JavaScript

Client Side Web Programming

2024/2025

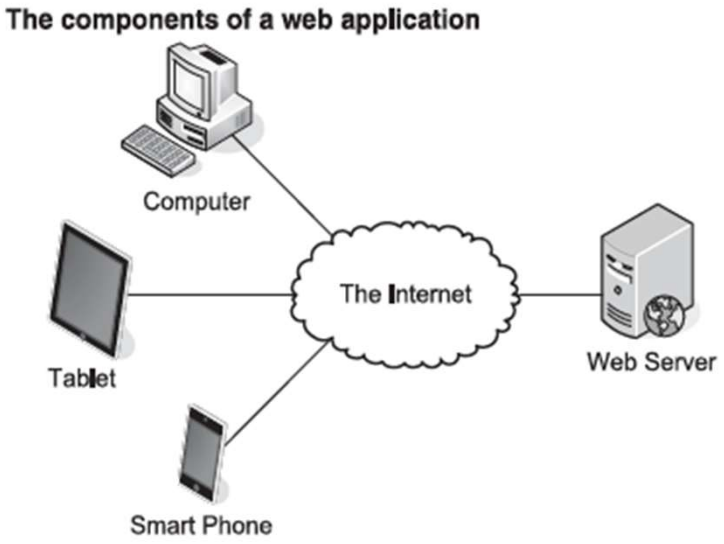
Jose Socuéllamos

How a web application works?

- Today websites follow a model based on client-server programming with three common elements:
  - **Server-side:** includes the HW and SW of the Web server as well as different programming elements and embedded technologies, such as CGI programs, PHP, ASP or Java servlets. They are ran on the server.
  - **Client-side:** it refers to web browsers and is supported by technologies such as HTML, CSS, and languages such as JavaScript and ActiveX controls, which are used to create the presentation of the page or provide interactive features. They are ran on the client, usually in the context of the web browser.
  - **Network:** describes the different elements of connectivity used to display the website to the user.

How a web application works?

- Client user programs (web browsers): they request web pages from the web server.
- Web server: it returns the requested pages to the browser.



How to use the right technology?

- Every technology has its pros and cons.
- We must think of **client-side** and **server-side** technologies as complementary rather than adversarial.
- For example, we want to collect information from a form and record it in a database:
  - It makes more sense to check the form on the client side to ensure that the information entered is correct, just before sending the information to the server database.
  - On the other hand, storing the data on the server would be much better managed by server-side technology.

Static vs Dynamic webpages

- Static websites:
  - They are fixed and display the same content for every user, usually written exclusively in HTML.
  - They doesn't change unless a developer modifies its code.
- Dynamic websites:
  - They can display different content and provide user interaction, by making use of advanced programming and databases in addition to HTML.
  - The final HTML code is created by a script on the browser (client-side) or on the web server (server-side) each time a webpage is requested.
  - The browser doesn't know if the HTML is created dynamically or not, it just displays the final HTML code.

What is JavaScript???

What is JavaScript???

- It's a programming language…
  - High level
  - Interpreted → We don't need to compile
  - Weakly typed
  - Born to be web
  - Based in objects, not object oriented → This is currently changing
  - Event oriented

JavaScript browser compatibility

- As we already said, JavaScript is interpreted by the client.
- There are currently multiple clients or browsers that support JavaScript, including Firefox, Google Chrome, Safari, Opera, Edge, Internet Explorer, etc.
- When we write a script on our web page, we have to be sure that it will be interpreted by different browsers and that it provides the same functionality and features in each of them.
- This is another of the differences with the server scripts in which we will have total control over their interpretation.

Web browser comparison – JavaScript support

Troubleshooting JavaScript

- Sometimes the problem is not our JS code, but the HTML source code.
- That's why it's very important that our HTML code follows the specifications of the W3C standard:

  validator.w3.org

- We have to be careful about the limitations in the use of JavaScript:
  - Not all browsers support JavaScript on the client side.
  - Some mobile devices will also not be able to run JavaScript.
  - JavaScript implementations are not fully supported across different browsers.
  - The user could manually disable JavaScript code execution on the client, so our JavaScript code might never run.
  - Some voice browsers do not interpret JavaScript code.

JavaScript security

- JavaScript provides great potential for malicious designers who want to distribute their scripts over the web.
- To prevent this, web browsers on the client apply two types of restrictions:
  - JavaScript code is executed in a safe execution space or "execution sandbox" in which you can only perform web-related tasks, no generic programming tasks such as file creation, etc.
  - The scripts are restricted by the "same origin" policy: scripts from one website will not have access to information such as usernames, passwords, or cookies sent from another website.

JavaScript limitations

- By security reasons, JavaScript will not be able to perform any of the following tasks:
  - Modify or access the client's browser preferences or appearance.
  - Launch the execution of an application on the client's computer.
  - Read or write files or directories on the client's computer (only cookies).
  - Write files directly to the server.
  - Capture data from a streaming transmission from a server, for retransmission.
  - Interact directly with server languages.
  - Access web pages stored on different domains.
  - Protect the origin of the images on our page.
  - Implement multiprocessing or multitasking.

What's the place of JavaScript in Web programming?

- This scheme shows the 4 layers of web development on the client side.

<table>
<tr><td>Behavior (JavaScript)</td><td rowspan="2"></td></tr>
<tr><td>Presentation (CSS)</td></tr>
<tr><td>Structure (DOM / HTML structure)</td><td rowspan="2">Structured content (HTML document)</td></tr>
<tr><td>Content (text, images, videos, etc)</td></tr>
</table>

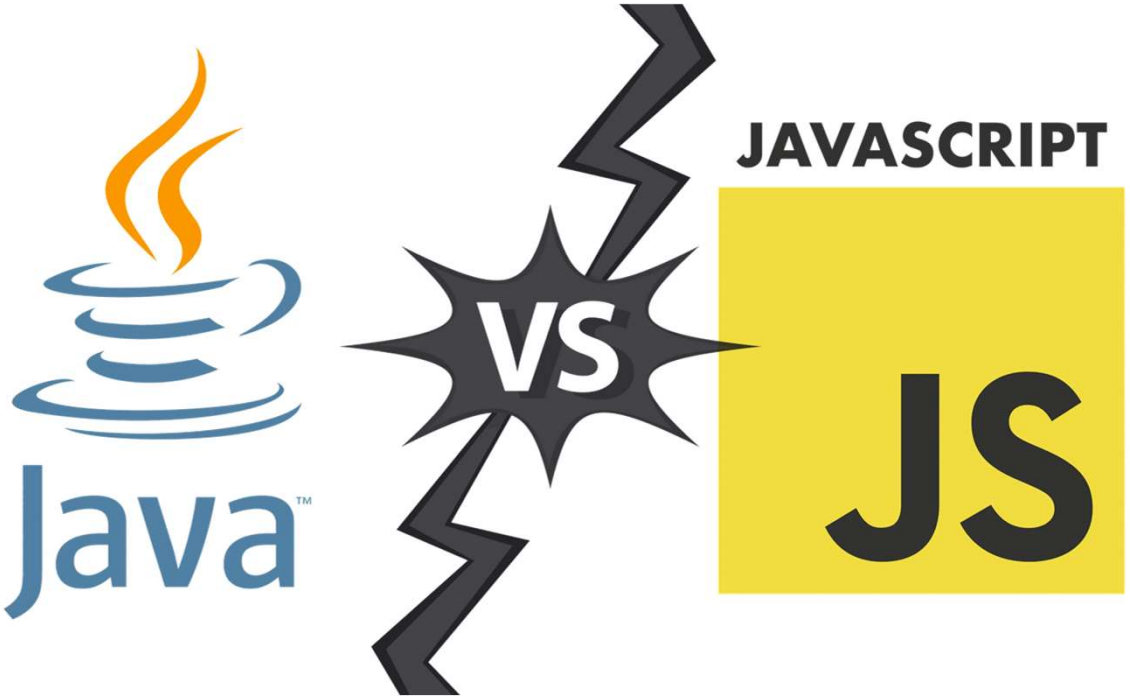
- As we can see, JavaScript is located in the upper layer, managing the behavior of the web page.

What is JavaScript good for?

