **do an implicit type conversion before the comparison**.

```
'' == '0'  // false
'' == 0  // true
0 == '0'  // true
[''] == ''  // true
false == undefined  // false
false == null  // false
null == undefined  // true
```

Equality

Instead of fixing == and != , the ECMAScript (ES) standard kept the existing behavior but **added** **===** and **!==**

```
'' === '0'  // false
'' === 0  // false
0 === '0'  // false
[''] === ''  // false
false === undefined  // false
false === null  // false
null === undefined  // false
```

For Checking Equality **always use === and !== and don't use == or !=**

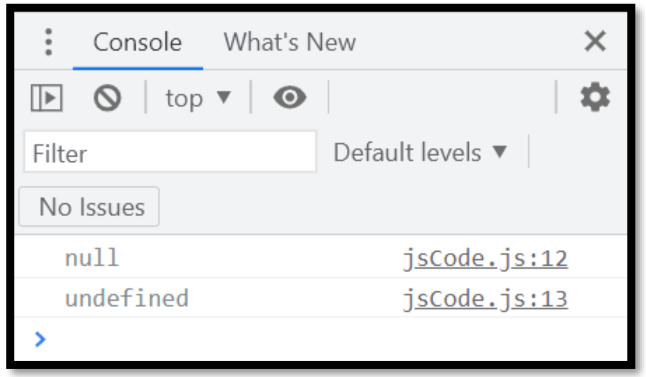
*ECMAScript (ES) was created to standardize JavaScript

null and undefined

What's the difference?

- **null is a value representing the absence of a value**, similar to null in Java and nullptr in C++.
- **undefined** is a **value given** (by JavaScript) to a **variable** that **has not been assigned a real value**.

```
let x = null;
let y;
console.log(x);
console.log(y);
```



```
null          jsCode.js:12
undefined     jsCode.js:13
```

```
let x = null;
let y = undefined;
console.log(x);
console.log(y);
```

If you do not provide a value to a variable, it value will be **undefined by default….**

….however, you **can also set a variable's value to undefined**

JavaScript Keywords

- Some JavaScript statements **often start** with a **keyword** to identify the **JavaScript action** to be **performed**.

Keyword	**Description**
var	Declares a variable (will have a **global scope**)
let	Declares a **block variable** (will have a **local scope**)
const	Declares a block constant variable
if	Marks a block of statements to be executed on a condition
switch	Marks a block of statements to be executed in different cases
for	Marks a block of statements to be executed in a loop
function	Declares a function
return	Exits a function
try	Implements error handling to a block of statements

JavaScript Variables

- There are **four ways** to **declare** a JavaScript **variable**:
  - Using **var**
  - Using **let**
  - Using **const** (this is for constant)
  - Using **nothing (undeclared)**

Notes: The **var** keyword is used in all JavaScript code from 1995 to 2015. The **let** and **const** keywords were added to JavaScript in 2015.
If you want your code to **run in older browsers**, you must use **var**.

```
const discount = 0.5;
var price = 6;
quantity = 5;
let totalPrice = price * discount * quantity;
```

JavaScript Variable name

- A JavaScript **variable name** must **begin with**:
  - A letter (A-Z or a-z)
  - A dollar sign ($)
  - Or an underscore (_)
- **Subsequent characters** may be **letters**, **digits**, **underscores**, or **dollar signs**.

Since JavaScript **treats a dollar sign as a letter**, identifiers containing $ are valid variable names:

```
let $ = "Hello World";
let $$$ = 2;
let $myMoney =  $ + $$$;
```

var VS let

Variables defined with **let cannot be redeclared** in the **same scope**.

```
let variable = "Chris";
let variable = 0;
```

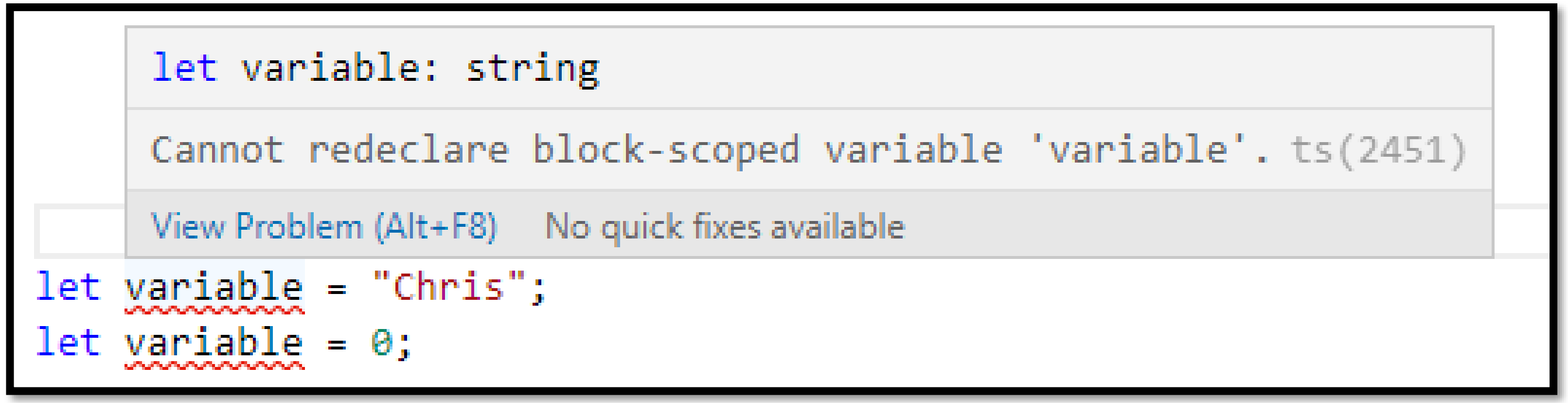
✗

```
var variable = "Chris";
var variable = 0;
```

✓

With **var** you can!!!

Variables defined with **let** have **block scope**.



```
let variable: string
Cannot redeclare block-scoped variable 'variable'. ts(2451)
View Problem (Alt+F8)    No quick fixes available
let variable = "Chris";
let variable = 0;
```

var VS let

- Before ES6 (2015), JavaScript had only **Global Scope** and **Function Scope** → ES6 introduced two important new JavaScript keywords: **let** and **const** to provide **Block Scope** in JavaScript.

```
{
  let x = 2;
}
// x can NOT be used here
```

```
{
  var x = 2;
}
// x CAN be used here
```

var VS let (Correct this)

Redeclaring a variable with let, in **another block**, **IS allowed**:

```
let x = 2;    // Allowed

{
let x = 4;     // Allowed
}
```

```
var x = 2;    // Allowed

{
var x = 3;    // Not allowed
}
```

However, even if it is allowed, I do not consider redeclaring a variable a good practice.

With **var** is **not Allowed!!!**

JavaScript Data Types

JavaScript has 8 Datatypes:

- String
- Number
- Bigint
- Boolean
- Undefined
- Null
- Object
- Symbol (check this link)

*All JavaScript numbers are stored in a 64-bit floating-point format. JavaScript **BigInt** is a new datatype (2020) that can be used to store integer values that are too big to be represented by a normal JavaScript Number.*

```
// Numbers:
let length = 16;
let weight = 7.5;

// Strings:
let color = "Yellow";
let lastName = 'Christophorou';

// Booleans
let x = true;
let y = false;

// Undefined
let age;

// Object:
const person = {firstName:"John", lastName:"Doe"};

// Array object:
const cars = ["Saab", "Volvo", "BMW"];

// Date object:
const date = new Date("2022-03-25");
```

Note: In JavaScript you **do not need to declare** the **type of the Variable**!!! This is **dynamically typed during initialization.**

JavaScript Data Types

- **JavaScript variables types** are **Dynamic**!!!
- This means that **the same variable can be used** to **hold values of different data types**:

```
let x;           // Now x is undefined
x = 5;           // Now x is a Number
x = "Chris";     // Now x is a String
x = [5, 6, 7];   // Now x is an Array of Integers
```

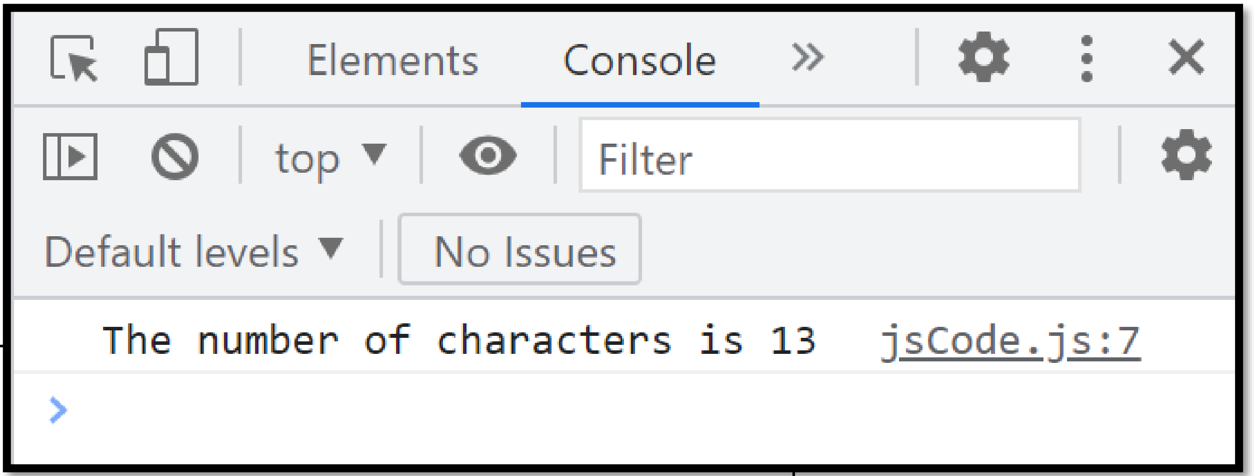
JavaScript Strings

- JavaScript String are defined using **double** or **single** quotes:

```
let carName1 = "Volvo XC60";  // Double quotes
let carName2 = 'Volvo XC60';  // Single quotes
```

- To find the **length of a String**, use the built-in **length** String **property**:

```
let text = "Christophoros";
let length = text.length;
console.log("The number of characters is " + length)
```



```
The number of characters is 13
```

JavaScript Strings

- Some String Methods:

Method	Description

slice()	**slice(start, end): Extracts a part of a string** and **returns** the **extracted part in a new string**. **End index** is **not included**. If you **omit** the **second parameter**, the method will slice out the rest of the string.
	
	`let text = "Apple, Banana, Kiwi";`
	`let part = text.slice(7, 13);  // Banana`
	`let part = text.slice(7);        // Banana, Kiwi`

JavaScript Strings

- Some String Methods:

Method	Description

replace()	Replaces a specified value with another value in a string. Does not change the string it is called on. It returns a new string. Replaces only the first match
	
	`let text = "Hello EPL425!";`
	`let newText = text.replace("Hello", "Geia sas"); // “Geia sas EPL425!";`

toUpperCase()
toLowerCase()
	A string is converted to upper case with toUpperCase(). A string is converted to lower case with toLowerCase(). Does not change the string it is called on. It returns a new string.
	
	`let text1 = "Hello World!";    // String`
	`let text2 = text1.toLowerCase();  // text2 is text1 converted to lower`
	`let text3 = text1.toUpperCase();  // text3 is text1 converted to lower`

JavaScript Strings

- Some String Methods:

Method	Description

trim()	**Removes whitespace** from **both sides** of a string. Does not change the string it is called on. It returns a new string.
	
	`let text1 = "      Hello World!      ";`
	`let text2 = text1.trim();    //text2 will take the value "Hello World!"`

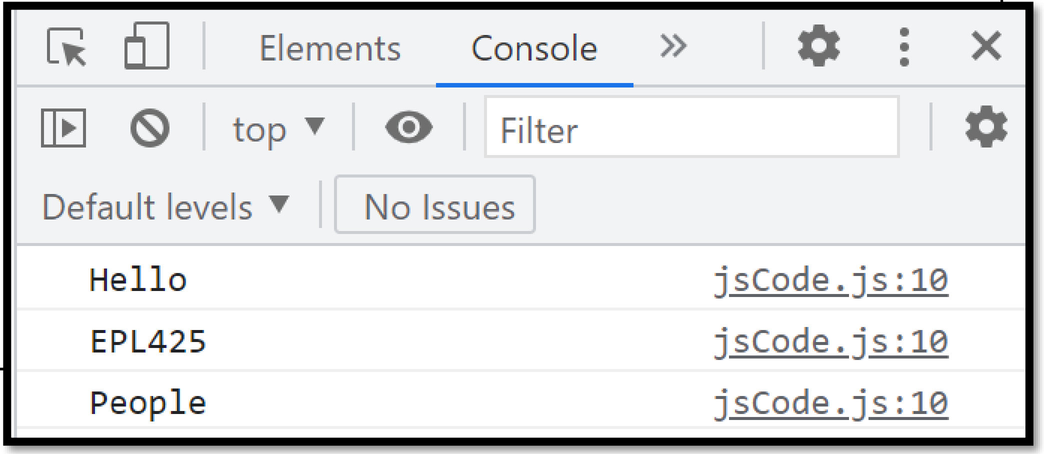
trimStart()	Similarly to trim() but removes whitespace only from the end of a string
trimEnd()	Similarly to trim() but removes whitespace only from the start of a string

JavaScript Strings

- Some String Methods:

Method	Description

split(separator)	Converts a String to an Array! If the separator is omitted, the returned array will contain the whole string in index [0]. If the separator is "", the returned array will be an array of single characters:
	
	`let text = "Hello EPL425 People";`
	`const myArr = text.split(" ");`
	
	`for (let i = 0; i < myArr.length; i++) {`
	`    console.log(myArr[i]);`
	`}`



```
Hello      jsCode.js:10
EPL425     jsCode.js:10
People     jsCode.js:10
```

JavaScript Strings

- Some String Methods:

Method	Description

indexOf(substring)
lastIndexOf(substring)
	indexOf(substring): Returns the index of (position of) the first occurrence of a string in a string. lastIndexOf() method returns the index (position) of the last occurrence of a specified value in a string. Return **-1** if the text is not found.
	
