



UNIT 7: JAVASCRIPT

Markup Languages and Information Management Systems



INDEX

01

Intro-
duction

02

Basic
Programming

03

Basic
Utilities

04

DOM

05

Events

06

Event
Handlers

07

Forms

08

Canvas



01

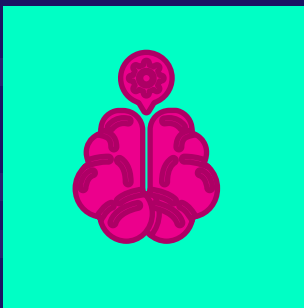
INTRODUCTION

INTRODUCTION

JavaScript is a...

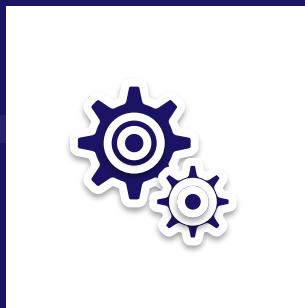
high-level

Instructions **understandable**
by human capacity



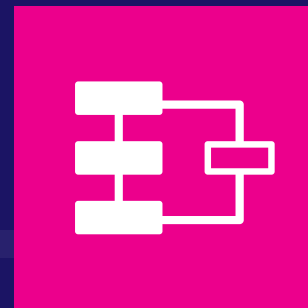
interpreted

Instructions translated to be
executed at **runtime**



weakly-typed

Generic variables
for any data type



...programming language

INTRODUCTION

JavaScript code can be declared...

- **Internally**: as the content of a `<script>` element, either within the HTML `<head>` or `<body>` blocks.
 - Discouraged in `<body>`.
- **Inline**: as the value of some attributes inside an HTML opening tag.
- **Externally**: in separate files.



INTRODUCTION

Inline example

```
<!DOCTYPE html>
<html>
  <head>
    <title>Inline Hello World</title>
  </head>
  <body>
    <h1 onclick="alert('Hello world')">
      My first JS web site
    </h1>
  </body>
</html>
```

INTRODUCTION

Good practice



Internal JS

Put the JS code in the `<head>` block, before any link to CSS files.



External JS

- File with **.js extension**
- Stored in **js folder**
- Linked from the `<head>` block by using `<script>` and its `src` attribute

INTRODUCTION

- HTML is **sequentially** executed.
 - Line by line, from top to bottom.
- To ensure that the JS code is **executed when it should**:
 - Write instructions within **functions**.
 - Associate execution with the realization of certain actions

 **Events**

INTRODUCTION

JS function syntax



```
function functionName(args){  
  //instruction1;  
  //instruction2;  
  //...  
  return returnedValue;  
}
```

The function can be called
from the HTML code

*return is optional and only required
when the function returns a value*

INTRODUCTION


Internal example

```
<!DOCTYPE html>
<html>
  <head>
    <title>Internal Hello World</title>
    <script>
      function helloworld(){
        alert('Hello World');
      }
    </script>
  </head>
  <body>
    <h1 onclick="helloworld()">
      My first JS web site</h1>
  </body>
</html>
```

INTRODUCTION

External example

```
<!DOCTYPE html>
<html>
  <head>
    <title>External Hello World</title>
    <script src="js/hw.js">
    </script>
  </head>
  <body>
    <h1 onclick="helloworld()">
      My first JS web page</h1>
  </body>
</html>
```



```
function helloworld(){
    alert('Hello World');
}
```

*JavaScript code in the
separate hw.js file*

INTRODUCTION

Some basic functions

alert

- Is a method of the `window` object.
 - The `window` word can be omitted.
- Displays a message on screen via pop-up window.
- The message to be displayed is passed as an argument to the function.

```
alert('Message here')
```

INTRODUCTION

Some basic functions

write

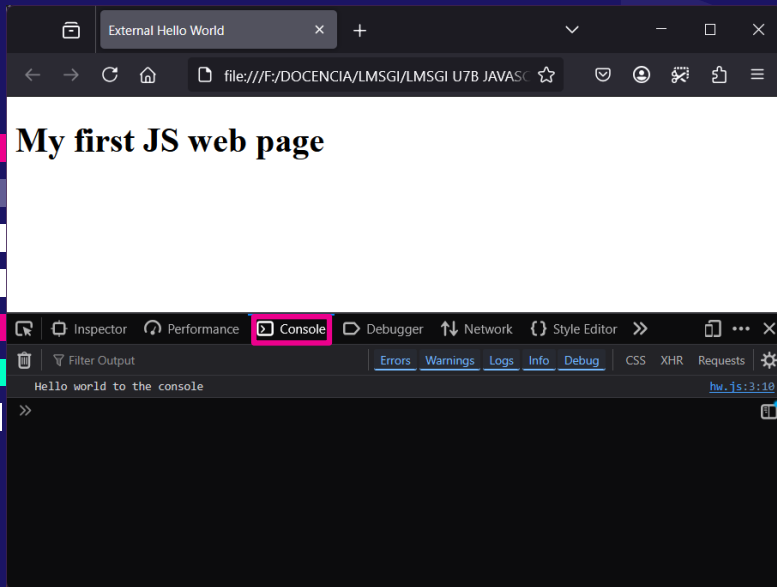
- Is a method of the `document` object
- Writes on the page the text passed as an argument.
- If used after the complete load of the page, it **overwrites** its contents.

```
document.write('Text here')
```

INTRODUCTION

Some basic functions

log



- Is a method of the `console` object
- Outputs a message to the browser console.
 - You need to open the browser's **DevTools** to read.

`console.log('Text here')`

INTRODUCTION

Some basic functions

prompt

- Is a method of the `window` object.
 - The `window` word can be omitted.
- Used to request **data entry** from the **keyboard**.

```
let variable =  
  prompt(message, initialValue)
```

INTRODUCTION

Function example
with prompt + alert



```
function dataRequest(){  
  let userName = prompt(  
    'Enter your name', '');  
  alert('Hello, ' + userName);  
}
```


INTRODUCTION

- JavaScript is an **object-oriented** programming language.
- Objects have members (*methods and properties*).
- Invocation of members of the **window** object can be made by **omitting the object name**.
 - These are equivalent:
 - `window.alert('Hello')`
 - `alert('Hello')`
- The name of **any other object cannot be omitted**.
 - `write('Hello')` causes an error.

INTRODUCTION

Comments

Single line

Start the line with `//`

JS

Multiline

Encase the block inside `/*` and `*/`

INTRODUCTION

<noscript>

- Defines an **alternative content** to be displayed on the browsers that **do not support script execution**, or have it **disabled**.
- Place it in the <body> block.

</>

```
<noscript>
  <p> This page uses JavaScript.
    Please, turn it on to get the full
    experience.
  </p>
</noscript>
```



02

BASIC PROGRAMMING

BASIC PROGRAMMING

BASICS

JavaScript is
very similar to Java

Spacing

Spaces and line breaks are ignored.

Case sensitive

Lowercase and uppercase letters are different characters

Semicolon

Not necessary (but advisable) to end instructions with a ;

BASIC PROGRAMMING

Constants

- Cannot modify their value after initialization.
- Must be initialized when declared.
- Reserved keyword `const`.
- Notice: It is possible to
 - change the value of the `elements` in a constant array.
 - change the value of the `properties` of a constant object.



```
const pi = 3.14159192;  
  
const daysOfWeek = 7;  
  
const errorMsg =  
    "An error occurred";
```

BASIC PROGRAMMING

Variables

- Their values can change.
- Declared only *once*.
- To use an already declared variable, just write its name (*identifier*).
- Two types, depending on its *scope*:
 - global / function: *var*
 - block: *let*

Braces delimit blocks

</>

```
var a = 10;  
let b = 3;  
{  
  let b = 6;  
  alert(a + b); //16  
}  
alert(a + b); //13
```

BASIC PROGRAMMING

Variables

- **Variables** (`var` or `let`) defined **within a function, or a block**, only apply within it.
 - **Local** variables.
- **`var`** variables **outside functions are global**.
 - **Problem**: a global **`var`** variable can be redefined from another scope.
- **BETTER USE `let`**.



```
var price = 30;
var quantity = 4;

if (quantity > 3) {
    var price = 45;
}

alert(price) //45
```


BASIC PROGRAMMING

Variables

Valid characters
in identifiers

Digits

Except for the first character

Letters

Beyond the American alphabet

\$

The dollar symbol

_

Underscore

BASIC PROGRAMMING

Variables

- **Weak** and **dynamic** typing:
 - variables are **generic** (undefined) **until they take value**.
 - they can take value of any type **indefinitely**.
- Data types:
 - primitive: **logical**, **numeric**, **textual**.
 - collections: **array**.
 - **object**.



```
let whatever;  
  
whatever = true;  
whatever = 4;  
whatever = 'Text string';  
whatever = ['Saturday', 'Sunday'];  
whatever = {name:Lisa, age:14};
```

BASIC PROGRAMMING



logical

- Also known as *boolean*.
- Two possible values: **true** and **false**.

numeric

- For **numbers of all kinds**.
- With and without decimals.

Data types

BASIC PROGRAMMING

textual

- For **characters** and **text strings**.
- Values in single or double **quotes**.
- If the string already has quotes of one kind, encase it in quotes of the other one.

```
let str1 = 'I am a text';  
let str2 =  
    'Begin "laser" ignition sequence';
```

Data types



BASIC PROGRAMMING

collection

- **Arrays** and such.
- Arrays contain **comma-separated** values encased in **brackets**.
- The elements in an array may, or may not, be of the same type (!).
- The first element is at position 0.
- Access to elements: number between brackets.

```
let wkend = ['saturday', 'sunday'];  
alert('wkend[1]'); //sunday
```

Data types

BASIC PROGRAMMING

object

- Collection of **property:value** pairs.
- The pairs are contained **in braces** and **separated by commas**.
- Access to members:
 - Property name in brackets.
 - Dot operator.

```
let console =  
  {brand:'Nintendo', model:'Switch'};  
alert(console['brand'] + console.model);
```

Data types

BASIC PROGRAMMING

Escape sequences

CRLF	<code>\n</code>
tab	<code>\t</code>
single quote	<code>\'</code>
double quote	<code>\"</code>
backslash	<code>\\</code>

To insert special or reserved characters in text strings, the **backslash** is used.



EXERCISE

Put all the month names into an array and show them all using `alert`.

BASIC PROGRAMMING

Assignment

- Using `=`
- Stores a value in a variable.
- Compound (*cumulative*) assignment can be used.

These assignments are equivalent to:

```
myNumber = myNumber + 4;  
myNumber = myNumber - 5;
```

```
let myNumber = 3;  
{ myNumber += 4 //myNumber is 7 now  
  myNumber -= 5 //myNumber is 2 now
```

Operators

BASIC PROGRAMMING

Logical

Given two booleans, **b1** and **b2**:

- **AND**: **b1** **&&** **b2** is only true if both are.
- **OR**: **b1** **||** **b2** is true if any of them is.
- **NOT**: **!b1** is the opposite value of **b1**

```
let b1 = true, b2 = false;  
let res1 = b1 && b2; //false  
let res2 = b1 || !b1; //true
```

Operators

BASIC PROGRAMMING

Mathematical

- Addition (+), subtraction (-), multiplication (*), division (/), exponentiation (**), modulo (%).
- Also, increase (++) y decrease (--).