- JavaScript was born to be run in the browser and it's oriented to provide solutions to:
  - Make our website respond or react directly to user interaction with form elements and hypertext links.
  - Distribute small groups of data and provide a friendly interface for those data.
  - Control multiple windows or browser frames based on the choices the user has made in the HTML document.
  - Pre-process data (validations, calculations…) on the client before sending it to the server.
  - Modify styles and content in browsers dynamically and instantly, in response to user interactions.
  - Request files from the server, and send read and write requests to the server languages.

JavaScript Frameworks

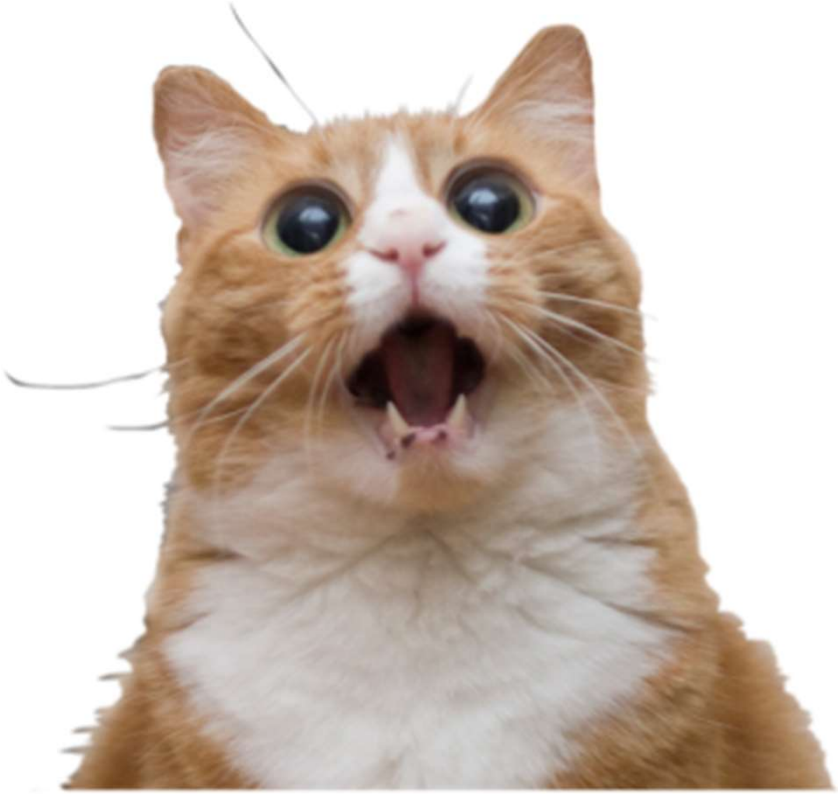
- JavaScript has gradually grown in popularity.
- JS frameworks are collections of JavaScript code libraries that provide developers with pre-written JS code to use for routine programming features and tasks.
- They're used to build websites or web applications around.
- Examples:
  - Responding requests in the server (NodeJS…)
  - Games (Phaser…)
  - Mobile applications (React Native, Ionic…)
  - Databases (MongoDB…)
  - APIs



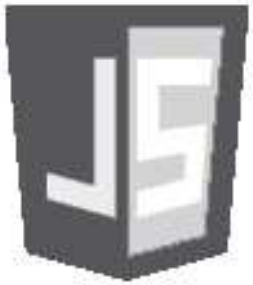
Are Java and JavaScript the same?


I have Java
I have Script
I have JavaScript

JavaScript != Java



JAVA *is to* JAVASCRIPT *as* HAM *is to* HAMSTER

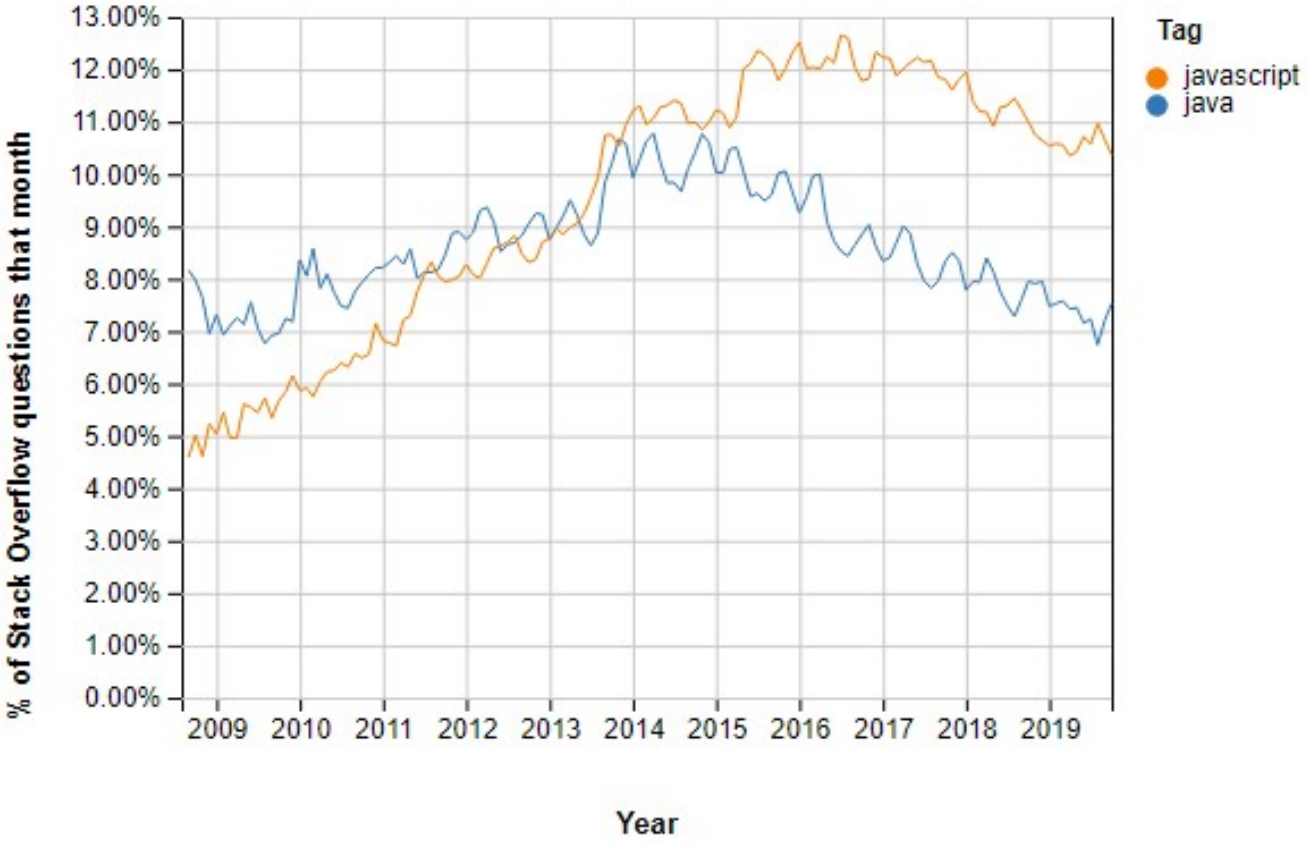


Java vs JavaScript

- Static
- Strongly typed
- Object oriented
- Compiled on the server before execution
- It runs on its own VM
- Strict

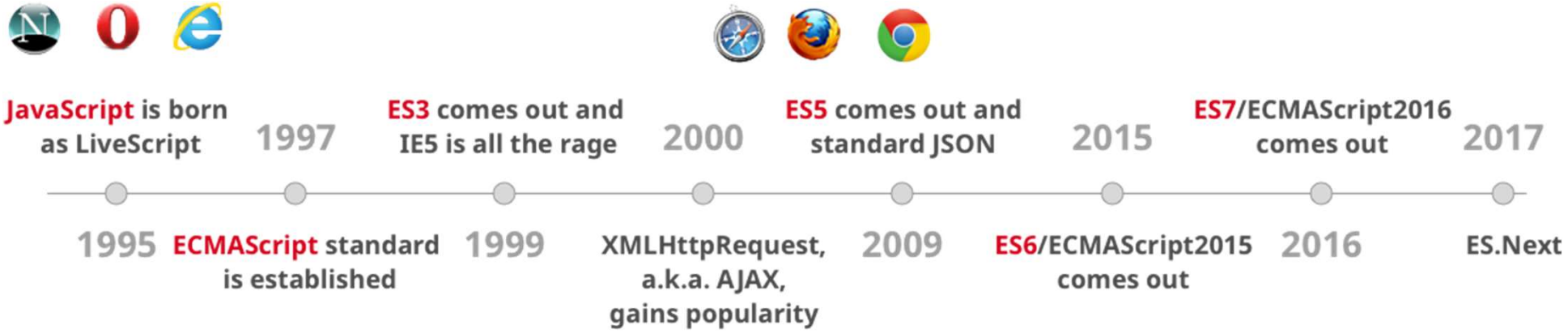
- Dynamic
- Weakly typed (no type)
- Based/built in objects
- Interpreted by the client
- It runs in a browser
- A bit anarchic