	`let str = "Please locate my students!";`
	`str.indexOf("locate");   // 7`

includes(substring)	Returns true if a string contains a specified value. Otherwise it returns false.
	
	`let str = "Please locate my students!";`
	`str.includes("locate");  // true`

Adding JavaScript Strings

- ❑ The + operator can also be used to add (concatenate) strings.

```
let text1 = "Chris";
let text2 = "Christophorou";
let text3 = text1 + " " + text2;
```

- ❑ The string stored in text3 will be "Chris Christophorou"

Adding JavaScript Strings and Numbers

- Adding two numbers, will return the sum, but **adding** a **number** and a **string** will **return a string**:

```
let x = 5 + 5;
let y = "5" + 5;
let z = "Hello" + 5;
```

Note 1: In other programming languages this will cause an ERROR, but not in JavaScript!!!

- The string stored in z will be "Hello5"

Note 2: In JavaScript, **if you add** a **number** and a **string**, JavaScript **will always treat the number as a string** and **the result will be a string**!

Adding JavaScript Strings and Numbers

JavaScript **evaluates expressions from left to right**.
Different sequences can **produce different results**:

```
let x = 16 + "Volvo";               Result: "16Volvo"
let x = "Volvo" + 16;               Result: "Volvo16"
let x = 16 + 4 + "Volvo";           Result: "20Volvo"
let x = "Volvo" + 16 + 4;           Result: "Volvo164"
```

JavaScript Strings: Template Literals

- **Template Literals** use **back-ticks** (\` \`) rather than the quotes (" ") to define a string:

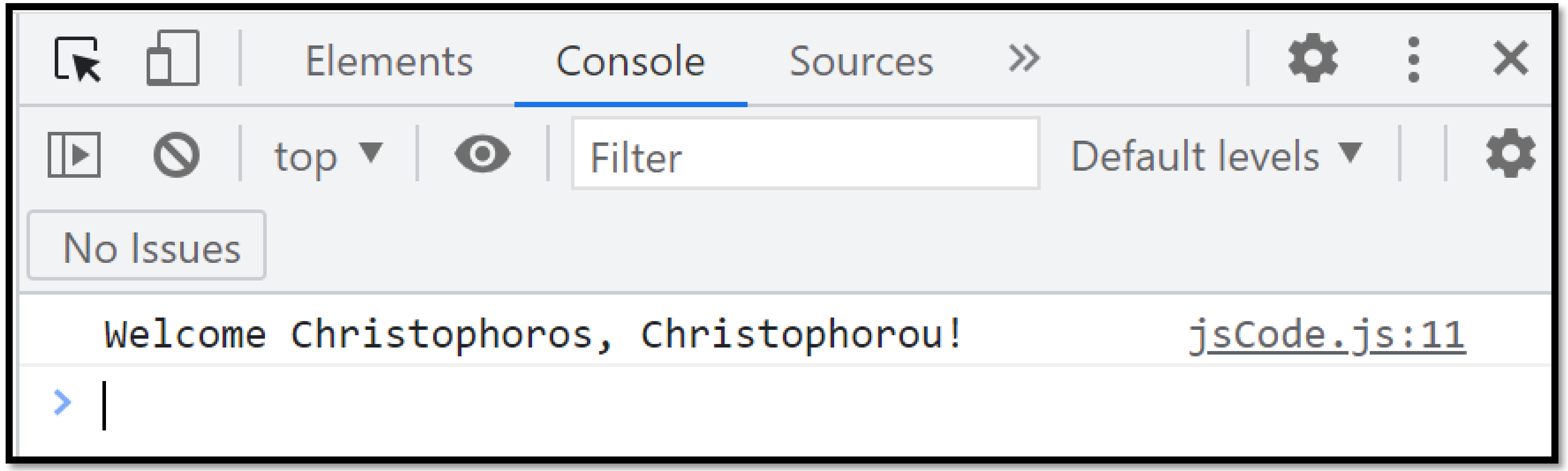
```
let greeting = `Hello EPL425 People!`;
```

- Template literals **provide** an easy way to **interpolate variables** and **expressions into strings**.
- **Automatic replacing** of **variables with real values** is **called string interpolation**. This is done by using **${varName}** in the string.

JavaScript Strings: Template Literals

❑ Variable Substitutions

```
let firstName = "Christophoros";
let lastName = "Christophorou";

let text = `Welcome ${firstName}, ${lastName}!`;
console.log(text)
```

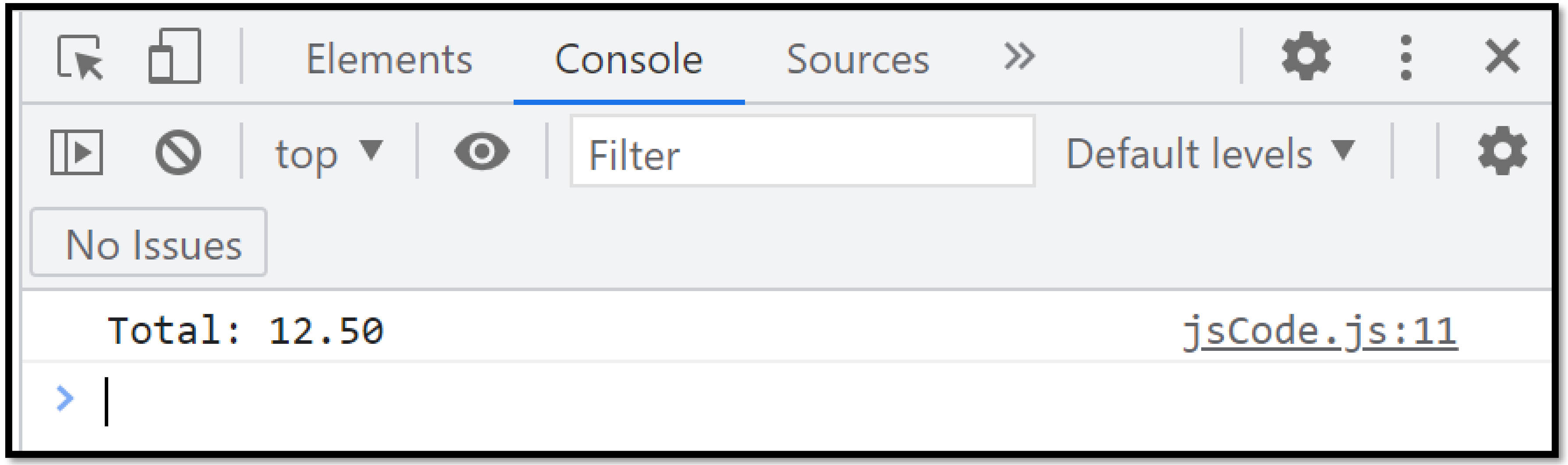


```
Welcome Christophoros, Christophorou!                jsCode.js:11
```

JavaScript Strings: Template Literals

- Expression Substitutions

```
let price = 10;
let VAT = 0.25;

let total = `Total: ${(price * (1 + VAT)).toFixed(2)}`;
console.log(total)
```



```
Total: 12.50
```

JavaScript Functions

- A **JavaScript function** is a **block of code** designed to perform a particular task. A JavaScript function is executed when "something" invokes it (calls it).
- One way of **defining a JavaScript function** is with the **following syntax**:

```
function name(p1, p2) {
  // code to be executed
  return result;
}
```

- A JavaScript function is defined with the **function** keyword, followed by a **name**, followed by **parentheses ( )**.
- The **parentheses** may include **parameter names separated by commas**: **(*p1, p2, ...*)**
- The **code to be executed** by the function is **placed inside curly brackets**: **{ }**
- Finally, the function might need to **return the result**.

JavaScript Functions

- The code inside the function will execute when **"something" invokes (calls)** the function. For example:
  - **When an event occurs** (when **a user clicks a button**; see the previous example)
  - When it is **invoked (called)** from other **JavaScript code**

Function Parameters

- Function **parameters** are the **names listed** in the **function definition in the parenthesis.**
- JavaScript function definitions **DO NOT specify data types for parameters.**

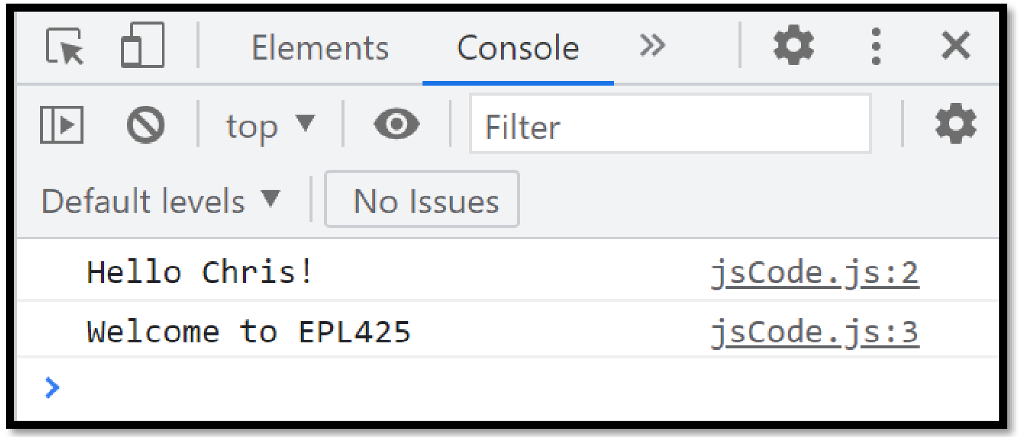
```
function name(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

JavaScript Code Execution

- There is **NO "main method".** The script file is **executed** from **top to bottom**.
- There's **NO compilation** **by the developer**.
- JavaScript is **compiled** and **executed** (***Interpreted***) **on the fly by the browser**.

jsCode.js

```
function hello(onoma) {
    console.log('Hello ' + onoma + '!');
    console.log('Welcome to EPL425’);
}

let onoma = prompt("What is your name?")
hello(onoma);
```



```
Hello Chris!                    jsCode.js:2
Welcome to EPL425               jsCode.js:3
```

JavaScript Code Execution

- **Q**: Does the following also works?

jsCode.js

```
let onoma = prompt("What is your name?")
hello(onoma);

function hello(onoma) {
    console.log('Hello ' + onoma + '!');
    console.log('Welcome to EPL425');
}
```

A: Yes, for this particular JavaScript syntax!

*This works because function declarations are "**hoisted**". Hoisting refers to the process whereby the interpreter appears **to move the declaration** of **functions**, **variables** or **classes** **to the top of their scope**, **prior to execution of the code**.*

JavaScript Functions Variables

- **Variables declared within a JavaScript function**, become **Local to the function**. Local variables **can only be accessed from within the function**.