Equivalent to:

```
res1 = res1 + 1;
```

```
let res1 = 2 ** 3; //8  
→ res1++; //9  
let res2 = res1 % 4; //1
```

Operators

BASIC PROGRAMMING

Textual

- **Concatenation**, or text strings connection.
- Using `+`.
- *Adding* a number and a string results in a string.
- Compound (*cumulative*) assignment can be used.

Equivalent to:
`s1 += s2;`

```
let s1 = 'spanish', s2 = ' omelette';  
s1 = s1 + s2;  
let order = 1 + ' ' + s1;
```

Operators

BASIC PROGRAMMING

Relational

- Greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), equals (==), different from (!=).
- Same value and type (===), different value or different type (!==).

```
let n1 = 3, n2 = 6, s3 = '3';  
let r1 = (n1 > n2); //false  
let r2 = (n1 == s3); //true
```

Operators

BASIC PROGRAMMING

Control structures

- When used, the flow of instructions in a program **stops being linear** and branches out, opening up different **alternative paths**.
- The **decision** of following one path or another will depend on the fulfillment of certain **conditions**.



BASIC PROGRAMMING

Control structures

if

- Encases a code that **will only execute if some conditions are met**.
- Conditions are logical sentences.
- If the condition is multiple, then logical operations are expected (AND, OR).



```
if (n1 >= 30 && n1 < 40) {  
    alert ('The tens digit is 3');  
}
```

BASIC PROGRAMMING

Control structures

if-else

- Allows the definition of the paths to follow both **when the conditions are met** (if block) and **when they are not** (else block).



```
if (age < 18) {  
    alert('I cannot sell you tobacco.');}  
else {  
    alert('Your tobacco. Thanks.');}
```


BASIC PROGRAMMING

Control structures

if-else if

- Allows the addition of a **new condition** in case the **if** block condition is not met.



```
if (condition1) {  
    alert('Condition 1 is met.');}  
else if (condition2) {  
    alert('Condition 1 is not met, but  
condition 2 is');}  
else {  
    alert('None of the conditions are met');}
```

BASIC PROGRAMMING

Control structures

switch

- Allows the definition of a **decision with more than two possible paths**.
- Specific instructions for one case are written after a **case** clause and end with **break**
- A **default** block can be used to specify instructions when there was no match in the previous cases.



```
switch (expression){  
  case value1:  
    //instructions for this case  
    break;  
    // possibly more cases here  
  case valueN:  
    //instructions for this case  
    break;  
  default:  
    //instructions in case there was no  
    // match in the previous cases  
}
```

BASIC PROGRAMMING

Control structures

switch

- The expression **is compared** with the values of the cases, in order. When they **match**, the instructions encased in that block are executed.
- If there is no match, the code encased in the **default** block will be executed.
- The **default** block is optional and does not have to be the last one.
- There is no need to use **break** to close the last block.



```
switch (carColor){  
  case "red":  
    riskFactor = 0.8;  
    break;  
  case "white":  
    riskFactor = 0.15;  
    break;  
  default:  
    riskFactor = 0.25;  
}
```

BASIC PROGRAMMING

Control structures

for

- Allows the execution of repetition **loops** for a **given number** of iterations.
- *For all the times the condition is met, repeat the code.*
- A **control variable** is used:
 - It gets an initial value before the first iteration.
 - It is included in the condition that is checked before every iteration.
 - It gets updated after each iteration.



```
for (initialization; condition; update) {  
    //instructions that will be repeated  
}
```

BASIC PROGRAMMING

Control structures

In this example:

- **initialization**: the control variable **i** is created and takes 0 as its initial value.
- **condition**: every time that **i** is less than 5, display the message.
- **update**: the value of **i** increases by 1.



```
let message = "This is a loop";  
for (let i=0; i<5; i++) {  
  alert(message);  
}
```

BASIC PROGRAMMING

Control structures

for-of

- Allows you to easily traverse **arrays** and other array-like collections.
- **Automatic initialization and updating** of the control variable.



```
let days = ['monday', 'tuesday', 'wednesday',  
            'thursday', 'friday', 'saturday', 'sunday'];  
for (let d of days) {  
    alert(d);  
}
```

BASIC PROGRAMMING

Control structures

while

- Allows the execution of repetition **loops** for an **indeterminate number** of iterations.
- *While the condition is met, repeat the code.*
- **The condition is checked before** the execution of the encased instructions.
- The **control variable** included in the condition **should vary** to avoid infinite loops.



```
let result = 0;
let number = 5;
let i = 0;
while (i <= number) {
    result += i;
    i++; //update of the control variable
}
```

BASIC PROGRAMMING

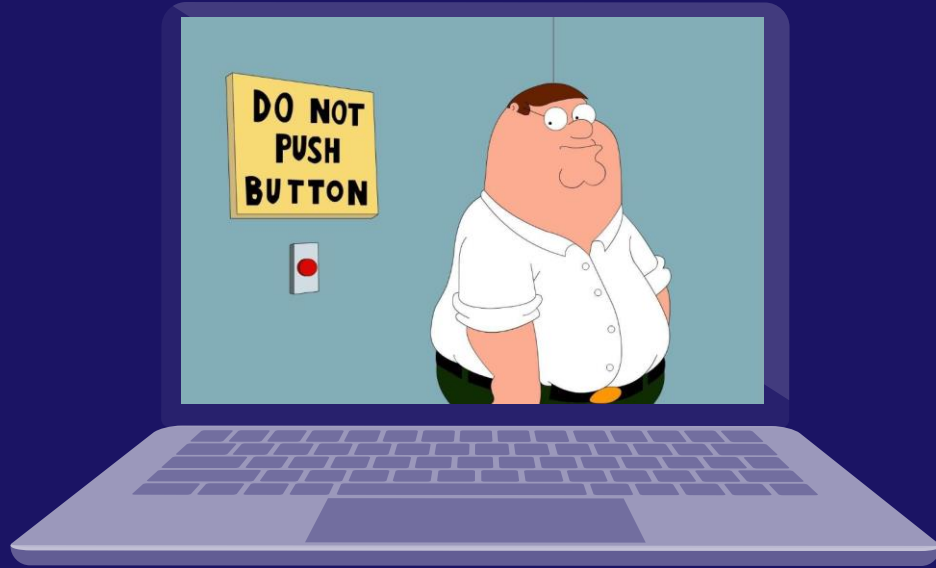
Control structures

do-while

- Same as `while`, but **the condition is checked after** the execution of the encased code.
 - Executed at least once.



```
let result = 0;
let number = 5;
do {
  result += number;
  number--;
} while (number > 0);
alert(result);
```

PRACTICAL EXERCISE 7.1





03

BASIC UTILITIES

BASIC UTIL

- f tools and
properties) for
tion with variables
y with these

BASIC UTILITIES

text

*You can distinguish between a property and a function because a **function** is always followed by parentheses. The function's **arguments** (input parameters) are indicated inside these parentheses, if any are needed*

length

Its value is the number of characters of the string.

concat(*string*)

Returns the concatenation of both strings, like the **+** operator.



```
let lm = "Lenguajes de Marcas";  
alert(lm.length); //19
```

```
let str1 = "Lenguajes ";  
let str2 = str1.concat(" de Marcas");  
alert(str2); //"Lenguajes de Marcas"
```

BASIC UTILITIES

text

toUpperCase()

Returns the string in uppercase.

toLowerCase()

Returns the string in lowercase.



```
let str = "JavaScript";

let str2 = str.toUpperCase();
// 'JAVASCRIPT'

let str3 = str.toLowerCase();
// 'javascript'
```

BASIC UTILITIES

text

`charAt(position)`

Returns the character that occupies the indicated *position*.

`indexOf(character)`

Returns the position of the **first** occurrence of that *character* (or `-1` if not found).

`lastIndexOf(character)`

Returns the position of the **last** occurrence of that *character* (or `-1` if not found).



```
let str = "JavaScript";

let godaime = str.charAt(4);           //'S'

let firstA = str.indexOf('a');         //1

let lastA = str.lastIndexOf('a');      //3
```

BASIC UTILITIES

text

substring(*start*, *end*)

Returns the fragment of the string between the indicated positions.

- *start* included; *end* excluded.
- Inverts positions if *start* > *end*.
- If only one value → *start*.

split(*delim*)

Splits the string, cutting where the *delimiter* is found; returns an array with the fragments.



```
let str = "JavaScript";
let lm = "Lenguajes de Marcas";

let sub1 = str.substring(2,6); // 'vaSc'
let sub2 = str.substring(6,2); // 'vaSc'
let sub3 = str.substring(7);   // 'ipt'

let letters = sub2.split("");
// ['v','a','S','c']
let words = lm.split(" ");
// ["Lenguajes", "de", "marcas"]
```

BASIC UTILITIES

array

length

Number of items of the array.

concat(*array*)

Returns the concatenation of both arrays.

join(*delim*)

Returns a string formed by joining all the items in the array and using the indicated *delimiter* (commas default).



```
let abcde = ['a', 'b', 'c', 'd', 'e'];  
alert(abcde.length); //5
```

```
let arr1 = [1,2,3];  
let arr2 = arr1.concat([4,5]);  
//[1,2,3,4,5]
```

```
let lm=["Lenguajes", "de", "Marcas"];  
let strlm = lm.join(" ");  
//"Lenguajes de Marcas"
```


BASIC UTILITIES

array

pop()

Deletes the **last** item in an array.

Returns the deleted item.

push(*items*)

Attaches those *items* to the **end** of the array.

Returns the new length.



```
let lm =["Lenguajes", "de", "Marcas"];
```

```
let deleted = lm.pop();  
// deleted is "Marcas"  
//lm contains ['Lenguajes', 'de']
```

```
let amount =  
  lm.push('y', 'Sistemas');  
//amount is 4  
//lm contains ['Lenguajes', 'de', 'y',  
//'Sistemas']
```

BASIC UTILITIES

array

shift()

Deletes the **first** item in an array.

Returns the deleted item.

unshift(*items*)

Attaches those *items* to the **beginning** of the array.

Returns the new length.



```
let lm =["Lenguajes", "de", "Marcas"];
```

```
let deleted2 = lm.shift();  
//deleted2 is "Lenguajes"  
//lm contains ['de','Marcas']
```

```
let amount2 = lm.unshift(373);  
//amount2 is 3  
//lm contains [373, 'de', 'Marcas']
```

BASIC UTILITIES

array

`splice(pos, remAmount, items)`

- If two arguments, removes *remAmount* items from the array, starting at *pos*; returns an array with the removed items.
- If *remAmount* is 0, the remaining arguments (*items*) are attached to the array, starting at *pos*; returns an empty array.