Stack Overflow Questions
Java vs JavaScript



Old vs New JavaScript

- JS is constantly growing…



Old vs New JavaScript

- Problem: browsers don't grow fast enough to handle these changes.
- Nowadays there are old and new versions of JS coexisting on the same websites.
- Newer versions of JS are closer to other modern scripting languages (e.g., Python) so they are much more productive.

```
function square(x) {
    return x * x;
}
var myArray = [1, 2, 3, 4, 5];
var newArray = [];
for (var i = 0; i < myArray.length; i++) {
    var elem = myArray[i];
    newArray.push(square(elem));
}
```

```
var myArray = [1, 2, 3, 4, 5];
var newArray = myArray.map( n => n*n );
```

Including scripts in the HTML

- In HTML, JS is included between `<script>` and `</script>` tags.
- It can be included in the *head* or the *body* section

```
<html>
  <head>
    <meta charset="UTF-8">
    <title>Including JavaScript</title>
    <script>
      var a = "Hello";
      console.log(a);
    </script>
  </head>
  <body>
    HELLO WORLD!!!
  </body>
</html>
```

Including scripts in the HTML

- We can include it as an external reference in the *head* section.
  - with a full URL, with a file path or without any path:

```
<html>
  <head>
    <meta charset="UTF-8">
    <title>Including JavaScript</title>
    <script src="https://www.solvam.es/JavaScriptCode.js"></script>
    <script src="js/JavaScriptCode.js"></script>
    <script src="JavaScriptCode.js"></script>
  </head>
  <body>
    HELLO WORLD!!!
  </body>
</html>
```


```
var a = "Hello";
console.log(a);
```

Including scripts in the HTML

- Placing scripts in external files has some advantages:
  - It separates HTML and code
  - It makes HTML and JavaScript easier to read and maintain
  - Cached JavaScript files can speed up page loads
  - We can add several script files to one page by using several script tags.
- That's why it's recommended to place our scripts in external files and link them to the HTML file that uses them.

Including scripts in the HTML

- Later in the course we will need to access HTML elements that are already written in the *body* section of the document from JavaScript.
- That's why it will be necessary not to execute our JavaScript code until the HTML document has completely loaded in the browser.
- We can achieve this in three ways:
  - Writing/linking our JS code at the end of the *body* section
  - Writing/linking our JS code in the *head* section, but wrapping it inside the `window.onload` event (recommended).
  - Using async loading by adding the `defer` attribute in the *script* tag (recommended).

Including scripts in the HTML

Writing/linking our JS code at the end of the *body* section:

```
<body>
  ...
  <script src="myJSCode.js"></script>
</body>
```

Writing/linking our JS code in the *head* section, but wrapping it inside the `window.onload` event.

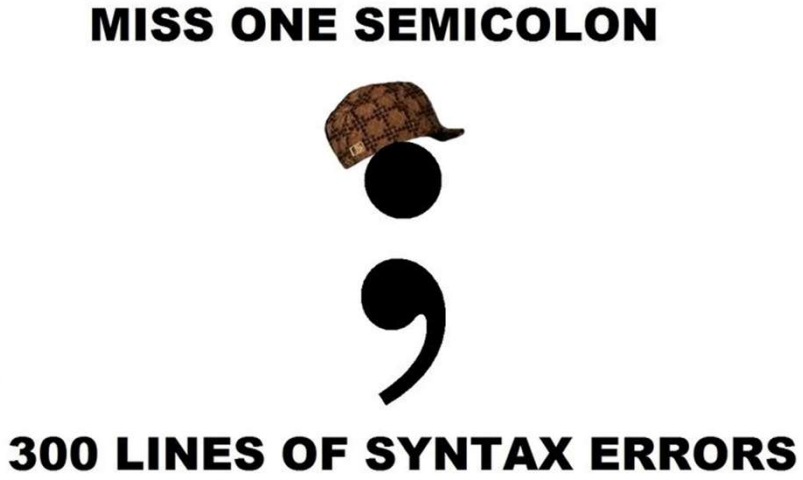
```
<head>
  <script>
    window.onload = function () {
      var a = "Hello";
      console.log(a);
    }
  </script>
</head>
```

Using the `defer` attribute in the *script* tag. The script loads in the background and runs once the DOM is complete.

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

JavaScript syntax

- Comments:
  - `// One line comment`
  - `/* Comments section with more than one line /*`
- Case sensitive: `Simpsons != simpsons`
- Ignores extra whitespaces
- Lines must end with semicolon (;)



Protect your code

- In order for the JavaScript code to be executed correctly, it must be loaded by the web browser and therefore its source code must be visible to the browser.
- That leaves our code in plain view of anyone who wants to read it and steal it to use it fraudulently.
- In case you are wondering, it's impossible to hide it from the visitors, but we can do a couple of things instead:
  - Include a **copyright** message in our source code (it can be deleted).
  - Obfuscate the code (www.javascriptobfuscator.com).
  - Add a **Creative Commons license** and let people to use it, copy it and keep it public (creativecommons.org/licenses/by/4.0/legalcode.es).

Data types and variables

Data types (basics)

Type	Example(s)
Number	`3.4;` `2;`
String	`'Hi, I'm a text string'` `"Hi, I'm a text string"`
Boolean	`true` `false`
null	It means nothing, empty.
undefined	It means not defined yet. Not the same than empty…

Data types (basics)

- In JavaScript there are 5 different data types that can contain values:
  - string (between double quotes " or single quotes ')
  - number (with or without decimals)
  - Boolean (*true* or *false*)
  - object (we'll learn it later)
  - function (we'll learn it later)
- You can use the **typeof** operator to find out the data type of a JavaScript variable.

```
typeof "John"   // Returns "string"
typeof 3.14     // Returns "number"
typeof NaN      // Returns "number"
typeof false    // Returns "boolean"
typeof myCar    // Returns "undefined"
typeof null     // Returns "object"
```

Variables

- Global scope:

```
var x = "Hello";
```

- Local scope:

```
const x = "Hello";
```

```
let x = "Hello";
```

Using `let`

```
let i = 43;
for (let i = 0; i < 10; i++) {
    console.log(i);
}
console.log(i);
```

If we use let in a global scope, it will work as a **var** variable.

However, when we use **let** to declare a variable inside a block, this variable and its value only exist inside the block.

Same with **const**.

`let` vs `const`

- Both are new ways of declaring variables in JS.
- Specially interesting in local blocks because they don't overwrite other global values or other visible variables.
- When do I use **let** or **const**?
  - If the value can change → let
  - If the value will never change→ const
    - If we try to change the value of a **const** variable we'll get an error (TypeError)
    - Useful to debug

Data types
Useful methods and properties

- Strings:

```
let str = "Hello World";
console.log(str.length);           // String length = 11
console.log(str.toUpperCase());    // String to upper case = HELLO WORLD
console.log(str.toLowerCase());    // String to lower case = hello world
console.log(str.substring(0, 5)); // Substring from 0 to 5 = Hello
console.log(str.split(" "));       // Splits the string using the given
                                      delimiter = Array ["Hello", "World"]
console.log(str.charAt(0));        // Character at the given index = H
console.log(str.includes("Wor")); // True if str contains Wor = true

str ="15.37";
let intNum = parseInt(str);        // Converts string into integer = 15
let floatNum = parseFloat(str);    // Converts string into float = 15.37
```