```
// code here can NOT use carName

function myFunction() {
    let carName = "Volvo";

    // code here CAN use carName
}

// code here can NOT use carName
```

JavaScript Arrays

Note 1: It is a **common practice** to **declare arrays** with the **const** keyword.

- **Syntax** of **creating** an Array:

```
const array_name = [item1, item2, ...];
```

- **Examples** of **creating** an Array:

```
const cars = ["Saab", "Volvo", "BMW"];
```

```
const cars = [
  "Saab",
  "Volvo",
  "BMW"
];
```

Note 2: Spaces and line breaks are not important. A declaration can span multiple lines

```
const cars = [];
cars[0]= "Saab";
cars[1]= "Volvo";
cars[2]= "BMW";
```

Note 3: You can also create an array, and then provide the elements

JavaScript Arrays

Note: The elements in a JavaScript array can be of different types

```
const myArray = ["Chris", "Christophorou", 25];
```

✓

JavaScript Arrays

- You **access** an **array element** by referring to the **index number**:

```
const cars = ["Saab", "Volvo", "BMW"];
let car = cars[0];
cars[0] = "Opel";
```

- The **real strength** of **JavaScript arrays** are the **built-in array properties and methods**:

```
cars.length              // Returns the number of elements
cars.sort()              // Sorts the array
cars.push("Audi");       // Adds a new element (Audi) to array cars
```

JavaScript Array Methods

- **.toString()**: returns a string with array values separated by commas. This method **does not change** the original array.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
console.log(fruits.toString());
```

Result:

```
Banana,Orange,Apple,Mango
```

JavaScript Array Methods

- **.join(..)**: similar to **toString()** but in addition you can specify a **separator**.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
console.log(fruits.join(" * "));
```

Result:

```
Banana * Orange * Apple * Mango
```

JavaScript Array Methods

- **.pop()**: **removes** the **last** **element** from an array. Returns the value that was "popped out"

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fruit = fruits.pop();
console.log(fruit);
console.log(fruits);
```

Result:

```
"Mango"
["Banana", "Orange", "Apple"]
```

JavaScript Array Methods

- **.push()**: **adds** a **new element** to an array (**at the end**). The **push()** method **returns** the **new array length**:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let length = fruits.push("Kiwi");
console.log(fruits);
console.log(length);
```

Result:

```
["Banana", "Orange", "Apple", "Mango", "Kiwi"]
5
```

JavaScript Array Methods

- **.shift()**: **removes** the **first array element** and **"shifts" all other elements to a lower index**. Returns the value that was "shifted out"

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fruit = fruits.shift();
console.log(fruits);
console.log(fruit);
```

Result:

```
["Orange", "Apple", "Mango"]
"Banana"
```

JavaScript Array Methods

- **.unshift()**: **adds** a **new element** to an array (**at the beginning**), and "unshifts" older elements one index to the right. Returns the new array length.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let length = fruits.unshift("Lemon");
console.log(fruits);
console.log(length);
```

Result:

```
["Lemon", "Banana", "Orange", "Apple", "Mango"]
5
```

JavaScript Array Methods

- **.indexOf()**: Find the first index of the specified element. Returns **-1** if the **value is not found**.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let index = fruits.indexOf("Apple");
console.log(index);
```

Result: `2`

- You can also specify the index from which you want to start searching

```
const fruits = ["Banana", "Orange", "Apple", "Mango", "Apple"];
let index = fruits.indexOf("Apple", 3);
console.log(index);
```

Result: `4`

JavaScript Array Methods

- For **removing** an Element **by index** you can use the **delete** operator.
- However, the **delete operator** deletes the array element at the specified index, **but DOES NOT UPDATE the length of the array**, and the **value at that index** of the array **will be undefined**.

```
const myArray = [1, 2, 3, 4, 5];
delete myArray[1];
console.log(`myArray values: ${myArray}`);
console.log(`Index 1 value: ${myArray[1]}`);
```

Result:

```
myArray values: 1,,3,4,5
Index 1 value: undefined
```

Note: To avoid the aforesaid, use the splice() method instead!

JavaScript Array Methods

- **.splice()**: The **splice()** method **adds and/or removes** array elements. **Returns** an array containing the removed items (if any).
- **In contrast** with **delete operator, .splice() removes** the element(s) by index **without leaving undefined holes** in the array.

Syntax:

```
array.splice(index, howmany, item1, ....., itemX)
```

Parameter	Description
index	**Required**. The position to start adding/removing items. Negative value defines the position from the end of the array.
howmany	**Optional.** Number of items to be removed. **If not included** all the items from that index and after will be deleted.
item1, ..., itemX	**Optional.** New elements(s) to be added.

JavaScript Array Methods

- **Example** of **removing elements from an array using .splice():** Takes only **two arguments**, the **index of the element you wish to remove** and the **number of items to be removed**.
- **Creates** a **new array** that stores all the values that were removed from the original array. The **original array will no longer contain the values removed**, and **its length** will be **updated**.

```
const myArray = [1, 2, 3, 4, 5];
const x = myArray.splice(1, 1);
console.log(`myArray values: ${myArray}`);
console.log(`variable x value: ${x}`);
```

Result:

```
myArray values: 1,3,4,5
variable x value: 2
```

JavaScript Array Methods

- **Example** of **adding elements** to an array using **.splice():** Takes the **position/index to start adding items**, the **number of items to be removed** (**if any**; you **have to include this as well for the method to work correctly**), and the **new item(s) you want to include.**

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 0, "Lemon", "Kiwi");
console.log(`myArray values: ${fruits}`);
```

Result:

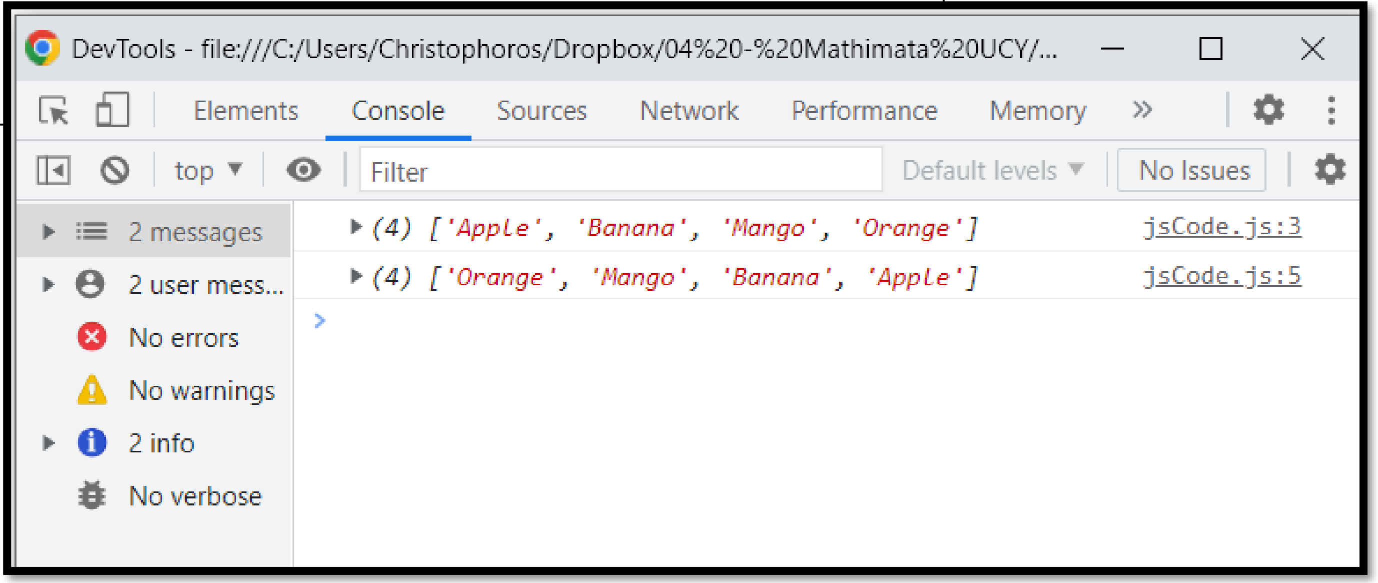
```
myArray values: Banana,Orange,Lemon,Kiwi,Apple,Mango
```

JavaScript Array Methods

- **.sort(): sorts the elements** of an array in **alphabetical** and **ascending** order
- **.reverse(): reverses** the order of the elements in an array.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
console.log(fruits);
fruits.reverse()
console.log(fruits);
```

Result:



```
▸ (4) ['Apple', 'Banana', 'Mango', 'Orange']      jsCode.js:3
▸ (4) ['Orange', 'Mango', 'Banana', 'Apple']      jsCode.js:5
```

JavaScript Array Methods

- **Note: .sort()** sorts the elements of an array in **alphabetical** and **ascending** order. **By default**, the **sort()** function **sorts values as strings**.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
console.log(fruits.sort());
```

Result:

```
['Apple', 'Banana', 'Mango', 'Orange']
```

JavaScript Array Methods

- **Sorting values as strings, works well for strings** ("Apple" comes before "Banana").
- **However**, **if numbers are sorted as strings**, "25" is bigger than "100", because "2" is bigger than "1".
- Because of this, the .**sort()** method will **produce incorrect result when sorting numbers**.

```
const numbers = [10, 5, 6, 20];
console.log(numbers.sort());
```

Result:

```
[10, 20, 5, 6]
```

JavaScript Array Methods

You can fix this by providing a **compare function**.

For more details see this link

```
const numbers = [10, 5, 6, 20];
numbers.sort( function(a, b){return a - b} );
console.log(numbers);
```

Result:

```
[5, 6, 10, 20]
```

JavaScript Array Methods

- **.reverse()**: reverses the order of the elements in an array.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
console.log(fruits.reverse());
```

Result:
```
['Mango', 'Apple', 'Orange', 'Banana']
```

```
const numbers = [23, 34, 1, 6, 78];
console.log(numbers.reverse());
```

Result:
```
[78, 6, 1, 34, 23]
```

Note: reverse() works well for both numbers and strings

JavaScript Objects

- In JavaScript, **objects** are **KING**. If you **understand objects**, you **understand JavaScript**.
- **An object** has **properties** (data fields/values) and **behaviour** (methods)!
- The **properties** are **written** as **name:value pairs**. **name** and **value** are **separated by** a **colon :**
- **Properties are separated** by a **comma ,**

JavaScript Objects

- The **following code** **creates** a **JavaScript** **object** and assigns the values **Chris**, **44**, and **blue** to an object **variable named** **person**:

```
const person = {firstName:"Chris", age:44, favColor:"blue"};
```

Note: It is a common practice to **declare objects** with the **const** keyword.

JavaScript Objects

- **Spaces** and **line breaks** are **not important**. An object definition can **span multiple lines**:

```
const person = {
   firstName:"Chris",
   age:44,
   favColor:"blue"
};
```

The **properties** are **written** as **name:value pairs**. **name** and **value** are **separated by** a **colon :**

Properties are separated by a **comma ,**

No comma is needed after the last property

JavaScript Objects

- You can **access object properties** in **two ways**:
  - ***objectName.propertyName***

    e.g., *person.firstName*

  - ***objectName["propertyName"]***

    e.g., *person["firstName"]*

JavaScript Objects: Nested Objects

- **Values in an object** can be **another object**:

```
const person = {
   name:"Chris",
   age:44,
   cars: {
      car1:"Ford",
      car2:"BMW",
      car3:"Fiat"
   }
}
```

You can access nested objects using the **dot . notation**:

```
person.cars.car2;
```

JavaScript Objects

- **Methods** are **actions** that **can be performed** on objects.

```
const person = {
    firstName:"Chris",
    lastName:"Christophorou",
    age:44,
    favColor:"blue",

    fullname: function(){
                return this.firstName + " " + this.lastName;
            },

    getOlder: function(years){
        this.age += years;
    }
}
```

Note1: All properties must be separated using comma ,

Note 2: In JavaScript, the **this** keyword refers to the **object**.

Note 3: A JavaScript method is defined as a property containing a function definition.

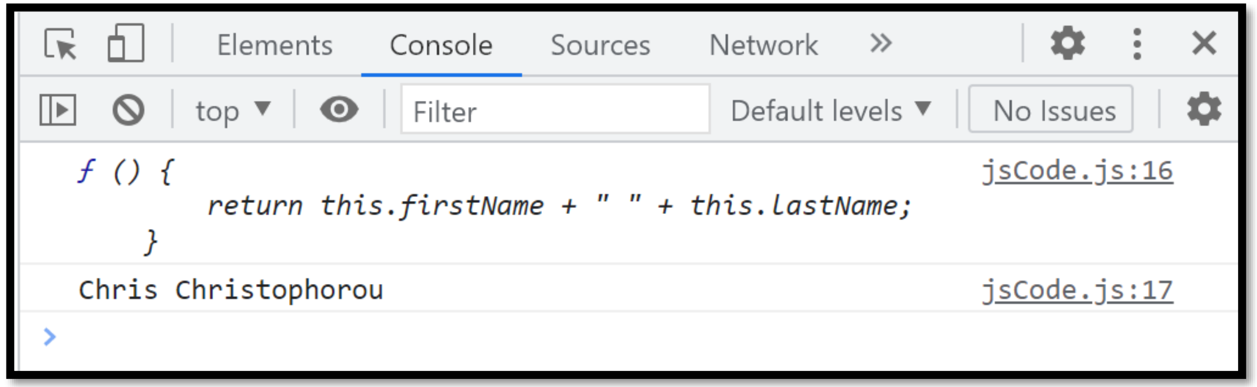
JavaScript Objects

- You **access** an **object method** with the following syntax:
  - *objectName.methodName()*

```
let name = person.fullname()
person.getOlder(5);
```

If you **access** the **fullName** **property**, **without the parenthesis** (), it **will return** the **function definition**:

```
console.log(person.fullname)
console.log(person.fullname())
```



```
ƒ () {                                                    jsCode.js:16
        return this.firstName + " " + this.lastName;
    }
Chris Christophorou                                       jsCode.js:17
>
```

JavaScript Objects

- **Properties can also** be **added** and **deleted**.
  - *You can **add new properties** to an **existing object** by just **giving it a value.***

```
person.nationality = "English";
```

```
person["favCarColor"] = "White";
```

```
person.speak = function(textToSpeak){
    return textToSpeak;
}
```

  - *You can **delete properties** of an **existing object** using the **delete** keyword.*

```
delete person.age;
```

```
delete person["favColor"];
```

Creating a JavaScript Object

- With **JavaScript**, you can **define** and **create your own objects**. There are **different ways** to **create new objects**:
  - **Create** a **single object**, using an **object literal**.
  - **Create** a **single object**, with the keyword **new Object()**.
  - **Define** an **object constructor**, **and then create objects** of the **constructed type**.
  - **Create** an **object** using **Object.create()**

Creating a JavaScript Object

Using an object literal:

- This is the **easiest way** to **create** a **JavaScript object**.
- Using an **object literal**, you **both define** and **create** an **object** in **one statement**.
- An **object literal** is a **list** of **name:value pairs** (like age:50) **inside curly braces { }**

Creating a JavaScript Object

Examples:

```
  const person = {firstName:"Chris", age:44, favColor:"blue"};
```

```
const person = {
   firstName:"Chris",
   age:44,
   favColor:"blue"
};
```

This following example **creates** an **empty JavaScript object** (named person), and **then adds 3 properties**

```
const person = {};
person.firstName = "Chris";
person.age = 44;
person.favColor = "blue";
```

Creating a JavaScript Object

Using the JavaScript Keyword new Object():

The following example creates a **new** JavaScript object using new Object(), and then adds 3 properties.

```
const person = new Object();
person.firstName = "Chris";
person.age = 44;
person.favColor = "blue";
```

Note: There is **NO NEED** to **use new Object() statement. For readability, simplicity** and **execution speed, use** the **object literal** method.