`reverse()`

Reverses the order of the items.



```
let lm =["Lenguajes", "de", "Marcas"];

lm.reverse();
//lm contains ['Marcas', 'de',
//'Lenguajes']

let p1 = lm.splice(1,2);
//lm contains ['Marcas']
//p1 contains ['de', 'Lenguajes']

lm.splice(1,0,"la","diferencia");
//lm contains ['Marcas', 'la',
'diferencia']
```

BASIC UTILITIES

number

NaN

Not a Number; the result of an operation impossible to calculate.

isNaN(*param*)

Returns `true` when *param* is NaN.



```
let num = 0 / 0; //NaN

if (isNaN(num1/num2)) {
  alert("The division cannot be done.");
}
```

BASIC UTILITIES

number

Infinity and -Infinity

Positive and negative infinity.

`isFinite(param)`

Returns `true` when *param* is a finite number.

Returns `false` when *param* is (positive or negative) infinity, or NaN.



```
let num1 = 10 / 0;    //Infinity

if (!isFinite(num3)){
    alert('Non-finite number.');
```

BASIC UTILITIES

number

`toFixed(numDigits)`

Returns, as a string, the number rounded to *numDigits* decimal places.



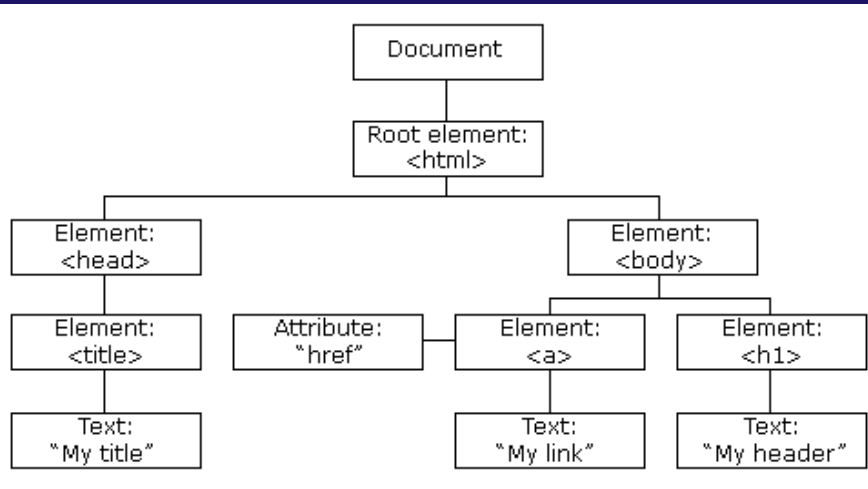
```
let num2 = 3.141591;  
let str = num2.toFixed(2);  
// "3.14"
```



04

DOCUMENT OBJECT
MODEL

HTML DOM



- The DOM is an **API** for manipulating **tree-like** representations of the **HTML** and **XML** documents.
- The **HTML DOM** is created by the browser when the page loads.
- An **object** is created for each element, attribute, text content and comment in the page.

Provides an API

- **Properties** that can be read and written.
- **Methods** that can be used for various actions.



- Modifications in content, structure and style

**Allows programs
to read from and
write to the page**

HTML DOM

The document object

- Is the object that **represents the page**.
- Has properties to easily access its main elements:
 - `document.head`
 - **`document.body`**
- Has methods that allow you to **access any of its elements**:
 - `document.getElementById(id)`
 - `document.getElementsByTagName(tagName)`
 - `document.getElementsByClassName(className)`
 - `document.getElementsByName(name)`
 - `document.querySelector(selector)`
 - `document.querySelectorAll(selector)`

HTML DOM

The document object

`getElementById(id)`

Returns the element with the matching `id` attribute.

`getElementsByTagName(tag)`

Returns, in an ~~array~~ `HTMLCollection`, the elements with the matching `tag` name.

HTMLCollections and NodeLists are array-like collections in that they can be iterated over and their items accessed using position numbers; but do not have array methods like `push()`, `pop()`, `join()`... The `length` property is also available



```
let element =  
  document.getElementById("example");  
  
let array1 =  
  document.getElementsByTagName("input");
```

HTML DOM

The document object

`getElementsByName(name)`

Returns, in an array a `NodeList`, the elements with the matching `name` attribute.

`getElementsByClassName(class)`

Returns, in an array `HTMLCollection`, the elements with the matching `class` attribute.



```
let array2 =  
  document.getElementsByName("tbx1");  
  
let array3 =  
  document.getElementsByClassName(  
    "main");
```

HTML DOM

The document object

`querySelector(selector)`

Returns the **first** element that matches the given **CSS selector**.

`querySelectorAll(selector)`

Returns, in an ~~array~~ a `NodeList`, **all** the matching elements with the given **CSS selector**, in order of appearance.



```
let element =  
  document.querySelector("#example");  
  
let array1 =  
  document.querySelectorAll("input");
```

HTML DOM

Access to element data

innerHTML

Property of an HTML element that holds the **content** encased by its opening and ending tags.

Can be both read and written.

tagName

(Read-only) property of an HTML element that holds its **tag name**, **UPPER-CASE**.

This methods return an array-like object, even if it only contains one item

</>

```
alert(document.getElementsByClassName("sth")[0].tagName.toLowerCase());
```

```
let mainTitle =  
    document.getElementsByTagName("h1")[0];  
alert(mainTitle.innerHTML);  
mainTitle.innerHTML = "My new header";
```

HTML DOM

Access to element data

children

(Read-only) array-like (HTMLCollection)
property of an element that **contains**
its children.



```
let string1 = "";
for (let e of document.body.children) {
  string1 += e.tagName.toLowerCase() + '\n';
}
alert(string1);
```

HTML DOM

Access to element data

`setAttribute(attName, value)`

Adds an attribute to the element, or modifies its **value**, if exists.

`getAttribute(attName)`

Reads the value of an attribute of the element.

Alternatively, you can use the **property with that same attribute name** for both reading and writing attribute values.



```
let a =  
    document.getElementsByTagName("a")[0];  
let newURL = "https://www.iesserpis.org";  
  
alert(a.getAttribute("href"));  
a.setAttribute("href", newURL);  
  
alert(a.href);  
a.id = "1stlink";
```


HTML DOM

Example: change content using innerHTML

```
<body>
  <p>TEXT1</p>
  <p>TEXT2</p>
  <p>TEXT3</p>
  <button type="button"
    onclick="changeContent()">CHANGE
  </button>
</body>
```

```
function changeContent(){
  let p1 =document.querySelector(
    "body p:first-child");
  p1.innerHTML = "TEXT CHANGED";
}
```

HTML DOM

Element creation and insertion

`document.createElement(tagName)`

Creates an element of the given *tag name*.

`document.createTextNode(text)`

Creates a text node with the given *text*.

`appendChild(newElement)`

Method of an element to add *newElement* as the last of its children.

`insertBefore(newElement, existingChild)`

Method of an element to add *newElement* before its *existingChild*.



HTML DOM

Example: inserting elements before and after

```
<html>
<head>
  <script>
    let newLi;
    function createLi(text) {
      let e = document.createElement("li");
      let content = document.createTextNode(text);
      e.appendChild(content);
      return e;
    }
    function liAfter() {
      newLi = createLi("Inserted after the sample");
      document.getElementById("u1").appendChild(newLi);
    }
    function liBefore() {
      newLi = createLi("Inserted before the sample");
      let sample = document.getElementById("li1");
      document.getElementById("u1").insertBefore(newLi, sample);
    }
  </script>
</head>
<body>
  <ul id=u1>
    <li id=li1>Sample item</li>
  </ul>
  <button type=button onclick=liAfter()>After</button>
  <button type=button onclick=liBefore()>Before</button>
</body>
</html>
```

Note: function with
a return value

It can be assigned
to variables

- Sample item

After Before

- Inserted before the sample
- Sample item

After Before

- Sample item
- Inserted after the sample

After Before

HTML DOM

Element replacement and removal

**replaceChild(
 newElement, oldElement)**

Replaces one of the element's children with a new one.

remove()

Method of an element that removes it from the DOM.



```
//replace <body>'s existing <h1> with a new <h2>
let h1 = document.getElementsByTagName("h1")[0];
let h2 = document.createElement("h2");
h2.innerHTML="Replaced";
document.body.replaceChild(h2, h1);
```

```
//remove 2nd item from a <ul>
let b = document.getElementsByTagName("ul")[0]
    .children[1];
b.remove();
```

HTML DOM

CSS inline style modification

style

Property of an element that gives access to its **style** attribute (**inline CSS**).

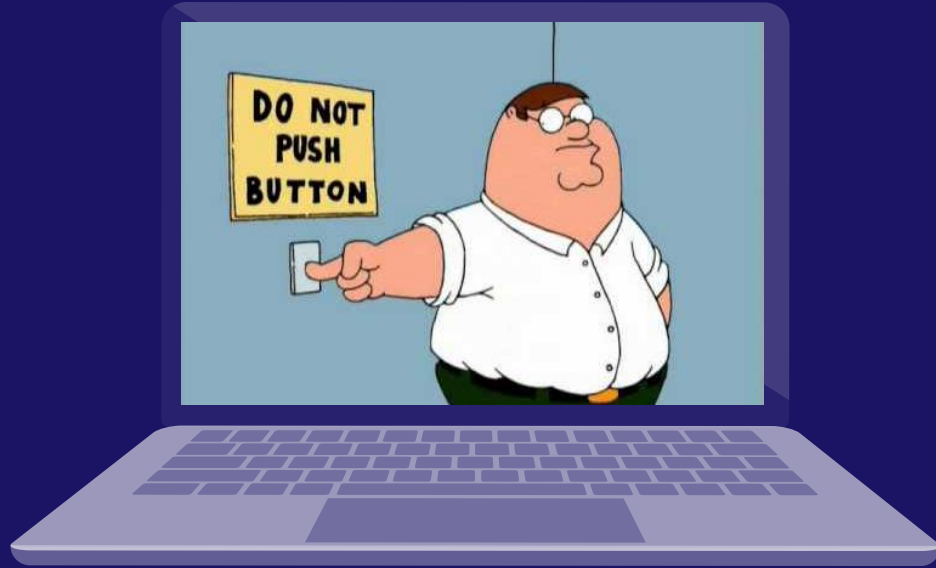
Access to CSS properties is done using the **dot operator**.

CSS property names containing dashes are converted to lowerCamelCase notation.