Data types
Useful methods and properties

- Math object - performs mathematical tasks on numbers:

```
let num ="15.37";
console.log(Math.round(num));        //Rounds to the nearest integer
console.log(Math.floor(num));        //Rounds down to the nearest integer
console.log(Math.ceil(num));         //Rounds up to the nearest integer
console.log(Math.trunc(num));        //Returns the integer part of num
console.log(Math.random());          //Returns a random number between 0 and 1
console.log(Math.pow(5,2));          //Returns 5 to the power of 2
console.log(Math.min(1,2,3,4,5));    //Returns the lowest value
console.log(Math.max(1,2,3,4,5));    //Returns the highest value
```

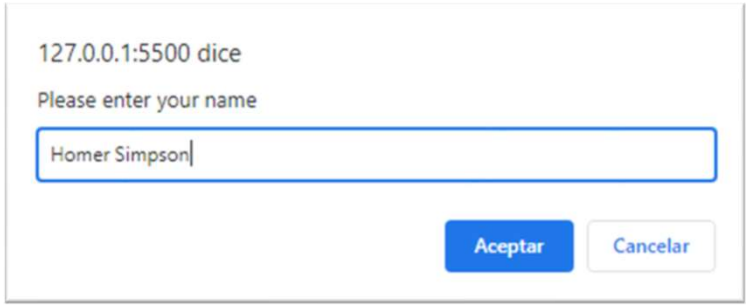
Asking and showing Information (Basic)

prompt() - Input

- The `prompt()` method displays a dialog box that prompts the user for input.

```
let name = prompt("Please enter your name", "Homer Simpson");
```

- When the prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed.
  - If the user clicks "OK", it returns the input value as a `string`, otherwise it returns null.



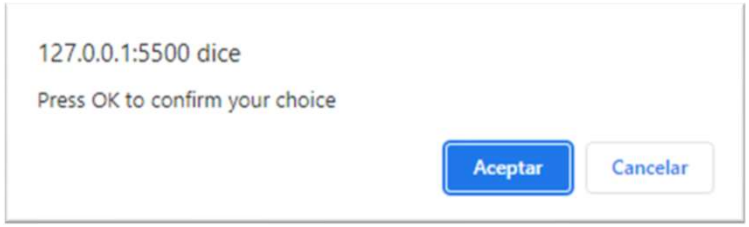
- We can pass an optional default text as parameter.
- We will use it for now, but it's not recommended.

confirm() - Input

- The `confirm()` method displays a dialog box with a message, an OK button, and a Cancel button.

```
let val = confirm("Press OK to confirm your choice");
```

- It returns `true` if the user clicked "OK", otherwise it returns `false`.
- The user is forced to read the message, so it's often used to make the user verify or accept something.
- We will use it for now, but it's not recommended.



alert() - Output

- The `alert()` method displays an alert box with a message and an OK button.

```
alert("Welcome to my website!");
```

- It's used when you want the user to read some information.
- The user is forced to read the message, and the alert box prevents the user from accessing other parts of the page until it's closed.
- We will use it for now, but it's not recommended.

Operators

Operators

- JavaScript is a language rich in operators: symbols and words that perform operations on one or more values, to obtain a new value.
- Any value on which an action (indicated by the operator) is performed is called an operand.
- An expression can contain:
  - an operand and an operator (unary operator), as in b++,
  - two operands, separated by an operator (binary operator), as in a + b.

Comparison Operators

- They compare the values of 2 operands, returning a true or false result (they are used extensively in conditional statements like if… else and in loop statements).

Operator	Name	Example
== (===)	equal to	4 == 5 (false)
!= (!==)	not equal	5 != 7 (true)
>	greater than	12 > 4 (true)
<	less than	6 < 3 (false)
>=	greater than or equal to	3 >= 3 (true)
<=	less than or equal to	22 <= 9 (true)

Use of == and === (!= and !==)

- JavaScript has two equality comparators:
  - Weak equality ==
    - If data types are the same (`4 == 3`), then the values are compared and returns true only if they are identical.
    - If data types are not the same (`4 == true`) it will try to convert the more complex data into the simpler. After that, if both values are equal, it will return true (`4 == "4"` → `4 == 4` → `true`).
  - String equality ===
    - Two objects are only equal if they have the same type and value. It doesn't try to convert them (`4 === "4"` → `false`).
    - If you're not sure of the behavior you need, this is your best option.

Arithmetic Operators

- They are used to perform arithmetic on numbers:

Operator	Name	Example

+	Addition *(Beware! When one of the operands is a string, it will return a string)*	`4 + 7 (11)`
		`4 + "7" (47)`
		`"Hello" + " World" ("Hello World")`

-	Subtraction	`9 - 3 (6)`
*	Multiplication	`3 * 8 (24)`
**	Exponentiation	`5 ** 2 (25)`
/	Division	`12 / 4 (3)`
%	Modulus (Division Remainder)	`14 % 5 (4)`
++ --	Increment - Decrement	`a++ (a = a + 1)` `a-- (a = a - 1)`

Assignment Operators

- They are used to assign values to JavaScript variables:

Operator	Syntax	Same as	Example
=	x = y	x = y	x = 6 y = "Hello" z = true
+=	x += y	x = x + y	x += 1 (x→7) y += 1 (y→"Hello1")
-=	x -= y	x = x - y	x -= 2 (x→5)
*=	x *= y	x = x * y	x *= 4 (x→20)
/=	x /= y	x = x / y	x /= 2 (x→10)
%=	x %= y	x = x % y	x %= 3 (x→1)

Boolean Operators

- They're used to determine the logic between variables or values.
- Given the following initial values of x and y:

```
x = 6;
y = 3;
```

Operator	Description	Example
&&	and	(x < 10 && y > 1) → true
||	or	(x == 5 || y == 5) → false
!	not	!(x == y) → true

The results are given by the AND, OR and NOT logical truth tables

Working with strings

Working with strings

- We can use single or double quotes:

```
let vehicle1 = "bycicle";
let vehicle2 = 'bycicle';
```

- We can use the backslash (\) as a escape character that turns special characters (" ' \)into string characters:

```
console.log("My favorite movie is \"Matrix\"");
console.log('It\'s a great movie!');
```

- Another useful escape sequences in JavaScript:
  - \n – New line
  - \t – Tabulator

```
console.log("Line1\nLine2\tTabulator");
```

```
Line1
Line2   Tabulator
```

Template literals

- We can use back-tics (` `) rather than quotes (" ") to define a string. This way we can use…

Quotes inside strings:

```
console.log(`My favorite movie is "Matrix"`);
```

Multiline strings:

```
console.log(`This is the first line
And this is the second line`);
```

Variables in strings:

```
let name = "John Doe"; let age = 28;
console.log(`My name is ${name} and I'm ${age}`);
```

Expressions in strings:

```
let price = 10; let discount = 0.25;
let total = `Total: ${(price * (1 - discount))}`;
```

HTML Templates:

```
let url = "https://www.solvam.es/";
let text = "Visit Solvam";
let html = `<a href=${url}>${text}</a>`;
```

Data collections

Data types (complex)

Type	Example(s)

Array	`let arrayA = [1, 2, 3, 4, 5];`
	`let arrayB = new Array();`
	`let arrayC = [];`