Creating a JavaScript Object

Define an object constructor, and then create objects of the constructed type:

- The **previous ways** of constructing an object, **only create single objects.**
- Sometimes we need a **"blueprint"** for **creating many objects** of the **same "type".**
- The way to create an **"object type"**, is to use an **object constructor function**.

Creating a JavaScript Object

- In the example below, **function** Person() is an **object constructor function**.

Note: After the object is created ***new properties can be added*** *or* ***existing properties*** *can be* ***deleted****.*

```
function Person(first, age, color) {
  this.firstName = first;
  this.age = age;
  this.favColor = color;
}
```

- **Objects** of the **same type** are **created** by **calling** the **constructor function** with the **new keyword**.

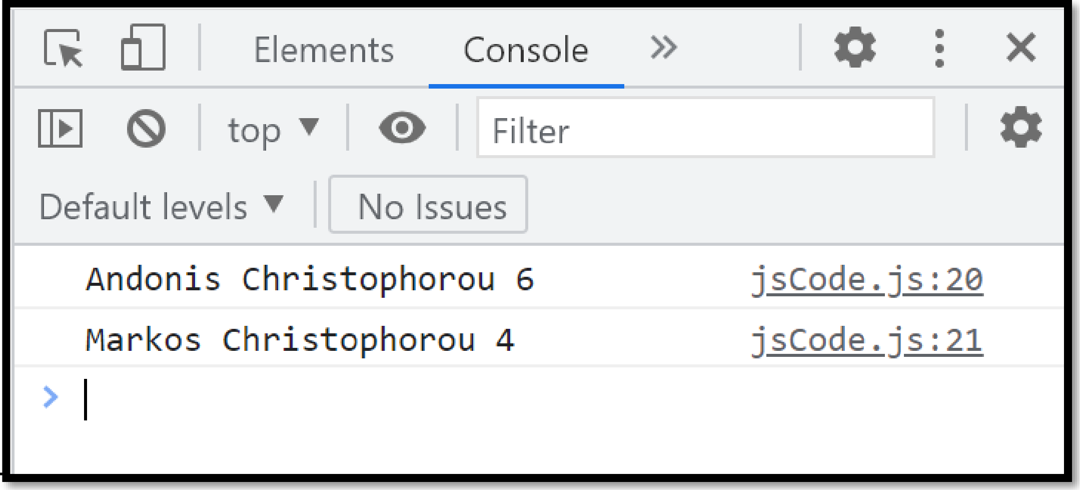
```
const mySon1 = new Person("Andonis", 5, "blue");
const mySon2 = new Person("Markos", 3, "green");
```

Creating a JavaScript Object

- Your **constructor function** can also **define methods**:

```
function Person(first, last, age, favcolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.favColor = favcolor;

    this.greeting = function(){
        return this.firstName + " " + this.lastName + " " + this.age;
    };

    this.getOlder = function(years){
        this.age += years;
    };
}

const mySon1 = new Person("Andonis", "Christophorou", 5, "blue");
const mySon2 = new Person("Markos", "Christophorou", 3, "green");
mySon1.getOlder(1);
mySon2.getOlder(2);
console.log(mySon1.greeting());
console.log(mySon2.greeting());
```

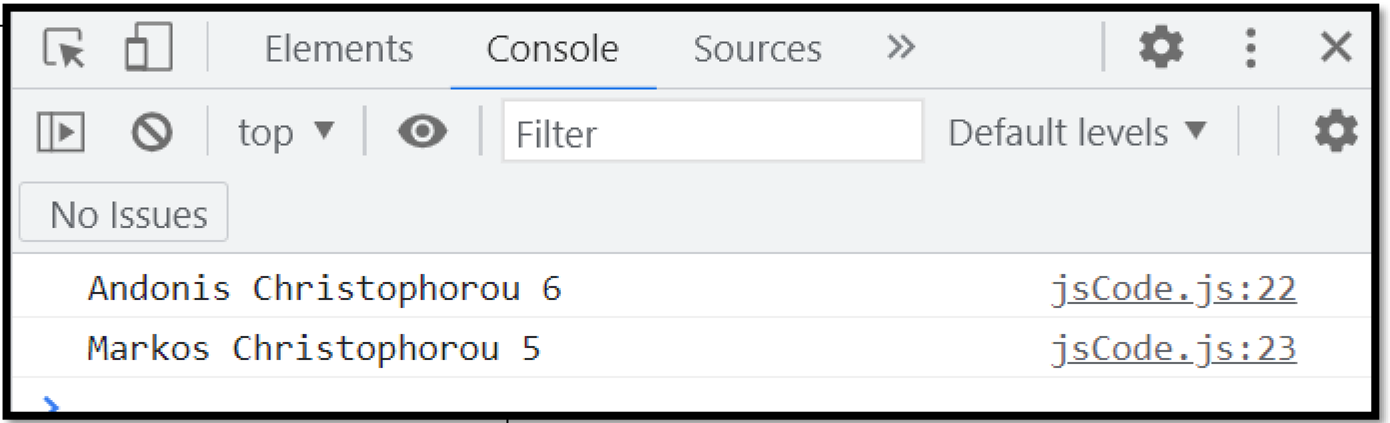
Note: In a constructor function we use a different way to define methods.



```
Andonis Christophorou 6                jsCode.js:20
Markos Christophorou 4                 jsCode.js:21
```

Creating a JavaScript Object using a class

```
class Person {
    constructor(first, last, age, favcolor){
        this.firstName = first;
        this.lastName = last;
        this.age = age;
        this.favColor = favcolor;
    }

    greeting() {
        return this.firstName + " " + this.lastName + " " + this.age;
    };

    getOlder(years) {
        this.age += years;
    };
}

const mySon1 = new Person("Andonis", "Christophorou", 5, "blue");
const mySon2 = new Person("Markos", "Christophorou", 3, "green");
mySon1.getOlder(1);
mySon2.getOlder(2);
console.log(mySon1.greeting());
console.log(mySon2.greeting());
```



```
Andonis Christophorou 6                jsCode.js:22
Markos Christophorou 5                 jsCode.js:23
```

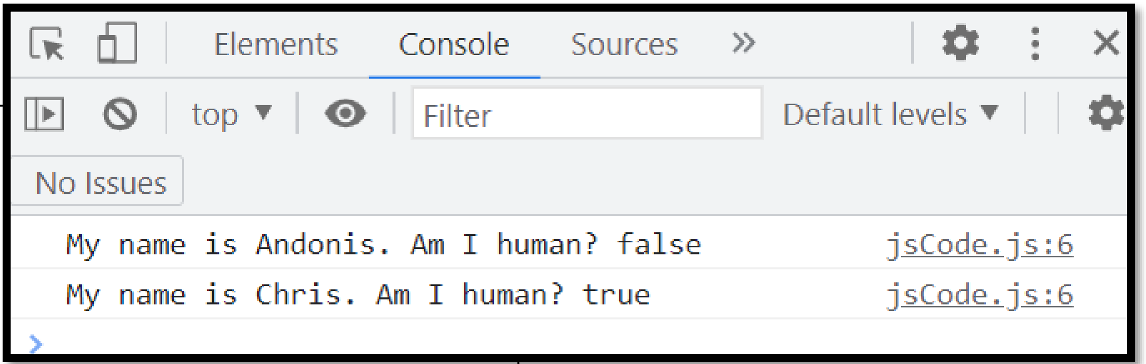
With this approach we can achieve the same results but with much cleaner syntax.

Creating a JavaScript Object

Using Object.create():

- The **Object.create()** static method creates a new object, using **an existing object as the prototype** of the newly created object.

```
const person = {
    isHuman: false,
    name: "Andonis",

    printIntroduction: function () {
        console.log(`My name is ${this.name}. Am I human? ${this.isHuman}`);
    }
};

const me = Object.create(person);
me.printIntroduction();

me.name = 'Chris'; // Inherited properties can be overwritten
me.isHuman = true; // Inherited properties can be overwritten
me.printIntroduction();
```



```
My name is Andonis. Am I human? false     jsCode.js:6
My name is Chris. Am I human? true        jsCode.js:6
```

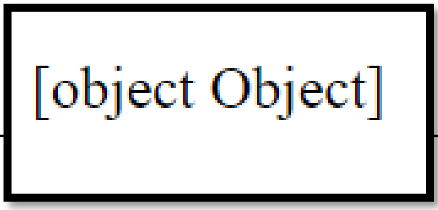
Displaying a JavaScript Object on the browser

- **Displaying** a JavaScript object in the HTML **using its variable name** (i.e., mySon1) will output **[object Object]**.

```
<!DOCTYPE html>
<html>

<head>
    <script src="JS/jsCode.js" defer></script>
    <title>Testing JavaScript</title>
</head>

<body>

    <p id="personPar"></p>

</body>

</html>
```

JS/jsCode.js

```
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
}

const mySon1 = new Person("Andonis", "Christophorou", 5, "blue");
document.getElementById("personPar").innerHTML = mySon1;
```



```
[object Object]
```

Displaying a JavaScript Object on the browser

- Some **common solutions** to **display JavaScript objects** on the **browser** are:
  - Displaying the **object properties by name**
  - Displaying the **object properties in a Loop**
  - Displaying the **object** using **Object.values()**
  - Displaying the **object** using **JSON.stringify()**

Displaying a JavaScript Object on the browser

- Displaying the Object Properties **by name**

```
const mySon1 = new Person("Andonis", "Christophorou", 5, "blue");
document.getElementById("personPar1").innerHTML =
        `${mySon1.firstName} ${mySon1.lastName} ${mySon1.age} ${mySon1.favColor}`;
```

Displaying a JavaScript Object on the browser

- Displaying the Object Properties **in a Loop**

```
const mySon1 = new Person("Andonis", "Christophorou", 5, "blue");

let text = '';
for (let prop in mySon1) {
    text += mySon1[prop] + " ";
};

document.getElementById("personPar2").innerHTML = text;
```

Note: This is like a for each loop. In every loop, **prop variable stores** the **property name as a string**. E.g. "firstName", "lastName", "age", "favColor"

Note 1: In the loop you must access the **property** using **objectName["propertyName"]!!! The reason is that prop is a variable name used in loop** for **storing** the **propertyName as a string!** Thus, **using mySon1.prop way** to **access the property will not work**!!!

Note 2: This approach will **display also the functions (**if any**) of the** JavaScript **object**, in case a function Constructor is used! If we **use a class**, the functions **will not be displayed.**

Displaying a JavaScript Object on the browser

- Displaying the Object using **Object.values()**

```
const mySon1 = new Person("Andonis", "Christophorou", 5, "blue");
const arrayDetails = Object.values(mySon1);
document.getElementById("personPar3").innerHTML = arrayDetails;
```

Note 1: Object.values(mySon1) will **return** a **JavaScript array with the properties' values**, **ready to be displayed**

Note 2: This approach will **display also the functions (**if any**) of the** JavaScript **object**, in case a function Constructor is used! If we **use a class**, the functions **will not be displayed.**

Note 3: Object.getOwnPropertyNames(mySon1) returns a **JavaScript array with the properties' names**, **ready to be displayed**
e.g., Object.getOwnPropertyNames(mySon1) **returns** ['firstName', 'lastName', 'age', 'eyeColor']

Displaying a JavaScript Object on the browser

- Displaying the Object using **JSON.stringify()**

```
const mySon1 = new Person("Andonis", "Christophorou", 5, "blue");
let stringDetails = JSON.stringify(mySon1);
document.getElementById("personPar4").innerHTML = stringDetails;
```

- **Any JavaScript object can be stringified** (converted to a string) **with** the JavaScript function **JSON.stringify()**

Displaying a JavaScript Object on the browser

- JSON.stringify() **will not stringify** the **object's functions** (**only the properties** of an object are **stringified**)
- The result **will be a string following** the **JSON format**

Example of JSON format:

```
{"firstName":"Andonis","lastName":"Christophorou","age":5,"favColor":"blue"}
```

Displaying a JavaScript Object on the browser – All ways in one example.