Example: font-family → fontFamily



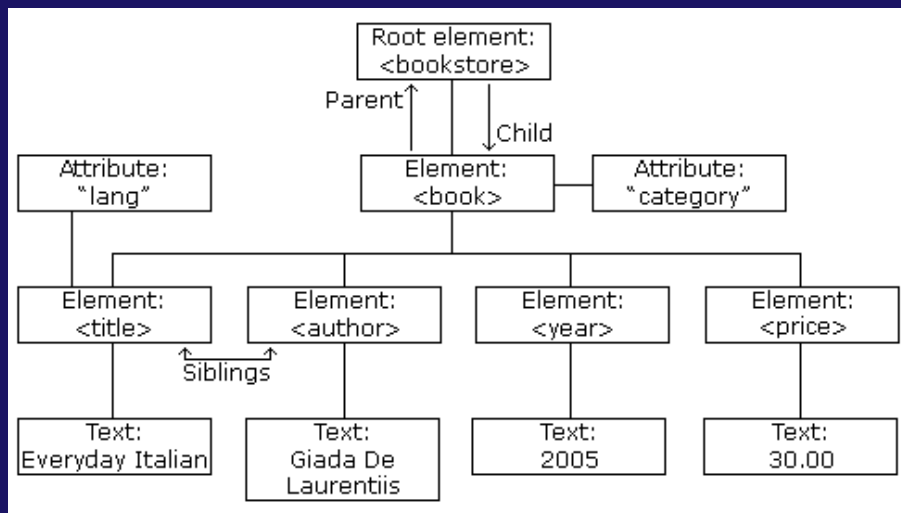
```
document.body.style
  .backgroundColor = "lightgrey";
document.getElementsByTagName("h1")[0]
  .style.fontFamily = "verdana";
```



PRACTICAL EXERCISE 7.2



XML DOM



```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cookbook">
    <title lang="en">Everyday Italian</title>
    <author>Giada de Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
</bookstore>
```

- The **XML DOM** is created by the browser's XML parser.
- An **object** is created for each node.
- In XML, everything is a **node**:
 - The document
 - Each element
 - **An element's text content**
 - Each attribute
 - Each comment
- Provides an API that allows reading from an writing to the XML document.



Basic XML node properties

nodeName

(Read-only) name of the node.

nodeValue

Text content of the node. Can be read and written.

attributes

Object that contains the attributes of the node, and their values.

parentNode

Contains a reference to the parent of the node.

childNodes

(Read-only) NodeList with the child nodes of the node.



Basic XML node methods

`getElementsByTagName(name)`

(XML document method that) Returns a `NodeList` with all the elements with the matching *name*.

~~`appendChild(newNode)`~~

~~Method of a node to add *newNode* as the last of its children.~~

~~`removeChild(childNode)`~~

~~Method of a node to remove an existing *child node*.~~

`getAttribute(attributeName)`

Method of a node that returns the value of its matching attribute.

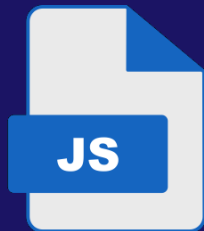
~~`setAttribute(attributeName, value)`~~

~~Method of a node to assign *value* to its matching attribute.~~

XML DOM

*Thus, appendChild(),
removeChild() and
setAttribute() will
not be used here*

- Our goal in this section is to **present** data taken from an XML file in an HTML page.
- Our HTML project needs, at least:
 - ✓ An **XML file** with the data to present.
 - ✓ A page where the data will be presented (**HTML file**).
 - ✓ The JavaScript code, which has to **access the XML DOM, read its content** and **write it to the HTML page**. (a external **JS file** in the example).



XML DOM



XML

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cookbook">
    <title lang="en">Everyday Italian</title>
    <author>Giada de Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
</bookstore>
```

XML DOM

A graphic of a document with a folded top-right corner. A purple rectangular label with the word "HTML" in white is positioned over the bottom-left of the document.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8"/>
  <script src="js/example.js"></script>
</head>
<body>
  <h1>Data taken from the XML file</h1>
  <div>
    <b>category</b> <i>(attribute)</i>: <span
id="cat"></span></br>
    <b>title:</b> <span id="title"></span><br>
    <b>author:</b> <span id="author"></span><br>
    <b>year:</b> <span id="year"></span><br>
    <b>price:</b> <span id="price"></span> €
  </div>
</body>
</html>
```

XHTML DOM

*"The readFromXml function will run on page load"
Events and their handlers are studied later*

JS

1

```
window.addEventListener("load", readFromXml);
```

3

```
function readFromXml() {  
  let xmlhttp = new XMLHttpRequest();  
  xmlhttp.open("GET", "xml/bookstore.xml", false);  
  xmlhttp.send();  
  let xmlDoc = xmlhttp.responseXML;
```

2

4

```
  document.getElementById("cat").innerHTML =  
  xmlDoc.getElementsByTagName("book")[0].getAttribute("category");  
  document.getElementById("title").innerHTML =  
  xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue;  
  document.getElementById("author").innerHTML =  
  xmlDoc.getElementsByTagName("author")[0].childNodes[0].nodeValue;  
  document.getElementById("year").innerHTML =  
  xmlDoc.getElementsByTagName("year")[0].childNodes[0].nodeValue;  
  document.getElementById("price").innerHTML =  
  xmlDoc.getElementsByTagName("price")[0].childNodes[0].nodeValue;  
}
```

XML DOM

XMLHttpRequest object

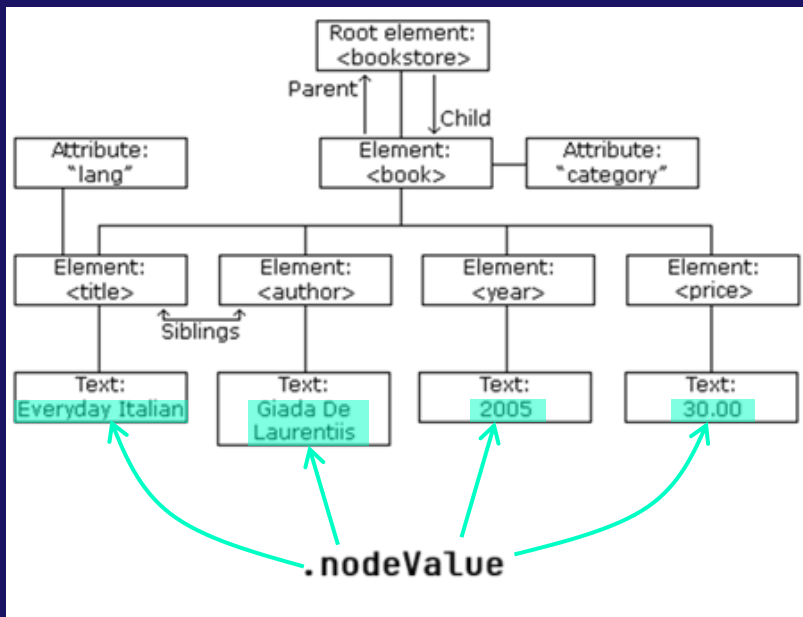
- Is used to **read XML from JS code**.
- How-to:
 - 1 Create a new one (**instantiation**).
 - 2 **Configure file access** (HTTP method, path and response type; **false** means *synchronous*).
 - 3 **Send** it.
 - 4 The response is a **Document** object:
 - Like **document** in the HTML DOM, but the entire XML document now.
 - Assign it to a variable.



The sender of the request stops its activity while waiting for the recipient's response

XHTML DOM

"In XML, everything is a node"



- **childNodes** is an **array-like** property that contains all the direct children of a node.
- If the node is a **leaf** (has no element children), then **childNodes[0]** or **firstChild** is the **node** that represents its (text) content.
- To get the actual text, the **nodeValue** property must be read:

```
xmlDoc.getElementsByTagName("author")[0].childNodes[0].nodeValue;
```

XHTML DOM

The screenshot shows a web browser window with the address bar displaying `file:///F:/DOCENCIA/LMSGI/L`. The page content is titled "Data taken from the XML file" and lists attributes: `category` (attribute), `title`, `author`, `year`, and `price`. A pink box highlights the text "Data from the XML file should be here". Below the browser window, the developer console is open, showing a "Cross-Origin Request Blocked" error. The error message states: "The Same Origin Policy disallows reading the remote resource at file:///F:/DOCENCIA/LMSGI/LMSGI%20U7B%20JAVASCRIPT/Ejercicios%20y%20pr%C3%A1cticas/sandbox/html_xml_dom/xml/bookstore.xml. (Reason: CORS request not http)." A link to "Learn More" is provided. Below this, another error is visible: "Uncaught DOMException: A network error occurred." with the stack trace pointing to `readFromXml` in `example.js:6`.

CORS blocking

- XMLHttpRequests with local files using `file:///` instead of `http://` or `https://` pose a **security risk** (*Cross-Origin Resource Sharing*).
- Modern **browsers block them** by default.

XHTML DOM

Avoid CORS blocking

You have several choices:

- Upload the HTML project to a **remote web server**.
- Set up a **local web server** for the HTML project and **use `http://localhost`** to load pages in the browser.
 - Easy with VS Code or IIS (Windows only).
- **Override the CORS blocking in your browser:**

Easy



Very easy

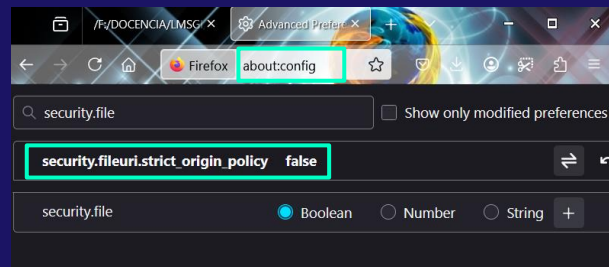


Go to `about:config`, accept risk warning, disable `security.fileuri.strict_origin_policy`

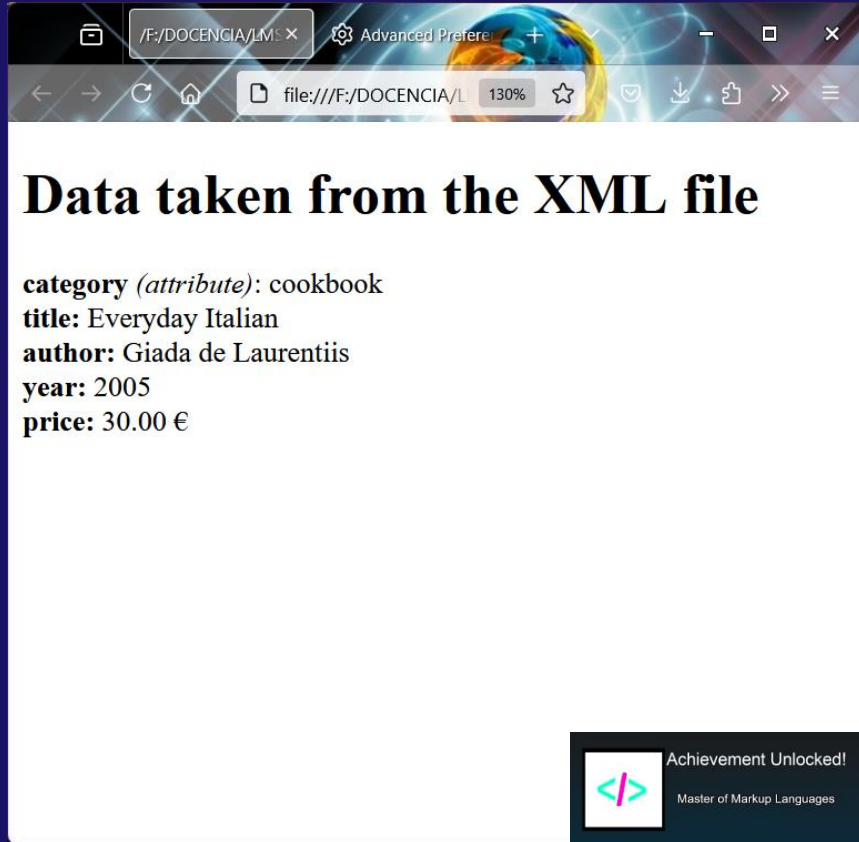
Easy, but slow



Run using command:
`msedge --disable-web-security --user-data-dir="path_to_root_dir"`



*Other
browsers?*



XML DOM



XML DOM

"In XML, everything is a node"

About the child nodes...