TypedArray	`let x = new Int32Arrays(2);`
	`x[0] = 32;`
	`x[1] = -1;`

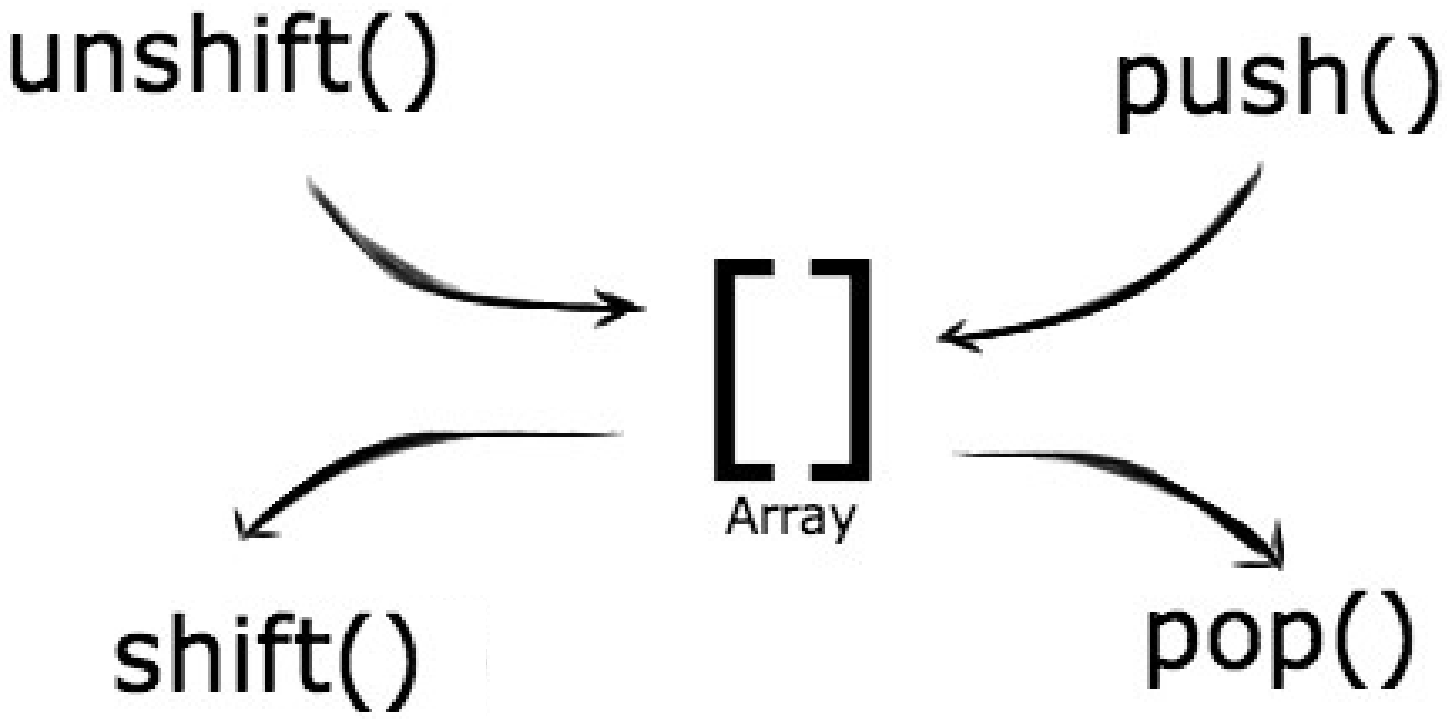
Set	`let mySet = new Set();` `x.add(4);`
Map	`let x = new Map();` `x.set('foo', 16);`
Object	`let obj = {name: "John", age: 30};`

Array methods & properties

Property	Explanation
`length`	Sets or returns the number of elements in an array

Method	Explanation
`indexOf()`	Search the array for an element and returns its position
`shift()`	Removes the first element of an array, and returns that element
`pop()`	Removes the last element of an array, and returns that element
`unshift()`	Adds new elements to the beginning of an array, and returns the new length
`push()`	Adds new elements to the end of an array, and returns the new length
`splice()`	Adds/Removes elements from an array
`slice()`	Selects a part of an array, and returns the new array
`sort()`	Sorts an array alphabetically
`reverse()`	Reverses the elements in an array

Array methods & properties



Conditionals

Conditional statements (if … else if … else)

```
if (condition1) {
    // code block (c1=true)
} else if (condition2) {
    // code block (c1=false)
    //            (c2=true)
} else {
    // code block (c1=false)
    //            (c2=false)
}
```

- condition1 and condition2 are expressions that evaluate to a Boolean value, either true or false.
- The **if** statement specifies a block of JavaScript code to be executed if a condition is true.
- The **else** statement specifies a block of code to be executed if the condition is false.
- The **else if** statement specifies a new condition if the first condition is false.

Conditional statements (if … else if … else)

- Example:

```
let number = -6;
let state;

if (number < 0) {
    state = "negative";}
else if (number > 0) {
    state = "positive";}
else {
    state = "zero";}

console.log(state);
```

```
let age = 25;

if (age >= 18 && age < 30) {
    console.log("10% discount");}
else if (age < 18 || age >= 65) {
    console.log("25% discount");}
else {
    console.log("No discount");}
```

Ternary operator

- The JS ternary operator (? :) makes the code more concise:

```
if (condition) {
    codeblockIfTrue
} else {
    codeblockIfFalse
}
```

```
condition ? codeblockIfTrue : codeblockIfFalse
```

- The condition is an expression that evaluates to a Boolean value, either true or false.
- If the condition is true, the first section (`codeblockIfTrue`) executes. If it is false, the second section (`codeblockIfFalse`) executes.

Ternary operator

- Example:

if-else conditional

```
let age = 18;
let message;
```

```
if (age >= 16) {
    message = 'You can drive.';
} else {
    message = 'You cannot drive.';
}
```

```
console.log(message);
```

1

```
age >= 16 ? (message = 'You can drive.') : (message = 'You cannot drive.');
```

2

```
message = age >= 16 ? 'You can drive.' : 'You cannot drive.';
```

Conditional statements (switch-case)

```
switch(expression) {
    case x:
        // code block
        break;
    case y:
        // code block
        break;
    default:
        // code block
}
```

- The **switch** expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated code block is executed.
- If there is no match, the **default** code block is executed.
- When JS reaches a **break** keyword, it breaks out of the switch block stopping its execution.

Conditional statements (switch-case)

- Example:

```
switch (age) {
    case 0:
        alert("You're a newborn");
        break;
    case 18:
        alert("You can vote");
        break;
    case 65:
        alert("You're retired");
        break;
    default:
        alert("Your age is not important");
        break;
}
```

```
switch (animal) {
    default:
        alert('This is not an animal');
        break;
    case 'dog':
    case 'cat':
    case 'hamster':
        alert('This is a pet animal');
        break;
    case 'lion':
    case 'giraffe':
    case 'crocodile':
        alert('This is a wild animal');
        break;
}
```

JavaScript loops

JavaScript for loop (ES5)

```
          S1          S2      S3
for (let i = 0; i < 10; i++) {
    code block
}
```

- **Statement 1:**
  - sets a variable before the loop starts (let i = 0).
- **Statement 2:**
  - defines the condition for the loop to run (i must be less than 10).
- **Statement 3:**
  - increases a value (i++) each time the code block in the loop has been executed.

- It's used to execute a block of code a number of times.
- Loops are useful to run the same code over and over again, each time with a different value.

JavaScript `while` loop

```
while (condition) {
    code block
}
```

```
var a = 1;
while (a < 10) {
    alert("a is: " + a);
    a++;
};
```

- Used to execute a block of code as long as a specified condition is true.
- We don't know hoy many times we need to run the code block, but we do know the exit condition.
- The variable in the condition must be increased or the loop will never end, and the browser will crash.

JavaScript do...while loop

```
do {
    code block
} while (condition);
```

```
var a = 1;
do {
   alert("a is: " + a);
   a++;
} while (a < 10);
```

- It's a variant of the `while` loop.
- It will execute the code block once, before checking if the condition is true.
- Then it will repeat the loop as long as the condition is true.
- The loop will always be executed at least once, even if the condition is false.

JavaScript for...of loop (ES6)

```
for (variable of iterable) {
    code block
}
```

```
let myArray = [1,2,3,4,5];

for (let value of myArray) {
    console.log(value);
}
```

- New type of loop introduced in ES6.
- Requires a collection of objects or an iterable object (Arrays, Strings, Maps…).
- Requires a local variable (**value** in this case) where each iteration gets the value of one element from the collection.

JavaScript for...of loop (ES6)

```
let myArray = ["ASIR", "DAW", "DAM"];

for (let [i, v] of myArray.entries()) {
    console.log(i + ": index")
    console.log(v + ": value");
}
```

- If we need the index of the iteration, we can destructure the array with the `entries()` method.
- This method creates an Array Iterator object with **key/value** pairs.
- If we use it in a `for-of` loop it can be very useful.