In this case a function constructor is used.

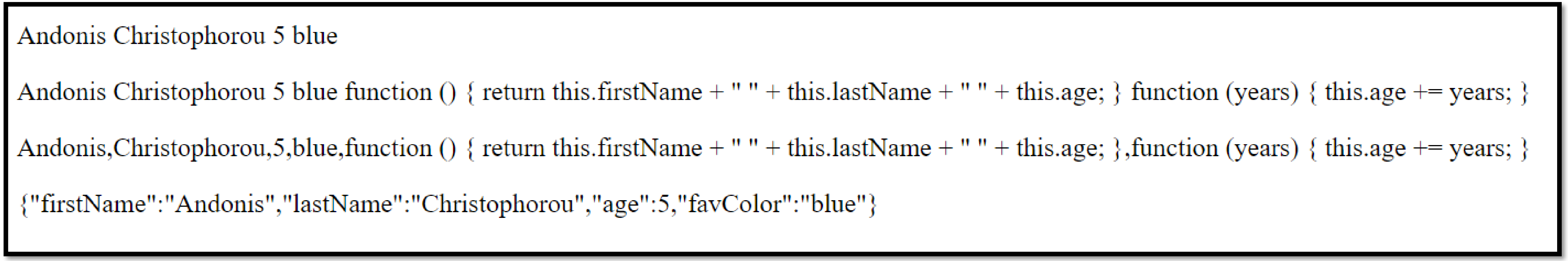
```html
<!DOCTYPE html>
<html>

<head>
    <script src="JS/jsCode.js" defer></script>
    <title>Testing JavaScript</title>
</head>

<body>

    <p id="personPar1"></p>
    <p id="personPar2"></p>
    <p id="personPar3"></p>
    <p id="personPar4"></p>

</body>

</html>
```

jsCode.js

```javascript
function Person(first, last, age, favcolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.favColor = favcolor;

    this.greeting = function () {
        return this.firstName + " " + this.lastName + " " + this.age;
    };

    this.getOlder = function (years) {
        this.age += years;
    };
}

const mySon1 = new Person("Andonis", "Christophorou", 5, "blue");

document.getElementById("personPar1").innerHTML =
    `${mySon1.firstName} ${mySon1.lastName} ${mySon1.age} ${mySon1.favColor}`;

let text = '';
for (let prop in mySon1) {
    text += mySon1[prop] + " ";
};
document.getElementById("personPar2").innerHTML = text;

const arrayDetails = Object.values(mySon1);
document.getElementById("personPar3").innerHTML = arrayDetails;

let stringDetails = JSON.stringify(mySon1);
document.getElementById("personPar4").innerHTML = stringDetails;
```

Displaying a JavaScript Object on the browser – All ways in one example.

In this case a function constructor is used.



```
Andonis Christophorou 5 blue

Andonis Christophorou 5 blue function () { return this.firstName + " " + this.lastName + " " + this.age; } function (years) { this.age += years; }

Andonis,Christophorou,5,blue,function () { return this.firstName + " " + this.lastName + " " + this.age; },function (years) { this.age += years; }

{"firstName":"Andonis","lastName":"Christophorou","age":5,"favColor":"blue"}
```

This will be displayed on the browser!!!

Displaying a JavaScript Object on the browser – All ways in one example.

In this case a class is used.

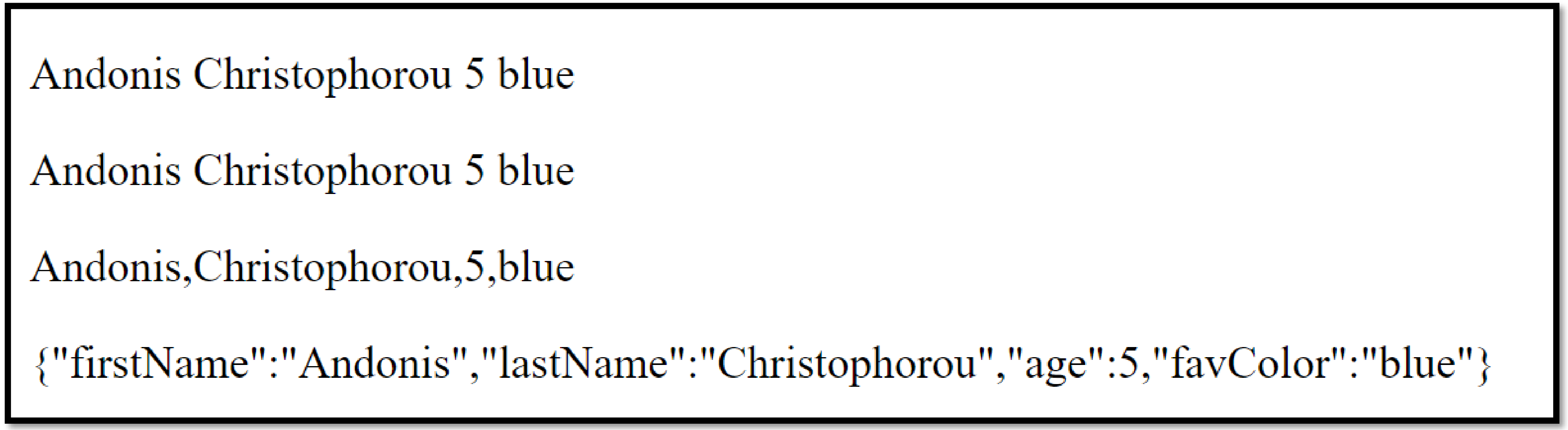
```html
<!DOCTYPE html>
<html>

<head>
    <script src="JS/jsCode.js" defer></script>
    <title>Testing JavaScript</title>
</head>

<body>

    <p id="personPar1"></p>
    <p id="personPar2"></p>
    <p id="personPar3"></p>
    <p id="personPar4"></p>

</body>

</html>
```

jsCode.js

```javascript
class Person {
    constructor(first, last, age, favcolor) {
        this.firstName = first;
        this.lastName = last;
        this.age = age;
        this.favColor = favcolor;
    }

    greeting() {
        return this.firstName + " " + this.lastName + " " + this.age;
    };

    getOlder(years) {
        this.age += years;
    };
}

const mySon1 = new Person("Andonis", "Christophorou", 5, "blue");

document.getElementById("personPar1").innerHTML =
    `${mySon1.firstName} ${mySon1.lastName} ${mySon1.age} ${mySon1.favColor}`;

let text = '';
for (let prop in mySon1) {
    text += mySon1[prop] + " ";
};
document.getElementById("personPar2").innerHTML = text;

const arrayDetails = Object.values(mySon1);
document.getElementById("personPar3").innerHTML = arrayDetails;

let stringDetails = JSON.stringify(mySon1);
document.getElementById("personPar4").innerHTML = stringDetails;
```

Displaying a JavaScript Object on the browser – All ways in one example.

In this case a class is used.



```
Andonis Christophorou 5 blue

Andonis Christophorou 5 blue

Andonis,Christophorou,5,blue

{"firstName":"Andonis","lastName":"Christophorou","age":5,"favColor":"blue"}
```

This will be displayed on the browser!!!

JavaScript Object Notation (JSON)

- JSON stands for **J**ava**S**cript **O**bject **N**otation and is a **format** (**text only**) for **storing** and **transporting** **data**.
- JSON is **language independent** and is **often used** when **data** is sent **from a server to a web page, or vice versa (used with AJAX)**.

A JSON example:

```
{
    "employees":[
    { "firstName": "John", "lastName": "Doe" },
    { "firstName": "Anna", "lastName": "Smith" },
    { "firstName": "Peter", "lastName": "Jones" } ]
}
```

JSON Syntax Rules:

- Data is in **name:value** pairs
- Data is **separated** by **commas**
- **Curly braces** hold **objects**
- **Square brackets** hold **arrays**

JavaScript Object Notation (JSON)

- The JSON format is **syntactically "almost" identical** to the **code** for **creating JavaScript objects**.
- Because of this similarity, a **JavaScript program can easily convert JSON data into native JavaScript objects (using JSON.parse(…))**

JavaScript Object Notation (JSON)

JSON Data - A Name and a Value

- JSON data is **written** as **name:value** pairs, **just like JavaScript object properties**.
- A **name:value** pair consists of a **field name** (in double quotes), followed by a **colon :**, followed by a **value**:

```
"firstName":"Chris"
```

JSON Objects

- **JSON objects** are **written inside curly braces { }**.
- Just like in JavaScript, **objects can contain multiple name:value pairs**:

```
{"firstName":"Chris", "lastName":"Christophorou"}
```

JavaScript Object Notation (JSON)

JSON Arrays

- JSON arrays are **written** inside **square brackets [ ]**.
- Just like in JavaScript, an array can contain objects:

```
"employees":[
{ "firstName": "John", "lastName": "Doe" },
{ "firstName": "Anna", "lastName": "Smith" },
{ "firstName": "Peter", "lastName": "Jones" } ]
```

- In the example above, the object "employees" is an array. It contains three objects. Each object is a record of a person (with a first name and a last name).

Converting a JSON Text to a JavaScript Object

- A common use of JSON is to **read data from a web server**, and **display** the data in a **web page (or vice versa)**.
- Since we did not talked about Back-end development yet, we will manually create a JavaScript string containing **JSON syntax...**

```
let JASONtext = '{ "employees" : [' +
'{ "firstName":"John" , "lastName":"Doe" },' +
'{ "firstName":"Anna" , "lastName":"Smith" },' +
'{ "firstName":"Peter" , "lastName":"Jones" } ]}';
```

- **... and use this** `JASONtext` **string** as input to our web page.

Note: Another way to do this is create a JavaScript array containing three objects and then use JSON.stringify()

Converting a JSON Text to a JavaScript Object

- Then, to convert the string into a JavaScript object, we will use the JavaScript built-in function **JSON.parse()**

```
const personObj = JSON.parse(JASONtext);
```

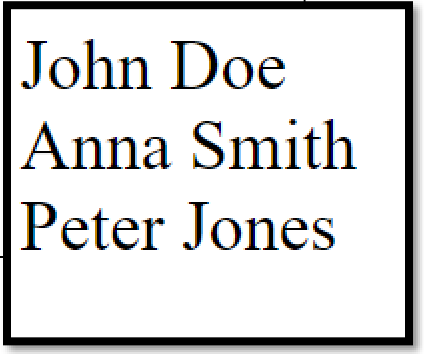
- Finally, we can use this new **personObj** JavaScript object in our page!!! See the example at the next slide!

Converting a JSON Text to a JavaScript Object

```html
<!DOCTYPE html>
<html>

<head>
    <title>Testing JavaScript JASON</title>
    <script src="JS/jsCode.js" defer></script>
</head>

<body>

    <p id="persons"></p>

</body>

</html>
```

jsCode.js

```javascript
let JASONtext = '{ "employees" : [' +
'{ "firstName":"John" , "lastName":"Doe" },' +
'{ "firstName":"Anna" , "lastName":"Smith" },' +
'{ "firstName":"Peter" , "lastName":"Jones" } ]}';

personObj = JSON.parse(JASONtext);

let text = "";
for (let i = 0; i <personObj.employees.length; i++){
    text = text + (personObj.employees[i].firstName + " "
                            + personObj.employees[i].lastName) + "<br>";
}

document.getElementById("persons").innerHTML = text;
```



```
John Doe
Anna Smith
Peter Jones
```

This will be displayed in your browser

HTML Events and JavaScript

- An **HTML event** is an **action** or **occurrence** that **happens** on **HTML Elements** or the **web page**, such as a user clicking on a button or the web page is loaded.
- HTML events **are used** to **trigger JavaScript functions**, which allows web developers to **add interactivity** and **dynamic behavior** to their web pages.
- HTML events **can be added** to **HTML elements, allowing developers** to **specify** the **type of event to listen for** and the **function to be executed** when the **event occurs**.

HTML Events and JavaScript

- **HTML events can be added** to **HTML elements in two ways:**
  - By using **event handler attributes in HTML Elements** such as the **"onclick"** attribute for a button,
  - By using **event listeners** in JavaScript (but **without** the "on" prefix); **These** are **attached to HTML elements** using the **"addEventListener"** **method (I prefer this way!!!).**

We will see later how to do these!!!