- Spaces and CRLF between tags are also in nodes!
- In this example, **<book>** has 9 child nodes:
 - 4 nodes containing **elements**
 - 5 nodes containing **CRLF and tabs**

```
>> xmlDoc.getElementsByTagName("book")[0].childNodes
← ▼ NodeList(9) [ #text, title, #text, author, #text, year, #text, price, #text ]
  ▶ 0: #text "\n\t\t"
  ▶ 1: <title lang="en">
  ▶ 2: #text "\n\t\t"
  ▶ 3: <author>
  ▶ 4: #text "\n\t\t"
  ▶ 5: <year>
  ▶ 6: #text "\n\t\t"
  ▶ 7: <price>
  ▶ 8: #text "\n\t"
  length: 9
```

- Use **getElementsByTagName()** to access to XML nodes whenever possible!

</>

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cookbook">
    <title lang="en">Everyday Italian</title>
    <author>Giada de Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
</bookstore>
```

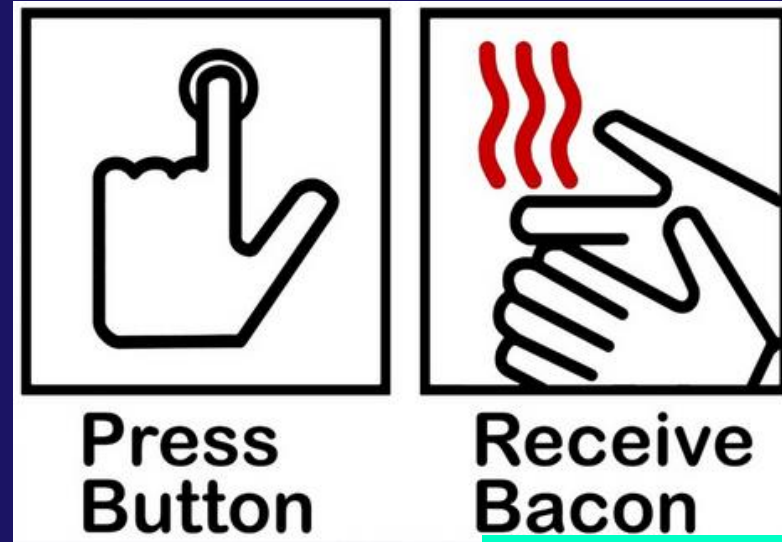


05

EVENTS

EVENTS

- Actions that can be detected by a program.
 - Example: the `click` event is detected when the user clicks on any element on the page.
- When included in HTML pages, JavaScript can react to them.
- The browser can be instructed to do something when certain event is detected.



EVENTS

Functioning

- Events in HTML files are set as **attributes for elements**.
- You can set one or more detectable events in the opening tag of an element.
- The corresponding **event attribute** has the **same name as the event, prefixed with on-**.
- The value these attributes accept is either JS code or a **call to the function** where the JS code is.
- Example:

```
<body onclick="alert('Some message');">
```

EVENTS

of the user interface

- **resize**: the size of the browser window has changed.
 - `<body>`
- **load**: the **resource** has finished loading on the page.
 - `window`; `<body>`, ``, `<input type="image">`, `<link>`, `<script>`, `<style>`
- **error**: the **resource** load failed and could not be completed.
 - ``, `<input type="image">`, `<audio>`, `<video>`, `<link>`, `<script>`

EVENTS

of the user interface

- **scroll**: detected when the user *scrolls* over the element.
 - *Scrollable* elements, blocks typically.
- **select**: the user has **selected** some text in a **text box or area**.
 - `<input>` (*all the text box types*), `<textarea>`

EVENTS

relative to the
focus of an element

- **blur**: the element has **lost focus**.
 - **focus**: the element has **gained focus**.
 - **focusout**: the element has **lost focus** (*bubbling*).
 - **focusin**: the element has **gained focus** (*bubbling*).
 - Can be applied to any element able to gain focus.
- *Event bubbling*: the event is detected either by the element or by any of its children.

EVENTS

generated with
the mouse

- `click`: the element was **clicked** (pressed + released).
 - `dblclick`: the element was **double clicked**.
 - `mousedown`: the element was **pressed, but not released**.
 - `mouseup`: the element was pressed and has just been released.
 - Happens before **click**.
 - `contextmenu`: like `click`, but using the **secondary button**.
- Can be applied to any clickable element.

EVENTS

generated with
the mouse

- `wheel`: the mouse wheel was rotated over an element.
- `mouseenter`: the mouse pointer entered an element.
- `mousemove`: the pointer moved while over an element.
- `mouseleave`: the pointer moved out of an element.
- `mouseover`: the pointer entered an element (*bubbling*).
`mouseout`: the pointer moved out of an element (*bubbling*).

➤ Can be applied to any element with size > 0.

EVENTS

of media elements

- `loadstart`: the element **started loading**.
- `loadeddata`: the **data for the first *frame*** was loaded.
- `canplay`: **enough** of the element was downloaded **to play**.
- `canplaythrough`: like before, but allowing **non-stop playback**.
- `error`: the **load** of the element **failed**.

EVENTS

of media elements

- **play**: the element **started playing**.
- **playing**: the element **resumed playing** after a pause (or after being halted due to lack of data).
- **pause**: the element was **paused**.
- **ended**: the element **ended** playback.
- **volumechange**: the **volume level** of the element **changed**.

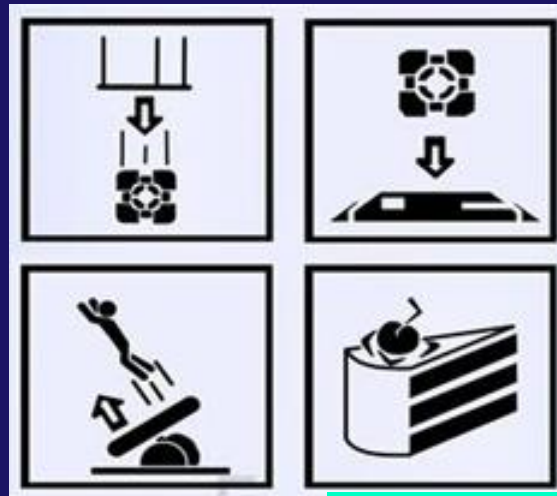


06

EVENT HANDLERS

EVENT HANDLERS

- JavaScript code usually runs after the user performs some actions.
- These user actions and other events will trigger JavaScript functions, which will **handle the event**.
- But first, **event-function assignments must be done**.



EVENT HANDLERS

ASSIGNMENT

Three ways to register an event
for an HTML element

Inline

Add the event attribute to its
HTML tag

HTML DOM

Assign as a value to the
corresponding property
of the HTML object

addEventListener()

As arguments of this method
of the element

EVENT HANDLERS

Inline

- Declaration of event and handler assignment are directly **done in the element's tag**.
- Using *attribute="value"* pairs.
- Deprecated technique, although very practical due to its simplicity.

```
<button onclick="doSomething()">PRESS</button>
```

EVENT HANDLERS

HTML DOM

- Assignment is done using JS code.
- Use the corresponding element's event attribute property to set the value:
 - Actual JS code, or
 - **Name of the triggered function, without parentheses** (if they are used, the function will immediately run and not be registered).
- If the function has input parameters, a *lambda expression* should be used in this form (not studied here).



```
<button id="myButton">  
  PRESS</button>
```

```
<script>  
  document.getElementById("myButton")  
    .onclick = doSomething;  
  
  function doSomething() {  
    //instructions here  
  }  
</script>
```

EVENT HANDLERS

addEventListener()

- Method of an HTML element.
- Is the most powerful way of assigning functions to events.
- Has three arguments:
 1. The **name of the event** (*without on-*), in quotes.
 2. The **name of the triggered function**, **without parentheses**.
 3. (*Optional*) A boolean value indicating, in case of nested elements, whose event will trigger first.
 - **false**: (*default*) the child's event first (*event bubbling*)
 - **true**: the parent's event first (*event capturing*).
- If the function has input parameters, a ***lambda expression*** should be used in this form (not studied here).

EVENT HANDLERS

`addEventListener()`



Register events inside
a function

```
//1) When the page is loaded, register events  
window.addEventListener('load', registerEvents);
```

```
//2) When the first <p> is clicked, show message  
function registerEvents() {  
  let elem = document.querySelector(  
    "body p:first-child");  
  elem.addEventListener('click', showAlert);  
}
```

Call that function
on page load

```
//3) This function shows a message  
function showAlert() {  
  alert('YOU CLICKED!');  
}
```

Declare
event handlers

EVENT HANDLERS

addEventListener()

- `addEventListener()` makes it possible to assign **multiple functions** to handle the **same event** of an element:

```
someElement.addEventListener("click", function1);  
someElement.addEventListener("click", function2);  
someElement.addEventListener("mouseover", function3);
```

EVENT HANDLERS

removeEventListener()

- This method of an element can be used to **remove a previously registered event handler**.