JavaScript Functions

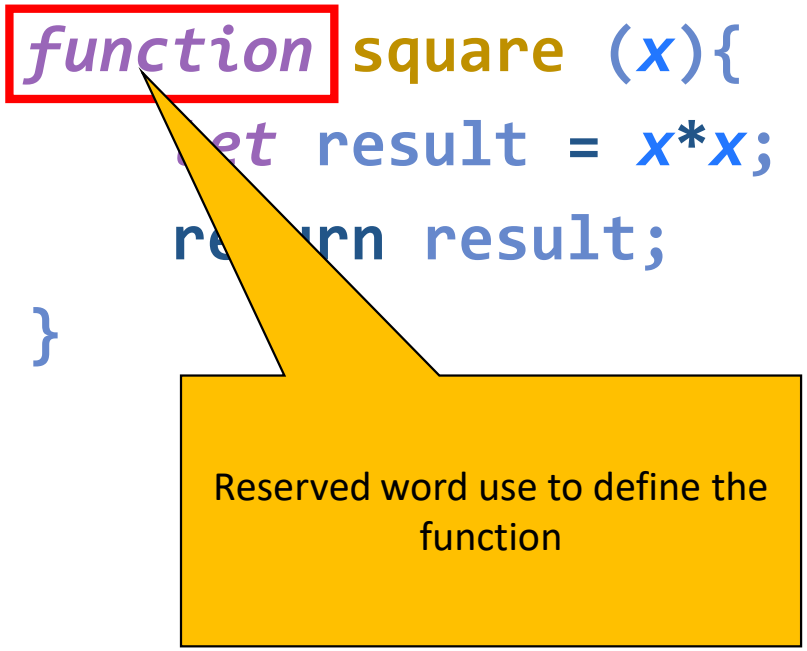
JavaScript Functions

- A JavaScript function is a block of code designed to perform a particular task.
- They're often called *methods*.
- The code inside the function will execute when "something" invokes (calls) it:
  - When an event occurs (when a user clicks a button)
  - When it is invoked (called) from JavaScript code
  - Automatically (self invoked)

```
function myFunction (a, b) {
    return a * b;
}

let x = myFunction(4, 3);
```

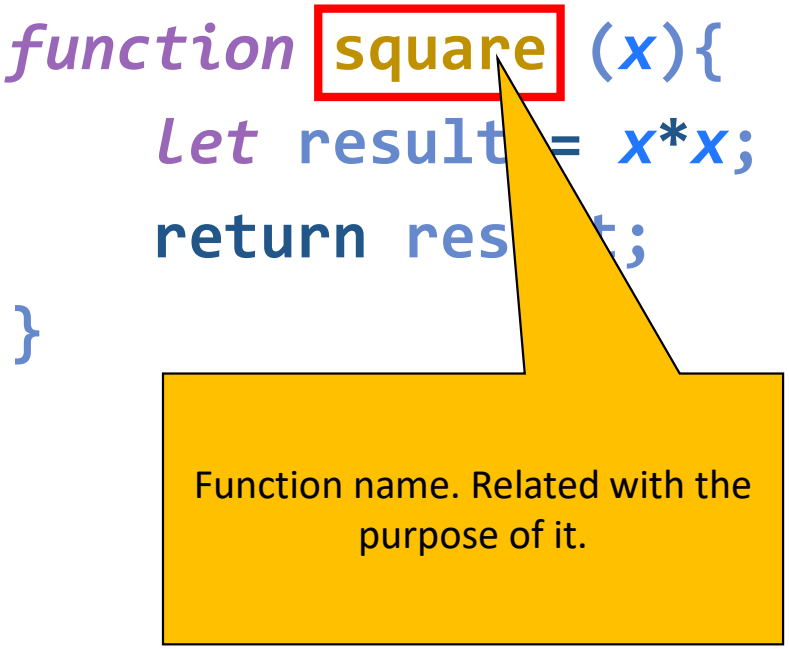
JavaScript Functions

- Parts of a function:
  - **Name:** Identify what the function does.
  - **Input data/Parameters:** Information we need to give to get the function working.
  - **Output data:** The result we get when the function finishes.
  - **Body of the function:** Instructions/steps to run from the beginning until the end.



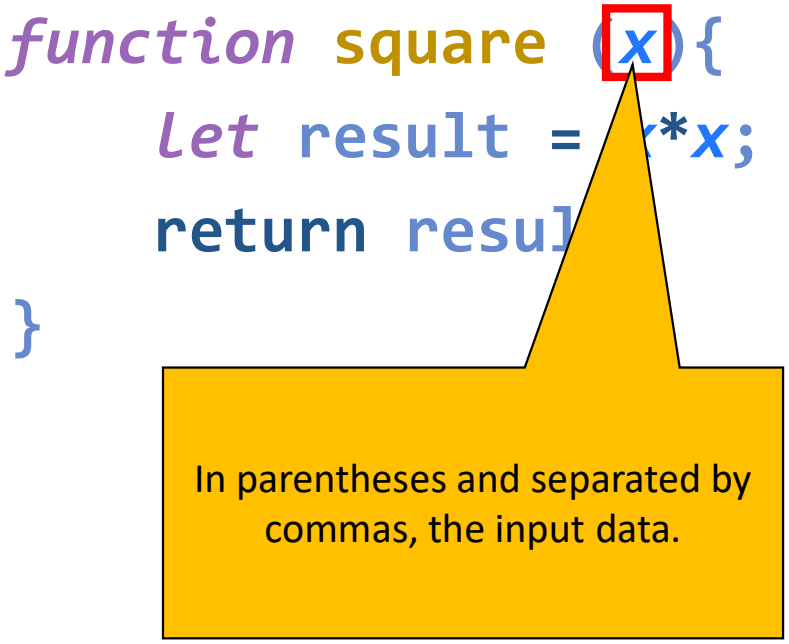
JavaScript Functions

- Parts of a function:
  - **Name:** Identify what the function does.
  - **Input data/Parameters:** Information we need to give to get the function working.
  - **Output data:** The result we get when the function finishes.
  - **Body of the function:** Instructions/steps to run from the beginning until the end.



JavaScript Functions

- Parts of a function:
  - **Name:** Identify what the function does.
  - **Input data/Parameters:** Information we need to give to get the function working.
  - **Output data:** The result we get when the function finishes.
  - **Body of the function:** Instructions/steps to run from the beginning until the end.



JavaScript Functions

- Parts of a function:
  - **Name:** Identify what the function does.
  - **Input data/Parameters:** Information we need to give to get the function working.
  - **Output data:** The result we get when the function finishes.
  - **Body of the function:** Instructions/steps to run from the beginning until the end.

```
function square (x){
    let result = x*x;
    return result;
}
```

In curly brackets, the body of the function.

JavaScript Functions

- Parts of a function:
  - **Name:** Identify what the function does.
  - **Input data/Parameters:** Information we need to give to get the function working.
  - **Output data:** The result we get when the function finishes.
  - **Body of the function:** Instructions/steps to run from the beginning until the end.

```
function square (x){
    let result = x*x;
    return result;
}
```

With return, we indicate what we are going to return to the calling instruction.

JS Functions (Rest parameter)

- ES6 provides a new kind of parameter so-called *rest parameter* that has a prefix of three dots (...).
- A rest parameter allows you to represent an indefinite number of arguments as an array.

```
function fn(a,b,...args) {
    code block
}

fn(1, 2, 3, "A", "B", "C");
```

The first 2 parameters will map to *a* and *b*, while the rest will be stored in the rest parameter *args* as an array:

```
a = 1
b = 2
args = [3,'A','B','C']
```

JS Functions (Rest parameter)

```
function maximum(...args) {
    let maxValue = undefined;
    for (let value of args) {
        if (!maxValue || value > maxValue) {
            maxValue = value;
        }
    }
    return maxValue;
}
console.log(maximum(5, 1, 89));
console.log(maximum(89, 23, -102, 0));
```

A variable and undetermined number of arguments

JS Functions (Rest parameter)

```
function maximum(...args) {
    let maxValue = undefined;
    for (let value of args) {
        if (!maxValue || value > maxValue) {
            maxValue = value;
        }
    }
    return maxValue;
}
console.log(maximum(5, 1, 89));
console.log(maximum(89, 23, -102, 0));
```

Inside the function, we work with the arguments as an iterable object:

```
args = [5, 1, 89]
args = [89, 23, -102, 0]
```

JS Functions (Rest parameter)

```
function maximum(...args) {
    let maxValue = undefined;
    for (let value of args) {
        if (!maxValue || value > maxValue) {
            maxValue = value;
        }
    }
    return maxValue;
}
console.log(maximum(5, 1, 89));
console.log(maximum(89, 23, -102, 0));
```

The function supports a different number of parameters

JS Functions (Spread operator)

- ES6 provides a new operator called *spread operator* that consists of three dots (...).
- It allows us to spread out elements of an iterable object such as an array, map, or set.

```
const myArray = [1, 2, 3];
const combined = [4, 5, 6, ...myArray];
console.log(combined);
```

OUTPUT: `[ 1, 2, 3, 4, 5, 6 ]`

JS Functions (Spread operator)

rest parameter

```
function maximum(...args) {
    let maxValue = undefined;
    for (let value of args) {
        if (!maxValue || value > maxValue) {
            maxValue = value;
        }
    }
    return maxValue;
}
let myArray = [89, 23, -102, 0];
console.log(maximum(...myArray));
```

=

```
console.log(maximum(89, 23, -102, 0));
```

spread operator

The *spread operator* extracts each element of the array and passes them as individuals to the function.