HTML Events and JavaScript

- When an **HTML event** **occurs**, the **web browser** **creates** an **event object** that **contains information** about the **event**, such as the **type of event** and **any data associated** with the event.
- This **event object** is then **caught** by the **event listener** attached to the HTML Element, and calls the associated JavaScript **function to handle the event.**

Examples of JavaScript events

- **Click event:** This event is triggered when the user clicks on an element on the web page. The attribute name for this event is **"onclick"**.
- **Mouseover event:** This event is triggered when the user moves their mouse over an element on the web page. The attribute name for this event is **"onmouseover"**.
- **Keydown event:** This event is triggered when the user presses a key on their keyboard. The attribute name for this event is **"onkeydown"**.
- **Load event**: This event is triggered when the web page finishes loading. The attribute name for this event is **"onload"**.

Examples of JavaScript events

- **Change event**: This event is triggered when the value of an input field or select element changes. The attribute name for this event is **"onchange".**
- **Focus event:** This event is triggered when an element receives focus, for example when a user clicks on an input field. The attribute name for this event is **"onfocus".**

When **JavaScript is used** in HTML pages, **JavaScript** can **"react"** on **these HTML events.**

HTML Events and JavaScript

- HTML allows **event handler attributes with JavaScript code, to be added** to **HTML elements.**
- The JavaScript code is added in quotes " "

Syntax:

```
<element event="some JavaScript code or calling JavaScript function">
```

JavaScript Events: Examples

Example 1: An onclick **event attribute**, is added to a <button> **element**. In this example, the JavaScript code changes the content of an element with id="date1".

```
<button onclick="document.getElementById('date1').innerHTML =
Date()">Show Time!</button>
```

Example 2: In the next example, the code **changes the content of its own element** (using **this**.innerHTML):

```
<button onclick="this.innerHTML = Date()">Show Time!</button>
```

JavaScript Events: Examples

- JavaScript code is often **several lines long**. It is **more common** to see **event attributes CALLING FUNCTIONS!!!**
- **We will use this approach!!!**

```
<button onclick="displayDate()">Show Time!</button>
```

JavaScript Events: Examples

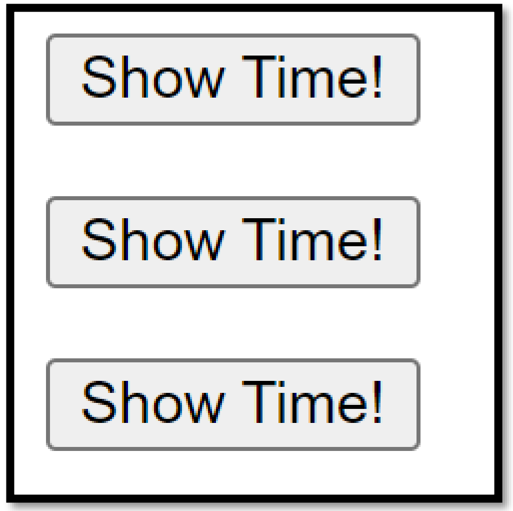
```
function displayDate(){
    document.getElementById('date2').innerHTML = Date()
}
```

jsCode.js

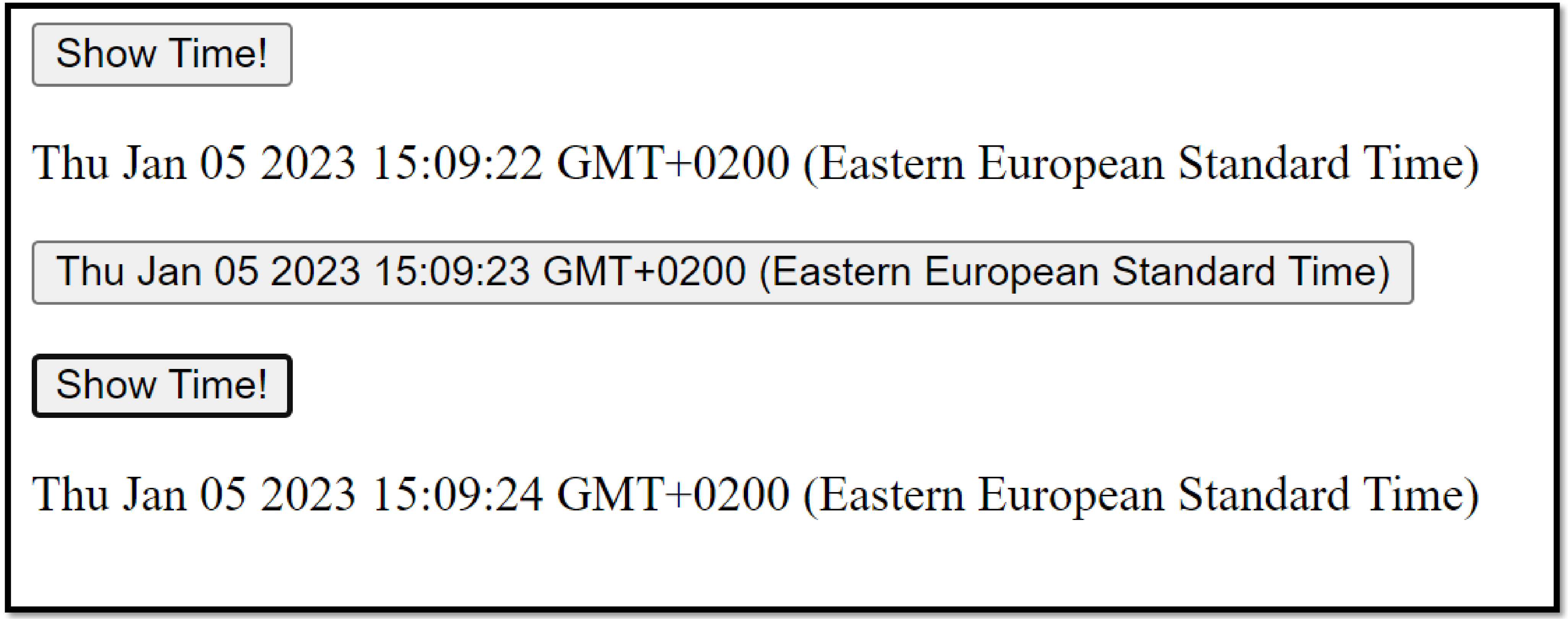
```
<!DOCTYPE html>
<html>

<head>
    <title>Testing JavaScript Events</title>
    <script src="JS/jsCode.js" defer></script>
</head>

<body>

    <button onclick="document.getElementById('date1').innerHTML=Date()">Show Time!</button>
    <p id="date1"></p>

    <button onclick="this.innerHTML = Date()">Show Time!</button> <p></p>

    <button onclick="displayDate()">Show Time!</button>
    <p id="date2"></p>

</body>

</html>
```

JavaScript Events: Examples

Before clicking the buttons



After clicking the buttons


Show Time!
Thu Jan 05 2023 15:09:23 GMT+0200 (Eastern European Standard Time)
Show Time!

Thu Jan 05 2023 15:09:22 GMT+0200 (Eastern European Standard Time)

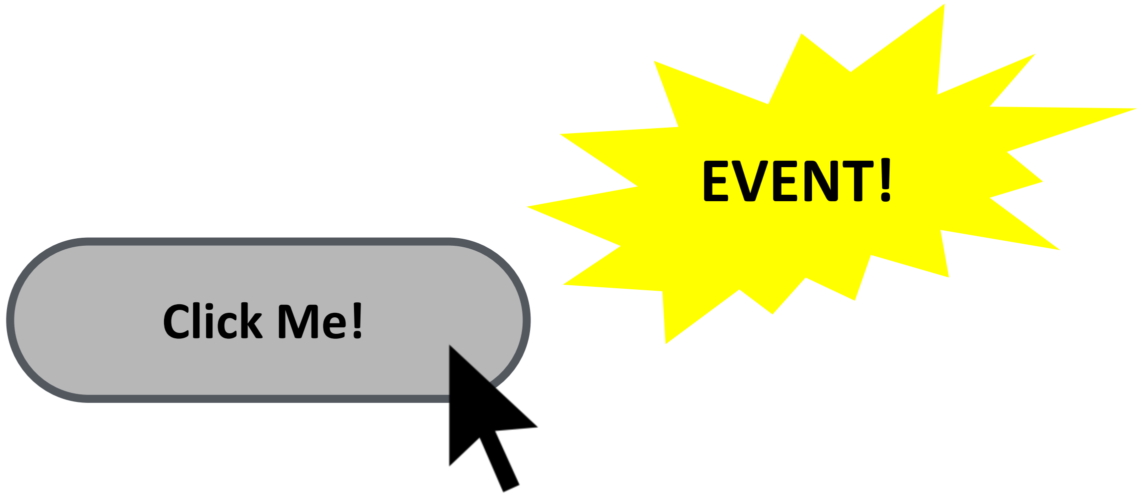
Thu Jan 05 2023 15:09:24 GMT+0200 (Eastern European Standard Time)

Event-driven programming

- **Most JavaScript** written **in the browser** is **event-driven!**
- This means that **the code does not run right away**, but it **executes after some event fires**.

Example:
Here is a UI
HTML element
(i.e., a button)
that the user can
interact with.

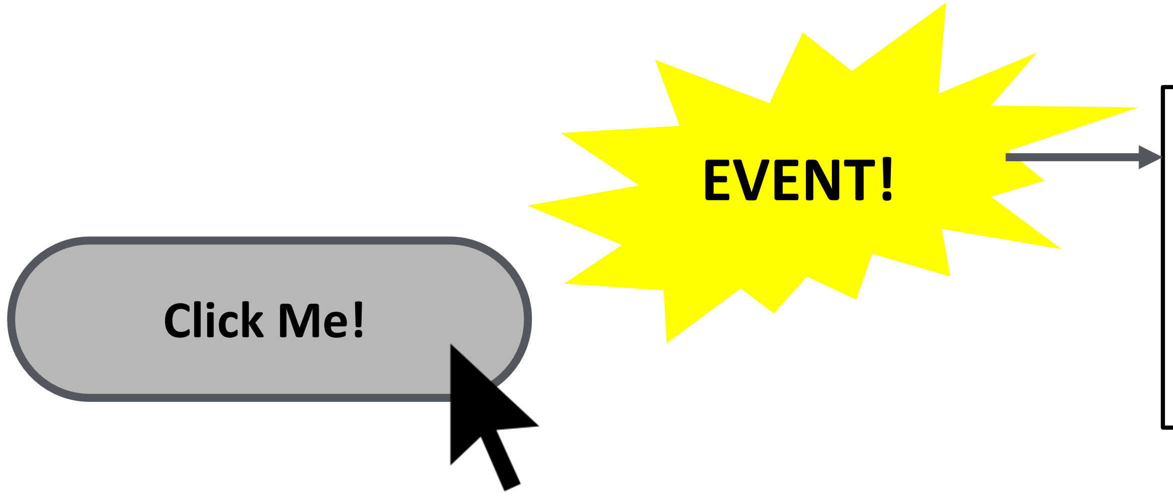
Event-driven programming



When clicked, the button **emits** an "**event**" which is **like an announcement** that an interesting thing **has occurred**.

Event-driven programming

Any function listening to **that event** now **executes**. This function is **called** an "**event handler**."



```
function onClick() {
    ...
}
```

Event-driven programming

Q: How can we access and change the elements of an HTML document using JavaScript?

A: Every element on an HTML document is accessible in JavaScript through the HTML DOM: Document Object Model

Event-driven programming

Q: What is the HTML DOM?

A: The HTML Document Object Model is a programming interface for web documents.

The DOM represents the document as nodes and objects.

That way, programming languages can interact with the page and change the document structure, style, and content.