```
someElement.removeEventListener("click", function2);
```



EVENT HANDLERS

In an event handler...

event

Refers to the triggered **event**



this

Refers to the **element** that the **event** is attached to

event.target

Refers to the **element that triggered the event**



PRACTICAL EXERCISE 7.3





07

FORMS

FORMS

- JavaScript has properties and functions that facilitate the programming of applications that handle the appearance of forms, client-side.

Apeology

TO:	
FROM:	DATE:
INFRACTION: <input type="checkbox"/> BEHAVIOR <input type="checkbox"/> ACTION <input type="checkbox"/> WORDS <input type="checkbox"/> INACTION	

REASON(S) FOR MY BEHAVIOR:

<input type="checkbox"/> I was in a foul mood.	<input type="checkbox"/> I wasn't thinking.	<input type="checkbox"/> Someone else made me.
<input type="checkbox"/> It seemed like a good idea.	<input type="checkbox"/> It just happened.	<input type="checkbox"/> I forgot you didn't like that.
<input type="checkbox"/> I ran out of my meds.	<input type="checkbox"/> I was planning a surprise for you.	<input type="checkbox"/> I couldn't help myself.
<input type="checkbox"/> I was feeling insecure.	<input type="checkbox"/> I had no idea it would hurt you.	<input type="checkbox"/> I was tired.
<input type="checkbox"/> You were pushing my buttons.	<input type="checkbox"/> Mercury was in retrograde.	<input type="checkbox"/> I was hungry.
<input type="checkbox"/> I was being selfish.	<input type="checkbox"/> I needed to vent.	<input type="checkbox"/> I was drunk.
<input type="checkbox"/> I forgot.	<input type="checkbox"/> I was traumatized in childhood.	<input type="checkbox"/> I'm in love with you.
<input type="checkbox"/> I didn't know.	<input type="checkbox"/> You were nearby.	<input type="checkbox"/> I'm a schmuck.
<input type="checkbox"/> I hate you.	<input type="checkbox"/>	<input type="checkbox"/>

This note represents my awareness that my ~~behavior~~ ~~actions~~ ~~words~~ some way upset, hurt, or otherwise alienated you. In light of this understanding, I ☐ **WILL** ☐ **WILL NOT** do it again.

KNOCKKNOCKSTUFF.COM • © 2006 WHO'S THERE, INC

FORMS

- There are many ways to access to forms on web pages through the HTML DOM.
- As with any other element, you can make use of the **document** object's **getter methods**; but also:



1. The **document.forms** property:
HTMLCollection (array-like) whose items are references to each form on the page.
 - For example, the first form would be:
document.forms[0]

FORMS

-
2. If the `<form>` has the **name** attribute, you can use its value, instead of the position number, to select it from **document.forms**.
3. If the `<form>` has the **name** attribute, a **property** for **document** is created with its value to point at that `<form>`.

*This also happens for
 and <a> elements*

*Assuming it's the first
form on the page*

```
</>
```

```
<form name="form1">  
  ...  
</form>
```



```
//the following are equivalent  
let form1_1 = document.forms[0];  
let form1_2 = document.forms["form1"];  
let form1_3 = document.form1;
```

FORMS

Access to input data from controls

`<textarea>` and all `<input>` controls:
(except `type=radio`, `type=checkbox` and `type=file`)

- the element's **value** property.



```
<input type="text" id="tbx1">  
<input type="range" id="rng1">  
<textarea id="ta1"></textarea>
```



```
let value1 = document  
  .getElementById("tbx1").value;  
let value2 = document  
  .getElementById("rng1").value;  
let value3 = document  
  .getElementById("ta1").value;
```

FORMS

Important: radios must have a value, because they share a name and their default value is on

Access to input data from controls

Radio buttons:

- The key point is knowing which one is **checked**, and its **value**.
- Done by reading the **checked** boolean property of the radios.

☐ OPTION 1
☐ OPTION 2
☒ OPTION 3



file://

Element with value: opt1; Selected: false
Element with value: opt2; Selected: false
Element with value: opt3; Selected: true

OK



```
<input type="radio" value="opt1"
      name="question" id="ropt1">OPTION 1<br>
<input type="radio" value="opt2"
      name="question" id="ropt2">OPTION 2<br>
<input type="radio" value="opt3"
      name="question" id="ropt3">OPTION 3<br>
```



```
let result="";
let elem =
  document.getElementsByName("question");
for (let item of elem) {
  result +=
    "\nElement with value: " + item.value +
    "; Selected: " + item.checked;
}
alert(result);
```

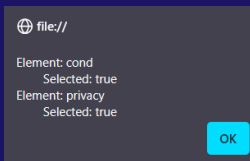
FORMS

Access to input data from controls

Checkboxes:

- Similar to radios, but **name** could be read instead of **value**. (+**checked**).
- It should be noted that they are **not mutually exclusive**.

- ☒ I have read and agree the conditions.
- ☒ I have read the privacy policy.



```
<input type="checkbox" value="cond"
      name="cond" id="cond"/>
I have read and agree the conditions.<br>
<input type="checkbox" value="privacy"
      name="privacy" id="privacy"/>
I have read the privacy policy.
```



```
let result = "";
let elem = document.getElementById("cond");
result += "Element: " + elem.value +
  "\n\tSelected: " + elem.checked;
elem = document.getElementById("privacy");
result += "\nElement: " + elem.value +
  "\n\tSelected: " + elem.checked;
alert(result);
```

Is null for any other
<input> types

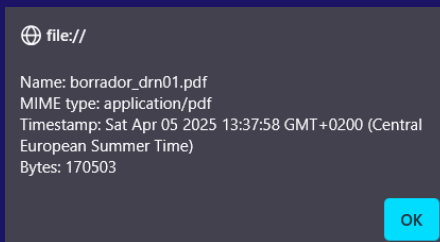
FORMS

Access to input data from controls

File attachment control:

- Its **files** property is an array-like collection of files.
- Items in this collection have file-related properties:
 - **name**
 - **size**
 - **type**
 - **lastModified**

Requires
format conversion



</>

```
<input type="file" name="file1">
```



```
let result="Empty";
let elem = document.
  getElementsByName("file1")[0];
if (elem.files.length>0) {
  let f1 = elem.files[0];
  result = "Name: " + f1.name;
  result += "\nMIME type: " + f1.type;
  result += "\nTimestamp: " +
    Date(f1.lastModified);
  result += "\nBytes: " + f1.size;
}
alert(result);
```



FORMS

Access to input data from controls

Selection controls:

- Key properties of the **<select>** element DOM object:
 - **selectedIndex**: position of the **first** selected option.
 - **value**: value of the **first** selected option.
 - **options**: array-like collection of options.
- Key properties of the **<option>** element DOM object:
 - **value**
 - **innerHTML**
 - **selected**

*If <select multiple>, then you should read/write its **selectedOptions** array-like property; and from its items, their index and value properties*



FORMS

```
<select name=selector>
  <option value=value1>~1st value~</option>
  <option value=value2>~2nd value~</option>
  <option value=value3>~3rd value~</option>
  <option value=value4>~4th value~</option>
</select>
```



```
let elem = document
  .getElementsByName("selector")[0];
let pos = elem.selectedIndex;
let result = "The selected option's value is "
  + elem.value + "\nand its content is "
  + elem.options[pos].innerHTML;
alert(result);
```



The selected option's value is value3
and its content is ~3rd value~

OK

FORM VALIDATION

- HTML5 offers different ways to validate the data entered in a form control:
 - **Default** validation (url, email and tel <input> controls).
 - The **required** attribute.
 - The **pattern** attribute (RegExp).
- Sometimes it will be necessary to apply more complex validation mechanisms.
 - Combination of controls.
 - Checking the results of a calculation.



FORM VALIDATION

The main use of JavaScript in form handling is the validation of data entered by the user

Name
This field is required.

City

State

Email ⓘ
Please enter a valid email address.

If users make any mistakes while filling the form, they can be immediately notified, even before the submit button is clicked.

FORM VALIDATION

Some useful **events**:



submit

Triggered when the **form** is **submitted** (either by clicking the submit button or another way).



input

Triggered when the **content** of a text box or area is **changed**

invalid

Triggered when the **data** in a form control is not **valid** when the form is submitted



keyup

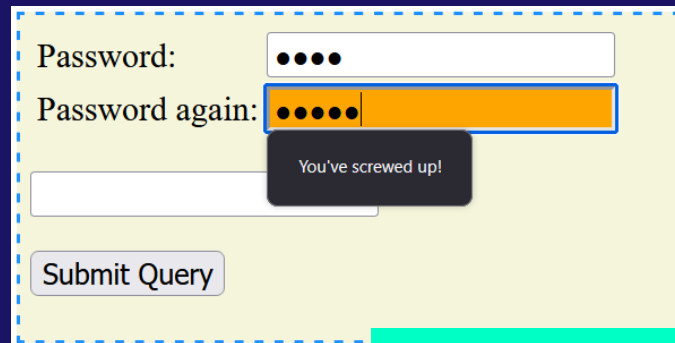
Triggered when any **key press** is **released**



FORM VALIDATION

Customizing the error message

- Browsers display an error message when the user **tries to submit a form** and any of its controls has **invalid data**.
- This message can be customized by using the control's **setCustomValidity()** method.



The image shows a web form with a yellow background and a dashed blue border. It contains two password fields: 'Password:' and 'Password again:'. The 'Password:' field has four black dots. The 'Password again:' field has five black dots and is highlighted with a blue border. A dark grey tooltip with the text 'You've screwed up!' is positioned over the 'Password again:' field. Below the fields is a 'Submit Query' button. A red rectangle is visible at the bottom right of the slide.

FORM VALIDATION

Customizing the error message

- Use the control's `setCustomValidity()` method to set a custom validation error message.
- This way, **the control is marked as invalid**.
 - `validity.customError = true`
- When the submit button is clicked, any controls with a custom validation message (*i.e., invalid*) will block the submission of the form.
- To remove the custom validation message:
`setCustomValidity("")`

FORM VALIDATION

setCustomValidity()

Checking that two password inputs are the same

```
<form>
  <input type="password" name="pw1">
  <input type="password" name="pw2">
  <input type="submit" id="sub1">
</form>
```



```
document.getElementById("sub1")
  .addEventListener('click', validatePw);
```

```
function validatePw() {
  let pass1 = document.getElementsByName("pw1")[0];
  let pass2 = document.getElementsByName("pw2")[0];
  if (pass1.value !== pass2.value) {
    pass2.setCustomValidity(
      "<Custom error message>");
  } else {
    pass2.setCustomValidity("");
  }
}
```

```
▼ pass2.validity: ValidityState
  badInput: false
  customError: true
  patternMismatch: false
  rangeOverflow: false
  rangeUnderflow: false
  stepMismatch: false
  tooLong: false
  tooShort: false
  typeMismatch: false
  valid: false
  valueMissing: false
```

```
▼ pass2.validity: ValidityState
  badInput: false
  customError: false
  patternMismatch: false
  rangeOverflow: false
  rangeUnderflow: false
  stepMismatch: false
  tooLong: false
  tooShort: false
  typeMismatch: false
  valid: true
  valueMissing: false
```


FORM VALIDATION

invalid event

- Is triggered for a **control that has invalid data** at the time the **submit button is pressed**.
- By handling this event, the response can be fully customized.

FORM VALIDATION

invalid event

If any of the
textboxes have
invalid data,
change its
background color
to orange

```
<form id="form2" action="program.php">  
  <input type="email" name="mail"  
    oninvalid="actionsInvalid(this)">  
  <input type="number" min="20" name="numbox"  
    oninvalid="actionsInvalid(this)">  
  <input type="submit" id="sub2">  
</form>
```