It's not the same as the *rest parameter*.

JS Functions (Spread operator)

Useful to concatenate arrays:

```
let numbers = [1, 2];
let moreNumbers = [3, 4];
let allNumbers = [...numbers, ...moreNumbers];
console.log(allNumbers); // [1, 2, 3, 4]
```

Useful to copy arrays:

```
let scores = [80, 70, 90];
let copiedScores = scores;          // WRONG - BOTH ARE THE SAME ARRAY
let copiedScores = [...scores]; // RIGHT - WE END UP WITH TWO ARRAYS
console.log(copiedScores);          // [80, 70, 90]
```

JavaScript functions are objects

```
function maximum(a, b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

console.log(typeof maximum);

maximum.
```

```
apply          (method) Function.apply(this: Function, t…
arguments
bind
call
caller
length
name
prototype
toString
```

- Even though technically, the type of a function is "function", in JavaScript functions are treated and behave as objects.

Functions are objects (in JS)

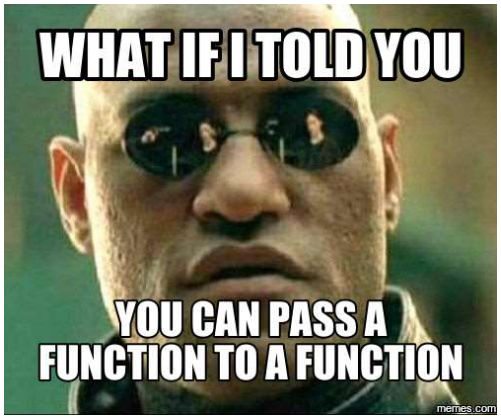
- If a function is an object…
  - **Can I store it in a variable?** → In fact, the name of the function is the name of the variable
  - **Can I assign it to another variable?** → Of course!
  - **Can I declare a variable and assign it to a function object created on the moment?** → Sure

Functions are objects (in JS)

```
let myFunction = function (param1, param2, param3) {
    return param1 + param2 + param3;
};
console.log(myFunction(1, 2, 3)); // 6

const myFunctions = [
    function () { console.log("First function"); },
    function () { console.log("Second function"); }
];
myFuncs[0]();  // Output: First function
myFuncs[1]();  // Output: Second function
console.log(typeof myFunction, typeof myFuncs[0]);   // function
```

Functions are objects (in JS)

- If a function is an object…
  - **Can I store it in a variable?** → In fact, the name of the function is the name of the variable
  - **Can I assign it to another variable?** → Of course!
  - **Can I declare a variable and assign it to a function object created on the moment?** → Sure
  - **Can I send/pass a function as a parameter to another function** → Why not?
  - **Can I return a function from a function** → That's right



JS functions are First-Class

- In JavaScript, functions are treated like any other variable. This means:
  - Functions can be **assigned to variables**.
  - Functions can be **passed as arguments** to other functions.
  - Functions can be **returned by other functions**.
  - Functions can be **stored in arrays, objects,** or other data structures, just like any other value.

```
function speak(string) {
    console.log(string);
}

speak("Hello"); // logs "Hello"

var talk = speak;
talk("Hi"); // logs "Hi"

var myFuncs = [talk];
myFuncs[0]("Bye"); // logs "Bye"
```

- In essence, functions are simply a value and are considered a special type of object.

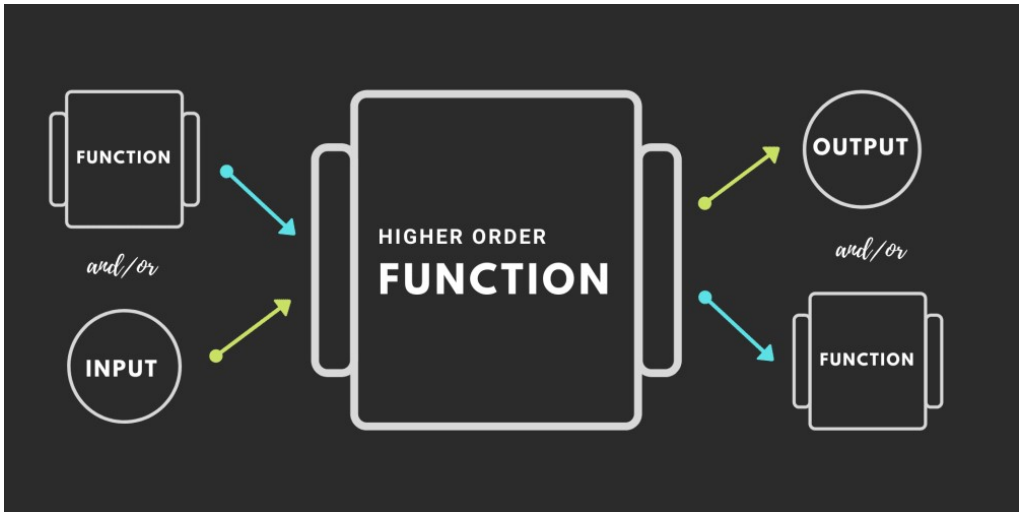
Higher-order functions

- A function can be called higher-order function if meets one or both of these conditions:
  - It takes another function as an argument.
  - It returns a function as a result.
- Higher-order functions are only possible because of the First-class functions.

```
function applyFunction(fn, val) {
    return function () {
        fn(val);
    };
}

function speak(string) {
    console.log(string);
}

var sayHi = applyFunction(speak, "Hi");
sayHi();          // logs "Hello";
```

Popular higher-order functions in JS

- There are some high order functions in JS already provided by the language and very useful.
- For the arrays we have:
  - reduce
  - map
  - forEach
  - filter
  - some
  - every



Popular higher-order functions (reduce)

```
const myArray = [1, 2, 3, 4];

const sum = myArray.reduce((accumulator, currVal) => {
    return accumulator + currVal;
});

console.log(sum); // 10
```

reduce iterates through an array and returns a single value.
On each iteration, accumulator is the value returned by the last iteration, and currVal is the current element

Popular higher-order functions (map)

```
function power2(x) {
    return x ** 2;
}

let array = [1, 2, 3, 4, 5];
let newArray = array.map(power2);
document.write(newArray);
```

map returns a new array with the result of applying the function passed as parameter to each element of the original array

Popular higher-order functions (forEach)

```
function createText(item, index) {
    text += index + ": " + item + "<br>";
}

let text = "";
const teachers = ["Jose", "Lorenzo", "Mariluz"];
teachers.forEach(createText);
document.write(text);
```

forEach applies a function to each element of the array, but does not return a new array

Popular higher-order functions (filter)

```
function even(x) {
    return x % 2 == 0;
}

let array = [1, 2, 3, 4, 5];
let newArray = array.filter(even);
document.write(newArray);
```

filter returns a new array filled with all the elements of the original array that pass a test (provided by the passed function)

Popular higher-order functions (some)

```
function even(x) {
    return x % 2 == 0;
}

let array = [1, 2, 3, 4, 5];
document.write(array.some(even));
```

some returns true if at least one of the elements of the original array passes a test (provided by the passed function)

Popular higher-order functions (every)

```
function even(x) {
    return x % 2 == 0;
}

let array = [1, 2, 3, 4, 5];
document.write(array.every(even));
```

every returns true if ALL the elements of the original array pass a test (provided by the passed function)

Passing additional parameters in higher-order functions

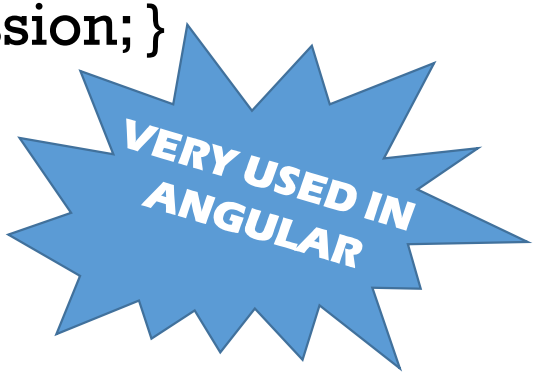
```
const notEqual = (x, y) => x !== y;
const teachers = ['Lorenzo', 'Jose', 'Mariluz'];
const res = teachers.filter(notEqual.bind(null, 'Jose'));
console.log(res); // ["Lorenzo","Mariluz"]
```

bind is used to bind values to function parameters. That's why we can pass a second parameter ('Jose') along with the main one, which we will pass as `null` since it's already passed by `filter`.