The HTML DOM

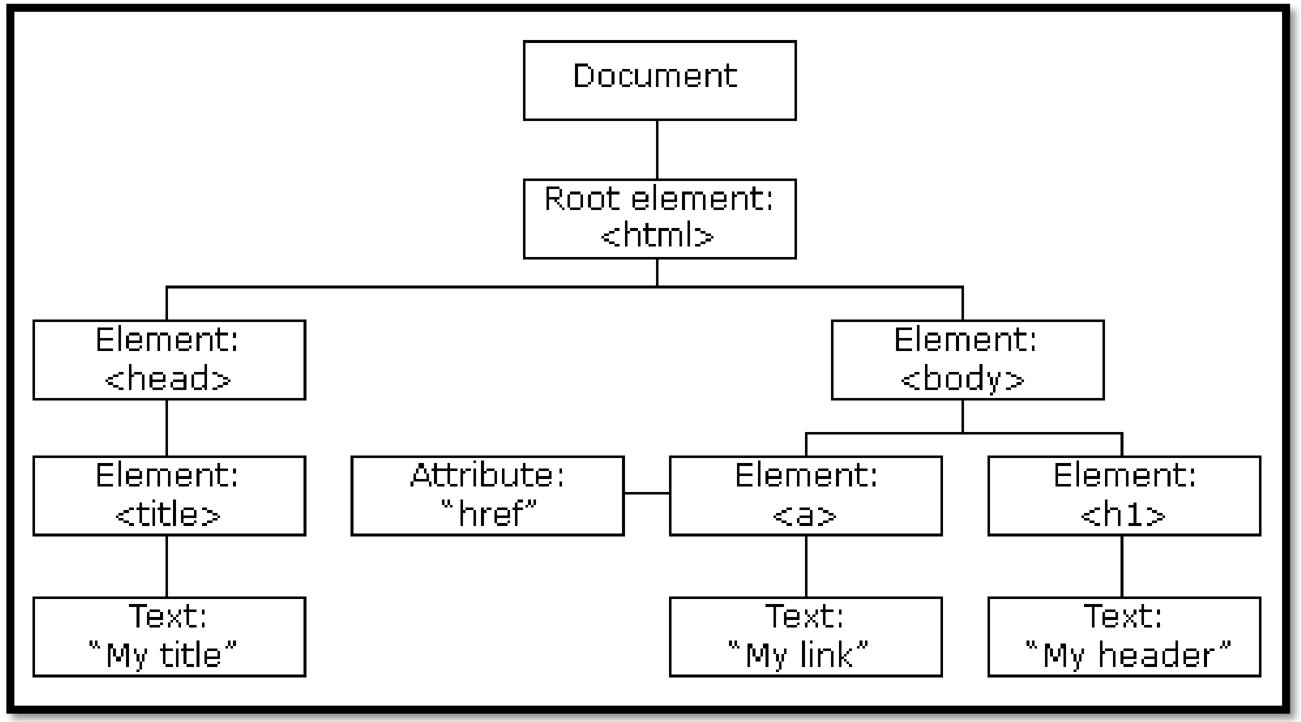
- The DOM is **NOT a programming language** and it’s **NOT a part of JavaScript**.
- It is one of the multiple **web Application Program Interfaces (APIs) built into web browsers** and **defines**:
  - The **HTML Elements as objects**
  - The **properties**, **methods** and **events for all HTML Elements**

Note: Every non-empty webpage has a DOM, even the ones that don't use any JavaScript.

The HTML DOM

- **After** the **browser** **downloads** the HTML document, it **converts its content** into a **tree like structure** called the DOM (Document Object Model) and **stores it** in **its memory**. The HTML DOM model is **constructed** as a **tree of objects**:

```
<html>

<head>
    <title>My Title</title>
</head>

<body>
    <h1>My Header</h1>
    <a href="myPage.html">My Link</a>

</body>

</html>
```



JavaScript and the HTML DOM

- The **HTML DOM** is an **API (Application Programming Interface) for JavaScript** that **gives it all the power** it needs to **create Interactive/Dynamic HTML pages**! Specifically, JavaScript can **access** and **manipulate** the DOM and:
  - **Add/change/remove HTML elements**
  - **Add/change/remove HTML attributes**
  - **Add/change/remove CSS styles**
  - **Add/change/remove HTML events**
  - **React to HTML events**

Manipulating the DOM

- **In the DOM**, all HTML elements (referred also as **nodes** in the DOM tree) **are defined** as **objects**.
- **Each "HTML element" object has its own properties,** that can be **manipulated,** and **methods** that can be **invoked** using JavaScript.
- **All** the **properties**, **methods** and **events** available for manipulating and creating web pages are **further organized** into **objects** that **we're going to call interfaces**.

Manipulating the DOM

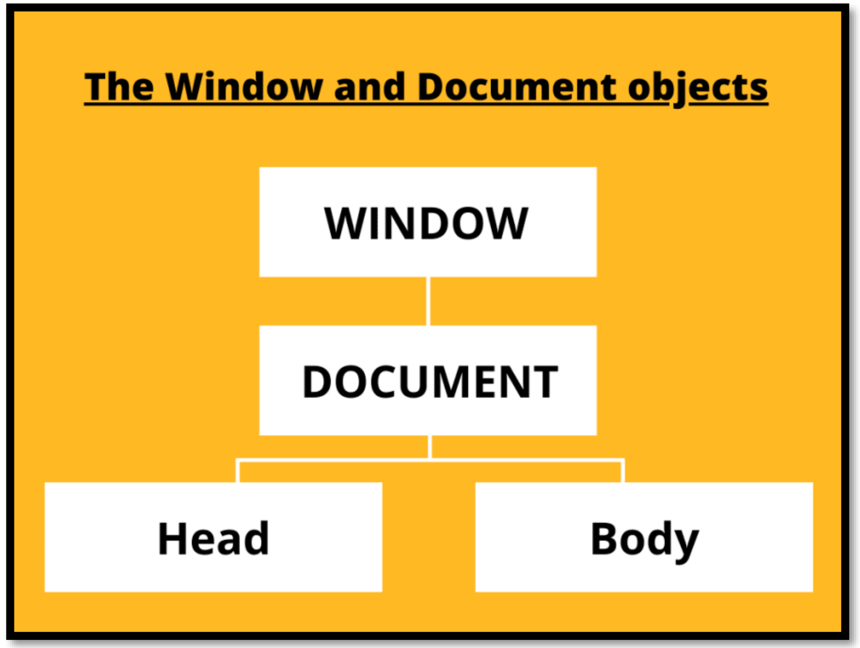
- There are many DOM interfaces working together but the ones that we're going to use most often are **window** and **document!**

A complete list of the DOM interfaces can be found here.

Manipulating the DOM

- **window** - The window interface represents a window containing a DOM document (an open window in a browser). It holds the **highest position** in the **DOM hierarchy**, as it is a parent of the **document object** and all of its children.

- **document** - The **document** interface **represents any web page loaded in the browser** and **serves** as an **entry point** into the **web page's content**, which is the DOM tree. Through this **document object** we will **select the nodes** (i.e., the HTML elements) that we want **to manipulate**.



Selecting HTML Elements

- In order to **interact with** and **manipulate** any node (HTML Element) in the DOM tree, we **first need** to **find it** and **select it**.
- We **can do this** using one of the multiple methods the DOM API offers and there are **several ways to do this**:
  - Finding HTML elements by **id**
  - Finding HTML elements by **tag name**
  - Finding HTML elements by **class name**
  - Finding HTML elements by **CSS selectors**
  - Finding HTML elements by **HTML object collections**

Note: All these methods are called on the **document object using the dot . operator**

Finding HTML elements by id

- **.getElementById():** The **easiest way** to find an HTML element in the DOM, is by using the **element id**. If the element is **found**, the method will return the **element as an object** (in element). If the element is **not found**, element will contain **null**.
- The example below finds the element with id="parag1" and **stores its reference address** in variable named **element.**
- **Through this element variable** we can now **have access,** using the dot **.** operator, to its **properties** and **methods**.

```
const element = document.getElementById("parag1");
console.log(element.innerHTML);
console.log(element.hasAttributes());
```

Finding HTML elements by Tag Name

- **.getElementsByTagName()**: Finds HTML elements based on their **tag names**. This method **returns** a live **HTMLCollection (an array-like list)** of all matching HTML elements, possibly of length 0 if no match is found.
- The following example finds all <p> HTML elements and **access** their **innerHTML attribute**.

```
const elements = document.getElementsByTagName("p");
console.log(elements[0].innerHTML)
console.log(elements[1].innerHTML)
```

Finding HTML elements by Tag Name

- You can also **search hierarchically**!!
- The following example **first finds** the element with id="div1", **and then finds** all <p> **elements contained** in it.

```
const element = document.getElementById("div1");
const pElements = element.getElementsByTagName("p");
```

Finding HTML elements by Class Name

- **.getElementsByClassName()**: **Finds HTML elements** based on their **class names**. This method **returns** a **live HTMLCollection (an array-like list)** of all matching HTML elements, possibly of length 0 if no match is found.
- The following example finds all HTML elements with class="notes" and access their innerHTML attribute.

```
const elements = document.getElementsByClassName("notes");
console.log(elements[0].innerHTML)
console.log(elements[1].innerHTML)
```

Finding HTML elements by CSS Selectors

- **.querySelector():** Finds and returns the **first HTML element** that **match** a specified **CSS selector** (id, class names, types, attributes, values of attributes, etc.). If no matches are found, **null** is returned.

- The following example finds and returns the **first element** in the **document** with the **class** = "myclass"

```
const element = document.querySelector(".myclass");
```

- The following example finds and returns the **first element** in the **document** with the **id** = "myclass"

```
const element = document.querySelector("#myclass");
```

Finding HTML elements by CSS Selectors

- **.querySelectorAll():** Finds **all HTML elements** that **match** a **specified CSS selector**. This method **returns** a **live HTMLCollection (an array-like list)** of all matching HTML elements, possibly of length 0 if no match is found.
- The following example finds and returns a list of all **<p>** elements with **class="notes"**

```
const elements = document.getElementsByClassName("p.notes");
console.log(elements[0].innerHTML)
console.log(elements[1].innerHTML)
```

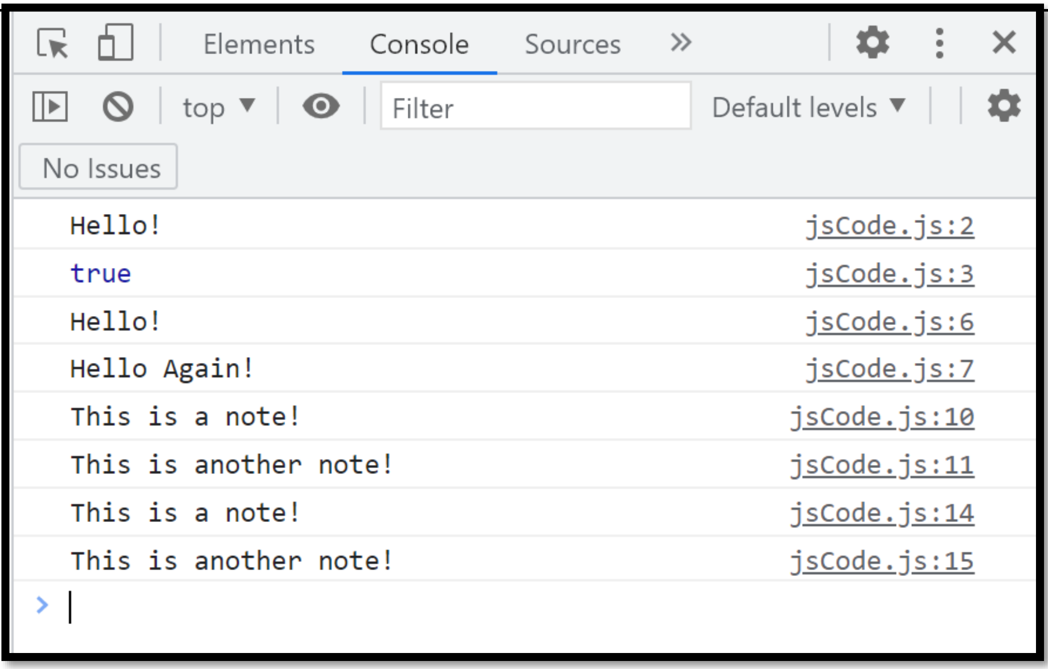
Note: For an extensive list of properties and methods that can be used on all HTML elements check this link.

Example

```html
<!DOCTYPE html>
<html>

<head>
    <title>Testing JavaScript Events</title>
    <script src="JS/jsCode.js" defer></script>
</head>

<body>

    <p id="par1" onclick="myFunction()">Hello!</p>
    <p id="par2">Hello Again!</p>
    <p id="par3" class="notes">This is a note!</p>
    <p id="par4" class="notes">This is another note!</p>


</body>

</html>
```

```javascript
const element = document.getElementById("par1");
console.log(element.innerHTML);
console.log(element.hasAttributes());

const elements1 = document.getElementsByTagName("p");
console.log(elements1[0].innerHTML)
console.log(elements1[1].innerHTML)

const elements2 =
document.getElementsByClassName("notes");
console.log(elements2[0].innerHTML)
console.log(elements2[1].innerHTML)

const elements3 = document.querySelectorAll("p.notes");
console.log(elements3[0].innerHTML)
console.log(elements3[1].innerHTML)
```

jsCode.js



```
Hello!                    jsCode.js:2
true                      jsCode.js:3
Hello!                    jsCode.js:6
Hello Again!              jsCode.js:7
This is a note!           jsCode.js:10
This is another note!     jsCode.js:11
This is a note!           jsCode.js:14
This is another note!     jsCode.js:15
```

Manipulating HTML Elements

- **Once you find the elements** in the HTML DOM you **can now have access** and **change their attributes.**
- The easiest way to **modify the content** of an HTML element is by using the **innerHTML property**. In the following examples we **add** the **JavaScript code** in the **HTML body in** <script> tag.

```
<!DOCTYPE html>
<html>
<body>
    <p id="p1">Hello EPL425!</p>
    <script>
        document.getElementById("p1").innerHTML = "Hello EPL425 People!";
    </script>
</body>
</html>
```

Manipulating HTML Elements

- You can also **change** the value of an **HTML attribute**. The following example **changes the value** of the **src attribute** of an <img> element:

```
<!DOCTYPE html>
<html>
<body>
  <img id="myImage" src="smiley.gif">
  <script>
    document.getElementById("myImage").src = "landscape.jpg";
  </script>
</body>
</html>
```

Manipulating HTML Elements

- You can also **change** the **style** of an **HTML element**. The following example changes the **text color** of a <p> **element** to blue:

```
<!DOCTYPE html>
<html>
<body>
  <p id="p2">Hello World!</p>
  <script>   document.getElementById("p2").style.color = "blue";</script>
</body>
</html>
```

Reacting to Events

- As we said before, a **JavaScript can be executed when an event occurs:** Examples of HTML events:
  - When a user clicks the mouse on an HTML element.
  - When a web page has loaded
  - When an image has been loaded
  - When the mouse moves over an HTML element
  - When an input field is changed
  - When an HTML form is submitted
  - When a user strokes a key

Adding Events to HTML elements

- To **assign events** to HTML elements you can either use **event** attributes….

```html
<!DOCTYPE html>
<html>
    <script src="JS/jsCode.js" defer></script>
<body>

<h1 onmouseover="changeText(this)" onmouseout="changeBack(this)">Click on this text!</h1>

</body>
</html>
```

```javascript
function changeText(element) {
    element.innerHTML = "This is the new Text!";
    element.style.color = "blue";
}

function changeBack(element) {
    element.innerHTML = "Click on this text!";
    element.style.color = "black";
}
```

jsCode.js

Try the code in Visual Studio Code!