```
function actionsInvalid(element){  
  element.style.background = 'orange';  
}
```

FORM VALIDATION

validity.valid
is true



Validation control

- **checkValidity()**: method of a **control** that returns **true** if has valid data.
- **submit()**: method of a **form** that submits the collected data.
- **reset()**: method of a **form** that sets all its controls to their default values.
- Used together, the validity of the controls can be checked before submitting the form.

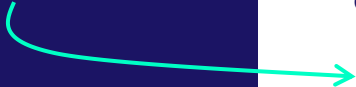
FORM VALIDATION

Validation control

(Using the same `<form>` as in the last example)

```
let form2 = document.getElementById("form2");
form2.addEventListener('submit', mySubmit);

function mySubmit() {
  let tbMail =
    document.getElementsByName("mail")[0];
  let tbNum =
    document.getElementsByName("numbox")[0];
  if (tbMail.checkValidity() &&
      tbNum.checkValidity()) {
    alert('No problems found. Submitting.');
```



```
    document.getElementById("form2").submit();
  }
  else {
    alert('Please double check');
    event.preventDefault();
  }
}
```

Display an alert
if any fields are
invalid

*Prevents the default
behavior of the event
(here: submitting)*

FORM VALIDATION

Real-time validation

- You have learned validation related to the submission of the form so far.
- To check the **validity of the data as the user enters it** into a control, its **validity** property should be read.
 - This property is a `ValidityState` object, with 11 properties.
 - You already learned about **customError**



FORM VALIDATION

PROPERTIES OF A ValidityState OBJECT

*All boolean
and read-only*



All definitions start
as: "This property is
true when..."

badInput

...the entered value does not
conform the expected
internal type. Example: text
in a numeric textbox.

patternMismatch

...the entered value does
not match the specified
pattern.

rangeOverflow

...the entered value is
greater than the value
of the **max** attribute.

rangeUnderflow

...the entered value is
less than the value of
the **min** attribute.

stepMismatch

...the entered value does
not fit the rule set by the
step attribute.

FORM VALIDATION

tooLong

...the entered value is longer than the value of the **maxlength** attribute.

tooShort

...the entered value is shorter than the value of the **minlength** attribute.

typeMismatch

...the entered value does not conform the value of the **type** attribute. Example: URL in an email textbox.

valid

...every other property of **ValidityState** is **false**.

valueMissing


...the control is **required** but has no value.

PROPERTIES OF A ValidityState OBJECT

FORM VALIDATION

Real-time validation

If the textbox is empty or its value is not an email address, change its background color to tomato.



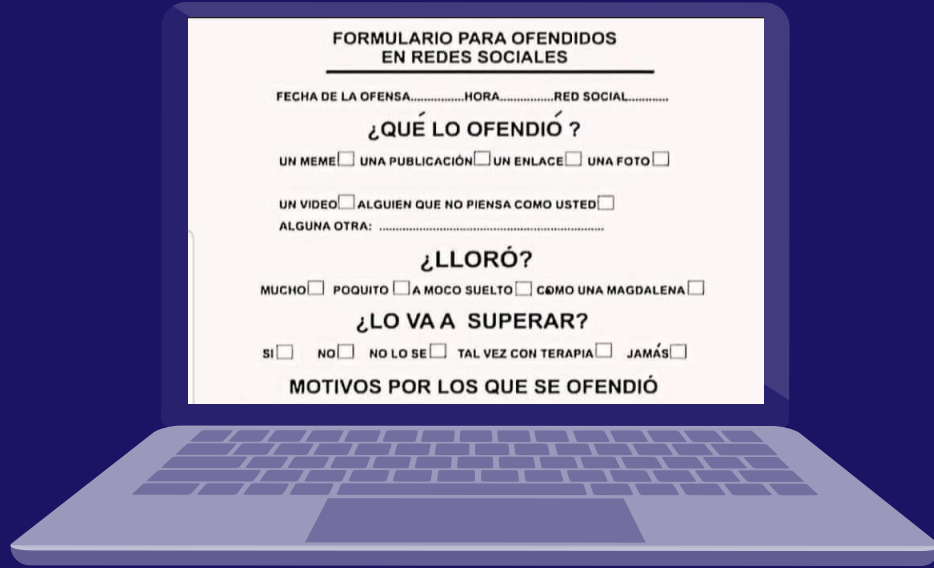
```
<form oninput="validateOnInput(event)">
  <input type="email" required>
  <input type="submit">
</form>
```



```
function validateOnInput(ev){
  let control = ev.target;
  if (control.validity.typeMismatch ||
      control.validity.valueMissing)
    control.style.background = "tomato";
  else
    control.style.background = "transparent";
}
```



PRACTICAL EXERCISE 7.4



**FORMULARIO PARA OFENDIDOS
EN REDES SOCIALES**

FECHA DE LA OFENSA.....HORA.....RED SOCIAL.....

¿QUÉ LO OFENDIÓ ?

UN MEME ☐ UNA PUBLICACIÓN ☐ UN ENLACE ☐ UNA FOTO ☐

UN VIDEO ☐ ALGUIEN QUE NO PIENSA COMO USTED ☐

ALGUNA OTRA:

¿LLORÓ?

MUCHO ☐ POQUITO ☐ A MOCO SUELTO ☐ COMO UNA MAGDALENA ☐

¿LO VA A SUPERAR?

SI ☐ NO ☐ NO LO SE ☐ TAL VEZ CON TERAPIA ☐ JAMÁS ☐

MOTIVOS POR LOS QUE SE OFENDIÓ



08

CANVAS

CANVAS

- `<canvas>` is an inline HTML element that basically serves as a **container for graphics**.
 - You have to draw with JS.
- Its optional **width** and **height** attributes default to 300 and 150, respectively.



CANVAS

`<canvas>`

- Can receive **CSS styles** like any other box.
 - But they only affect the canvas, **not what is drawn on it**.
 - If no style → transparent canvas.
- Initially **empty**.
 - The script first needs to access the **drawing context**, which supplies the drawing functions.
 - **getContext()** method.
 - Argument is the context type: **"2d"**, **"webgl"**...

CANVAS

empty canvas

You need to somehow style the canvas to make it visible (add border, bg color...)

```
<canvas id="canvas1" width="150"
  style="border: 1px solid black;
  background-color: lightgrey;">
</canvas>
```



```
function draw() {
  let canvas1 =
    document.getElementById("canvas1");
  let ctx1 = canvas1.getContext("2d");
}

window.addEventListener("load", draw);
```

We will only draw using **2d** drawing functions.

CANVAS

rectangles

- filled rectangle

`fillRect(x, y, width, height)`

- empty rectangle

`strokeRect(x, y, width, height)`

- erases rectangular area and lets the canvas show

`clearRect(x, y, width, height)`

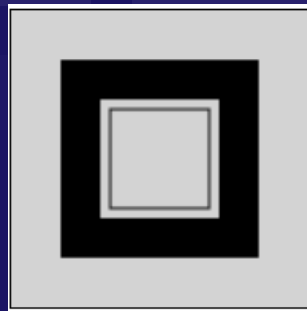
*x, y: coordinates of the starting point
(top left corner of the rectangle);
width is positive to the right;
height is positive down*

CANVAS

rectangles

Using the
previous empty
canvas example

```
function draw() {  
  let canvas1 =  
    document.getElementById("canvas1");  
  if (canvas1.getContext){  
    let ctx1 = canvas1.getContext('2d');  
  
    ctx1.fillRect(25,25,100,100);  
    ctx1.clearRect(45,45,60,60);  
    ctx1.strokeRect(50,50,50,50);  
  }  
}
```

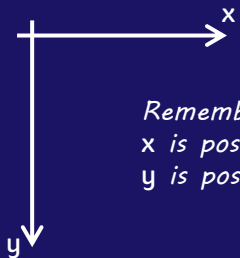


CANVAS

paths

Drawing **any other shape** requires:

1. **creating a path** (**beginPath()**)
2. using instructions to **draw** the path
3. closing the path (**closePath()**)
4. (optionally) setting the **width** of the path (**lineWidth;** default: 1)
5. applying a **contour** (**stroke()**) and/or **fill** (**fill()**)



*Remember:
x is positive to the right;
y is positive downwards*

CANVAS

paths

`beginPath()`

- internally, paths are stored as a **list of sub-paths** (segments, arcs...).
- every time this method is invoked, that list is **emptied** to begin drawing a new path.

`moveTo(x, y)`

- positions** the *pen* on the canvas.
- must be the **first instruction** of a line (not an arc).

CANVAS

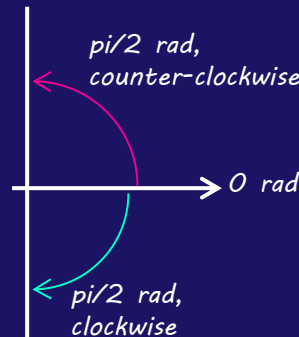
paths

`lineTo(x, y)`

- draws a **line** from the current position of the pen to the position passed as an argument (coordinates).

`arc(x, y, radius, startAngle, endAngle, counterCW)`

- draws an **arc** centered at **x, y**
 - angles, in **radians** (0 rad is equal to *East*)
 - the last argument, optional, is boolean.
 - false** by default (meaning **clockwise** rotation)



CANVAS

paths

`closePath()`

- optional use.
- **tries to close the path** with a straight line to the start point.
 - if already closed or impossible to close, does nothing.
- calling `fill()` automatically closes an open shape.

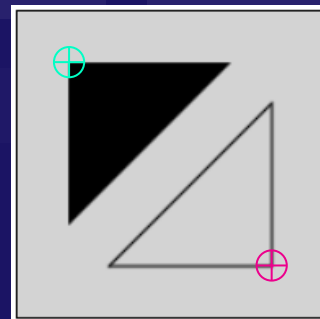
CANVAS

triangles

You can test this code by adding it to the `draw()` function.

Add a new `<canvas>` to the page for this example and assign it to a new variable, `canvas2`.

```
if (canvas2.getContext) {  
  let ctx2 = canvas2.getContext("2d");  
  
  ctx2.beginPath();  
  ctx2.moveTo(25,25);  
  ctx2.lineTo(105,25);  
  ctx2.lineTo(25,105);  
  //ctx2.closePath();  
  ctx2.fill();  
  
  ctx2.beginPath();  
  ctx2.moveTo(125,125);  
  ctx2.lineTo(125,45);  
  ctx2.lineTo(45,125);  
  ctx2.closePath();  
  ctx2.stroke();  
}
```



CANVAS

$$180^\circ = \pi \text{ rad}$$

$$90^\circ = \pi/2 \text{ rad}$$

$$360^\circ = 2\pi \text{ rad}$$

$$x^\circ = x * \pi / 180 \text{ rad}$$

arcs

You can test this code by adding it to the `draw()` function.

Add a new `<canvas>` to the page for this example and assign it to a new variable, `canvas3`.