Adding Events to HTML elements

For a list of all HTML DOM events, look at this complete HTML DOM Event Object Reference.

Adding Events: Using the addEventListener() method

- ….Or you can also **add event listeners using** the **addEventListener()** method.
- The **addEventListener()** method **adds** an **event handler** to an element **without overwriting existing** event handlers.
- You can **add as many event handlers** to one element **as you like.**

Adding Events: Using the addEventListener() method.

- One **advantage** of using the **.addEventListener()** method, is that the JavaScript is **separated from** the **HTML markup. This can:**
  - Provide for **better readability** and
  - **Allows** to **add event listeners even when you do not control the HTML markup**.
- You can **easily remove** an **event listener** by using the **.removeEventListener()** method.

Adding Events: Using the addEventListener() method.

Syntax:
```
element.addEventListener(event, function, useCapture);
```

- The **first parameter** is the type of the **event** (like "click" or "mousedown" or any other HTML DOM Event.)
- The **second parameter** is the **function** we want **to call** when the **event occurs**.
- The **third parameter** is a **boolean** value specifying whether to use event **bubbling** **(this is the default)** or event **capturing**. This parameter is **optional and rarely used in practice.**

Note: You **don't use the "on" prefix for the event**; For example use "click" instead of "onclick" and "mouseover" instead of "onmouseover".

Adding Events: Using the addEventListener() method.

```html
<!DOCTYPE html>
<html>
<script src="JS/jsCode.js" defer></script>

<body>

    <h1 id="h1">Click on this text!</h1>

</body>

</html>
```

Try the Code in Visual Studio Code!

Do not forget **defer**!!!

```javascript
document.getElementById("h1").addEventListener("mouseover", changeText);
document.getElementById("h1").addEventListener("mouseout", changeBack);

function changeText() {
    let element = document.getElementById("h1");
    element.innerHTML = "This is the new Text!";
    element.style.color = "blue";
}

function changeBack() {
    let element = document.getElementById("h1");
    element.innerHTML = "Click on this text!";
    element.style.color = "black";
}
```

jsCode.js

Adding Events: Using the addEventListener() method.

- If you want to also **pass parameter values**, use an "**anonymous function**" that **calls the specified function** with the **parameters**!!!

```
document.getElementById("h1").addEventListener("mouseover", function() {changeText("blue");});
document.getElementById("h1").addEventListener("mouseout", function() {changeBack("black");});

function changeText(textColor) {
    let element = document.getElementById("h1");
    element.innerHTML = "This is the new Text!";
    element.style.color = textColor;
}

function changeBack(textColor) {
    let element = document.getElementById("h1");
    element.innerHTML = "Click on this text!";
    element.style.color = textColor;
}
```

```
<!DOCTYPE html>
<html>
<script src="JS/jsCode.js" defer></script>

<body>

    <h1 id="h1">Click on this text!</h1>

</body>

</html>
```

jsCode.js

Reacting to Events

You can also **remove event handlers** that have been **attached with the addEventListener()** method using the **.removeEventListener()**

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" href="CSS/style3.css">
    <script src="JS/jsCode.js" defer></script>
    <title>Testing JavaScript</title>
</head>
<body>

<div id="myDIV">
  <p>This div element has an onmousemove event handler that displays a
random number (0 - 100) every time you move your mouse inside this orange
field.</p>
  <p><b>Click the button to stop estimating the random number!!!</b></p>

  <button type="button" id="myBtn">Stop Estimating Random</button>
</div>

<p id="par1"></p>

</body>
</html>
```

Reacting to Events

```
document.getElementById("myDIV").addEventListener("mousemove", estimateRandom);
document.getElementById("myDIV").addEventListener("click", removeHandler);

function estimateRandom() {
    document.getElementById("par1").innerHTML = Math.floor(Math.random() * 101);
                                                // Returns a random integer from 0 to 100:
}

function removeHandler() {
    document.getElementById("myDIV").removeEventListener("mousemove", estimateRandom);
}
```

jsCode.js

```
#myDIV {
    background-color: black;
    border: 1px solid;
    padding: 20px;
    color: white;
    font-size: 20px;
    text-align: center;
}
```

style3.css

Try the Code in Visual Studio Code!

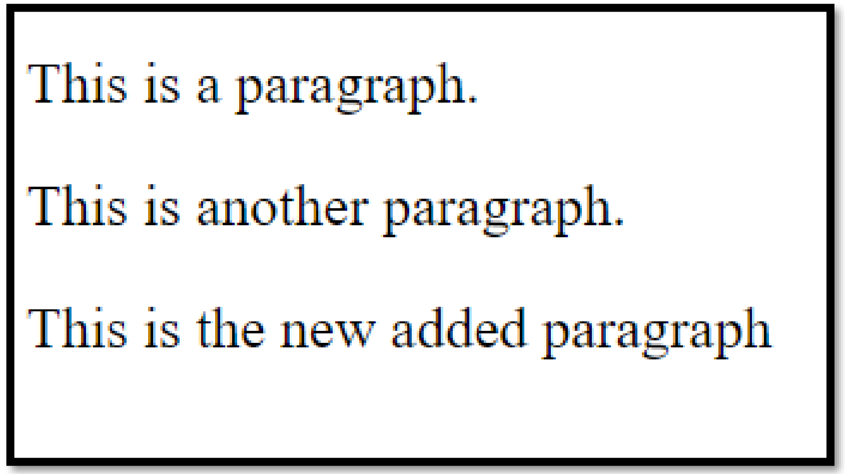
Creating new HTML Elements (Nodes)

- To **add** a **new element** to the **HTML DOM**, you must **create the element** (element node) **first**, and then **append** it to an **existing (parent) element**.

```
<!DOCTYPE html>
<html>

<head>
    <script src="JS/jsCode.js" defer></script>
    <title>Testing JavaScript</title>
</head>

<body>
    <div id="div1">
        <p id="p1">This is a paragraph.</p>
        <p id="p2">This is another paragraph.</p>
    </div>

</body>

</html>
```

jsCode.js

```
const par = document.createElement("p");
par.innerHTML = "This is the new added paragraph";
par.id = 'p3';

const element = document.getElementById("div1");
element.appendChild(par);
```


This is a paragraph.

This is another paragraph.

This is the new added paragraph

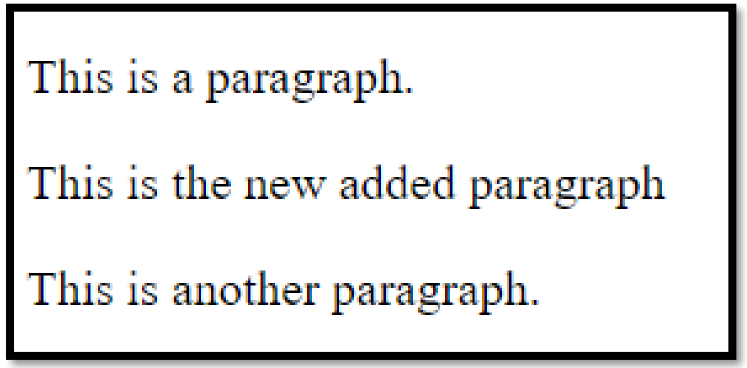
Creating new HTML Elements (Nodes)

- The **appendChild()** method in the previous example, **appended** the **new element as the last child of the parent**. If you want to **add the element** in a **specified location**, you can use the **.insertBefore()** method:

```html
<!DOCTYPE html>
<html>

<head>
    <script src="JS/jsCode.js" defer></script>
    <title>Testing JavaScript</title>
</head>

<body>
    <div id="div1">
        <p id="p1">This is a paragraph.</p>
        <p id="p2">This is another paragraph.</p>
    </div>

</body>

</html>
```

jsCode.js

```javascript
const newElement= document.createElement("p");
newElement.innerHTML = "This is the new added paragraph";
newElement.id = 'p3';

const element = document.getElementById("div1");
const position = document.getElementById("p2");
element.insertBefore(newElement, position);
```



```
This is a paragraph.

This is the new added paragraph

This is another paragraph.
```

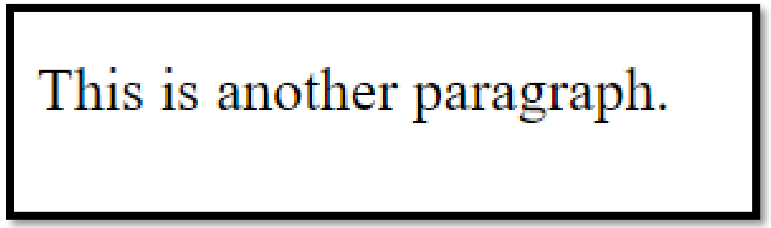
Removing an existing HTML Element

- To **remove** an **HTML element**, use the **remove() method**

```html
<!DOCTYPE html>
<html>

<head>
    <script src="JS/jsCode.js" defer></script>
    <title>Testing JavaScript</title>
</head>

<body>
    <div id="div1">
        <p id="p1">This is a paragraph.</p>
        <p id="p2">This is another paragraph.</p>
    </div>

</body>

</html>
```

jsCode.js

```javascript
const element = document.getElementById("p1");
element.remove();
```

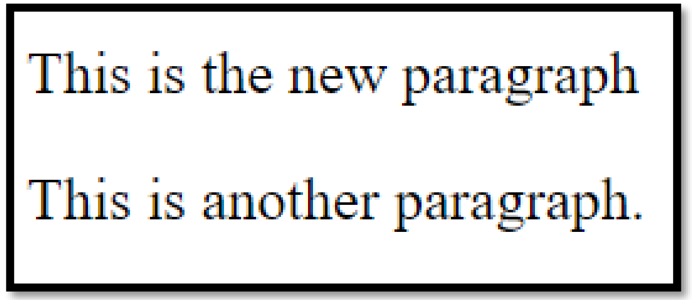

This is another paragraph.

Replacing HTML Elements

- To **replace** an **HTML element** to the HTML DOM, use the **replaceChild()** method:

```html
<!DOCTYPE html>
<html>

<head>
    <script src="JS/jsCode.js" defer></script>
    <title>Testing JavaScript</title>
</head>

<body>
    <div id="div1">
        <p id="p1">This is a paragraph.</p>
        <p id="p2">This is another paragraph.</p>
    </div>

</body>

</html>
```

```javascript
const newNode = document.createElement("p");
newNode.innerHTML = "This is the new paragraph";
newNode.id = 'p3';

const parent = document.getElementById("div1");
const toBeReplaced = document.getElementById("p1");
parent.replaceChild(newNode, toBeReplaced);
```



This is the new paragraph

This is another paragraph.

JavaScript HTML DOM Collections (HTMLCollection Object)

- An **HTMLCollection object** is **NOT an array**! → An HTMLCollection may look like an array, but it is not.
- An **HTMLCollection object** is an **array-like list** (collection) of **HTML elements**.
- You **can loop through the list** and **refer to** the **elements** with an index (just like an array), however, you **CANNOT USE array methods** like toString(), pop(), push(), or join() on an HTMLCollection.

JavaScript HTML DOM Collections (HTMLCollection Object)

- For example, the **.getElementsByTagName()**, method returns an HTMLCollection object.
- The following code selects all <p> elements in a **document** and puts them in **myCollection** list:

```
const myCollection = document.getElementsByTagName("p");
```

- The elements in myCollection can be **accessed** by an **index number with** indexing starting at 0. To **access** the second <p> element you can write:

```
myCollection[1];
```

Looping through the elements in a collection

- The **length property** is **useful** when you want **to loop through** the elements in a collection since it **defines** the **number of elements** in an HTMLCollection.

- In the following example we **change the style** (i.e., text color) of all <p> **elements** in myCollection list to "red".

```
const myCollection = document.getElementsByTagName("p");
for (let i = 0; i < myCollection.length; i++) {
  myCollection[i].style.color = "red";
}
```

Ερωτήσεις?