```
if (canvas3.getContext){
  let ctx3 = canvas3.getContext("2d");

  //round face
  ctx3.beginPath();
  ctx3.arc(75,75,50,0,Math.PI*2,true);
  ctx3.stroke();
  //mouth, clockwise
  ctx3.beginPath();
  ctx3.arc(75,75,35,0,Math.PI,false);
  ctx3.stroke();
  //left eye
  ctx3.beginPath();
  ctx3.arc(60,65,5,0,2*Math.PI,true);
  ctx3.fill();
  //right eye
  ctx3.beginPath();
  ctx3.arc(90,65,5,0,2*Math.PI,true);
  ctx3.fill();
}
```

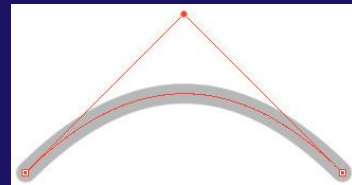


CANVAS

Bézier curves

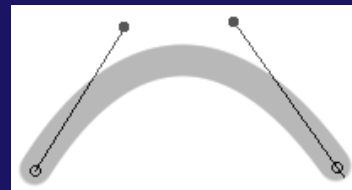
`quadraticCurveTo(cp1x, cp1y, x, y)`

- Draws a **quadratic Bézier curve** from the current pen position to (x, y) using the given **control point** (cp1x, cp1y).



`bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)`

- Draws a **cubic Bézier curve** from the current pen position to (x, y) using the given **control points** (cp1x, cp1y) and (cp2x, cp2y).



CANVAS

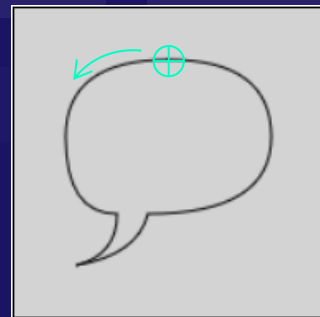
quadratic curves

You can test this code by adding it to the `draw()` function.

Add a new `<canvas>` to the page for this example and assign it to a new variable, `canvas4`.

```
if (canvas4.getContext){
  let ctx4 = canvas4.getContext("2d");

  ctx4.beginPath();
  ctx4.moveTo(75,25); ⊕
  ctx4.quadraticCurveTo(25,25,25,62.5);
  ctx4.quadraticCurveTo(25,100,50,100);
  ctx4.quadraticCurveTo(50,120,30,125);
  ctx4.quadraticCurveTo(60,120,65,100);
  ctx4.quadraticCurveTo(125,100,125,62.5);
  ctx4.quadraticCurveTo(125,25,75,25);
  ctx4.stroke();
}
```



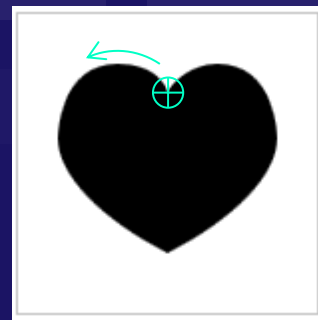
CANVAS

cubic curves

You can test this code by adding it to the `draw()` function.

Add a new `<canvas>` to the page for this example and assign it to a new variable, `canvas5`.

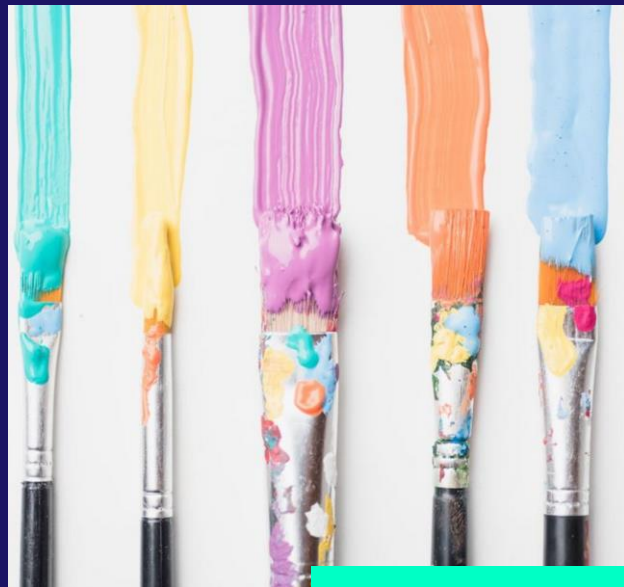
```
if (canvas5.getContext) {  
  let ctx5 = canvas5.getContext('2d');  
  
  ctx5.beginPath();  
  ctx5.moveTo(75,40);  
  ctx5.bezierCurveTo(75,37,70,25,50,25);  
  ctx5.bezierCurveTo(20,25,20,62.5,20,62.5);  
  ctx5.bezierCurveTo(20,80,40,102,75,120);  
  ctx5.bezierCurveTo(110,102,130,80,130,62.5);  
  ctx5.bezierCurveTo(130,62.5,130,25,100,25);  
  ctx5.bezierCurveTo(85,25,75,37,75,40);  
  ctx5.fill();  
}
```



CANVAS

PEN MODIFICATIONS

- So far, our drawings are **black** and have default one pixel wide lines.
- The line width of the *pen* can be modified using the context **lineWidth** property.
- You can color **paths** and **fills** using the following context properties:
 - **strokeStyle**
 - **fillStyle**
- Their values should be set **before actually drawing**.



CANVAS

line width

- **lineWidth** property of the 2D context.
- Positive integer values are accepted (*default is 1*)

```
ctx4.lineWidth = 10;  
ctx4.stroke();
```



CANVAS

colors

- As mentioned above, you can color **paths** and **fills** using these properties:
 - **strokeStyle**
 - **fillStyle**
- Their accepted values are:
 - a color
 - ~~a gradient~~ (*not studied here*)
 - an image pattern



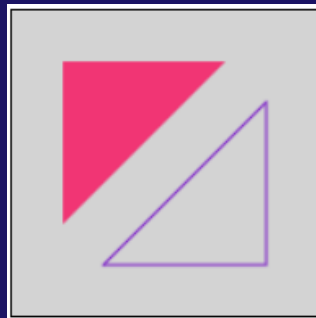
CANVAS

colors

- In quotes.
- Any valid HTML format.
- Default value is "#000000" ("black").

```
ctx2.fillStyle = "#F1E574";  
ctx2.fill();
```

```
ctx2.strokeStyle = "hsl(273, 68%, 47%)";  
ctx2.stroke();
```



CANVAS

image pattern

`createPattern(image, repetition)`

- Arguments:
 1. **reference** to an image.
 2. "repeat", "repeat-x", "repeat-y" or "no-repeat".
- The canvas can only **draw fully loaded images**.
 - After assigning an image file to a variable, you must wait for it to finish loading.
 - Handling the image **load** event with an internal function.

CANVAS

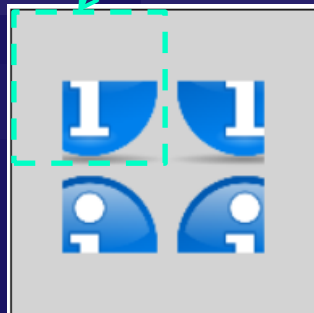
image pattern

You can test this code by adding it to the `draw()` function.

Add a new `<canvas>` to the page for this example and assign it to a new variable, `canvas6`.

```
if (canvas6.getContext) {  
  let ctx6 = canvas6.getContext('2d');  
  let img1 = new Image();  
  img1.src = "img/help.png";  
  img1.addEventListener("load", myFill);  
  
  function myFill() {  
    let patt =  
      ctx6.createPattern(img1, "repeat");  
    ctx6.fillStyle = patt;  
    ctx6.fillRect(25, 35, 100, 85);  
  }  
}
```

The first image sticks to the upper left corner of the canvas



CANVAS

text

- Three context **properties** allow setting the insertion of text into the canvas:
 - **font**: *shorthand* for the `font-family`, `font-size`, `font-style`, `font-weight` and `line-height` CSS properties, **in quotes**.
 - **textAlign**: defines the **horizontal alignment**.
 - **NOTE** → aligned with respect to the **x** coordinate of the text position.
 - valores: `start`, `end`, `left`, `right`, `center`.
 - ~~**textBaseline**: defines the vertical alignment.~~
 - ~~accepted values: `top`, `hanging`, `middle`, `alphabetic`, `ideographic`, `bottom`.~~

CANVAS

text

- Then, to insert text, the following context **functions** are available:
 - **fillText**("text", x, y, *maxWidth*)
 - **strokeText**("text", x, y, *maxWidth*)
 - **x**: horizontal coordinate of the insertion point.
 - **y**: vertical coordinate of the text **baseline**.
 - **maxWidth**: the text shrinks if exceeded (*optional*).

CANVAS

text

You can test this code by adding it to the `draw()` function.

Add a new `<canvas>` to the page for this example and assign it to a new variable, `canvas7`.

```
if (canvas7.getContext){
  let ctx7 = canvas7.getContext("2d");

  ctx7.font = "italic 40px Calibri";
  ctx7.strokeStyle = "mediumvioletred";
  ctx7.strokeText("Markup", 10, 40); ⊕

  let img1 = new Image();
  img1.src = "img/colors.png";
  img1.onload = myFill;

  function myFill() {
    let patt = ctx7.createPattern(img1, "no-repeat");
    ctx7.fillStyle = patt;
    ctx7.font = "bold 40px Consolas";
    ctx7.textAlign = "right";
    ctx7.fillText("Languages!", 250, 110); ⊕
  }
}
```



CANVAS

image

- Use `drawImage()` to insert an image into the canvas:

```
drawImage(image, x, y ,width, height)
```

- **image**: the **already loaded** image.
- **(x, y)**: coordinates of the insertion point (top left corner).
- **width, height**: (*optional*) modified lengths of its sides.
- You need to consider the loading of the image from the file and **handle its load** event.

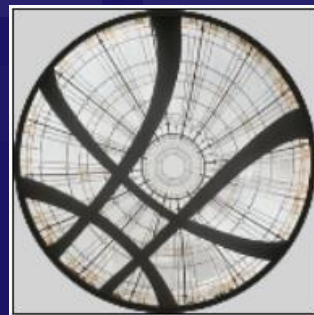
CANVAS

image

You can test this code by adding it to the `draw()` function.

Add a new `<canvas>` to the page for this example and assign it to a new variable, `canvas8`.

```
if (canvas8.getContext) {  
  let ctx8 = canvas8.getContext("2d");  
  let agmt = new Image();  
  agmt.src = "img/agamotto.webp";  
  agmt.onload = imgLoadHandler;  
  
  function imgLoadHandler() {  
    ctx8.drawImage(agmt, 0, 0, 150, 150);  
  }  
}
```





PRACTICAL EXERCISE 7.5



THANKS!

Do you have any questions?



g.domingomartinez@edu.gva.es



CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, and infographics & images by Freepik.