Arrow functions

- Many of the functions we are using are very simple.
- But, there's a more concise syntax for writing function expressions.
- Arrow functions allows a developer to accomplish the same result with fewer lines of code and approximately half the typing.
- Syntax: (param1, param2, paramN) => { expression; }

```
// ES5
var add = function (x, y) {
    return x + y;
};
// ES6
var add = (x, y) => { return x + y };
```



Arrow functions

- Many of the functions we are using are very simple.
- But, there's a more concise syntax for writing function expressions.
- Arrow functions allows a developer to accomplish the same result with fewer lines of code and approximately half the typing.

```
var add = (x, y) => { return x + y };
```

Function created in the same sentence.

Arrow functions

- Many of the functions we are using are very simple.
- But, there's a more concise syntax for writing function expressions.
- Arrow functions allows a developer to accomplish the same result with fewer lines of code and approximately half the typing.

```
var add = (x, y) => { return x + y };
```

Parameters of the function

Arrow functions

- Many of the functions we are using are very simple.
- But, there's a more concise syntax for writing function expressions.
- Arrow functions allows a developer to accomplish the same result with fewer lines of code and approximately half the typing.

```
var add = (x, y) => { return x + y };
```

Output of the function

Arrow functions

- Curly brackets aren't required if only one expression is present:

```
var add = (x, y) => x + y;
```

- If there's only one argument, then parentheses are not required:

```
var squareNum = x => x * x;
```

- If there're no arguments, we must include parenthesis () or underscore _:

```
var hi = () => { return 'Hello World' };
console.info( hi() );
```

Objects

Data types (complex)

Type	Example(s)

Array	`let arrayA = [1, 2, 3, 4, 5];`
	`let arrayB = new Array();`
	`let arrayC = [];`

TypedArray	`let x = new Int32Arrays(2);`
	`x[0] = 32;`
	`x[1] = -1;`

Set	`let mySet = new Set();` `x.add(4);`
Map	`let x = new Map();` `x.set('foo', 16);`
Object	`{}`

JavaScript Sets

- A JavaScript Set is a collection of unique values, where each value can only occur once.
- A Set is an object and can hold any value of any data type.
- Arrays and Sets are very similar, but Arrays can have duplicate values and Sets cannot.
- Besides, data in an array is ordered by index whereas Sets use keys and the elements are iterable in the order of insertion.

```
const mySet = new Set([1, 2, 3, 3]); // Initializes a set with 3 items
console.log(mySet.values()); // logs SetIterator {1, 2, 3}
mySet.add(4); // Successfully adds the value
mySet.add(3); // Doesn't add the value. No error thrown!
mySet.forEach((value) => {
    document.write(value + "<br>"); // Invokes a function for each Set element
})
mySet.delete(1); // Deletes the element 1
mySet.clear(); // Removes all the elements in the set
```

JavaScript Sets

- An interesting use of Sets…
- Since Sets cannot have duplicate values, we can use them to remove duplicates from an Array:

```
var initialArray = [1, 2, 1, 2, 3, 3, 4, 3, 4];
var set = new Set(initialArray);
const newArray = Array.from(set);
// Or just var newArray = [...new Set(initialArray)];
console.log(newArray); // logs 1,2,3,4
console.log(set.size); // logs 4
console.log(set.has(3)); // logs true since 3 is in the set
```

JavaScript Maps

- Maps holds key-value pairs where the keys can be any datatype (string, numbers, functions or objects).
- Maps remember the original insertion order of the keys.
- A map is the same as an associative array or a dictionary.

```
const map = new Map([[1, 10], [2, 20]]); // map = {1=>10, 2=>20}
console.log(map.get(1)); // returns 10
console.log(map.has(1)); // returns true
map.set(3, 30); // {1=>10, 2=>20, 3=>30}
map.set(1, 15); // {1=>15, 2=>20, 3=>30}
let isDeleted = map.delete(1); // {2=>20, 3=>30}
console.log(isDeleted); // true
console.log(map.size); // 2
```

JavaScript Objects

- Objects are similar to arrays, except that instead of using indexes to access and modify their data, data in objects is accessed through what are called properties.
- Objects are useful for storing data in a structured way, and can represent real world objects.
- A Literal Object is a set of zero or more pairs *name: value*.
- Example: a cat object…

```
var cat = {
    "name": "Whiskers",
    "legs": 4,
    "tails": 1,
    "enemies": ["Water", "Dogs"]
};
```

JavaScript Maps vs Objects

- Wait… From the definition of Object, you can tell that Map and Objects sound very similar.
- In fact, Objects have been used as maps until Map was added to JavaScript.
- Both are based on the same concept: using key-value pairs to store data.
- How are Maps and Objects different?
  - Keys in an Object can only be simple types (String, Symbol), but in a Map they can be any data-type including Function and Object.
  - In an Object the original order of element isn't always preserved but in a Map it is.
- Objects are better for JSON transfers, and quick and small data exchanges
- Maps are better when working with big data loads

JavaScript Objects - Logging

- Since Objects are data collections, we have a new and better way of logging them: `console.table()`
- It's compatible with Arrays too, but more useful with Objects

```
var cat = {
    "name": "Whiskers",
    "legs": 4,
    "tails": 1,
    "enemies": ["Water", "Dogs"]
};
```

(índice)	0	1	Valores
name			Whiskers
legs			4
tails			1
enemies	Water	Dogs	

JavaScript Objects - Access

- There are two ways to access the properties of an object:
  - Dot notation (.):You use Dot notation when you know the name of the property you're trying to access ahead of time.

```
var myObj = { prop1: "val1", prop2: "val2" };
var prop1val = myObj.prop1; // val1
var prop2val = myObj.prop2; // val2
```

JavaScript Objects - Access

- There are two ways to access the properties of an object:
  - Bracket notation ([]): it's similar to an array. You use it when the property of the object you are trying to access has a space in its name.

```
var myObj = {
    "Space Name": "Luke",
    "More Space": "Han",
    "NoSpace": "Millenium Falcon"
};
myObj["Space Name"]; // Luke
myObj["More Space"]; // Han
myObj["NoSpace"]; // Millenium Falcon
```

JavaScript Objects - Update

- After you've created a JavaScript object, you can update its properties at any time just like you would update any other variable.
- You can use either dot or bracket notation to update.

```
var cat = {
    "name": "Whiskers",
    "legs": 4,
    "tails": 1,
    "enemies": ["Water", "Dogs"]
};
cat.name = "Little Whiskers";
cat["name"] = "Little Whiskers";
console.log(cat.name);
```

Objects iteration – for…in loop

- The JavaScript `for...in` statement loops through the properties of an Object.
- Each iteration returns a key (*prop*) and that key is used to access its value.

```
var cat = {
    "name": "Whiskers",
    "legs": 4,
    "tails": 1,
    "enemies": ["Water", "Dogs"]
};
for (var prop in cat) {
    console.log(prop + ": " + cat[prop]);
}
```

```
name: Whiskers
legs: 4
tails: 1
enemies: Water,Dogs
```

Using Objects for Lookups

- Objects can be thought of as a key/value storage, like a dictionary.
- If you have tabular data, you can use an object to "lookup" values rather than a switch statement or an if/else chain.
- This is most useful when you know that your input data is limited to a certain range.

```
function phoneticLookup(val) {
    var result = "";
    var lookup = {
        "a": "Albacete",
        "b": "Badajoz",
        "c": "Castellon",
        "d": "Daroca",
        "e": "Estepona",
        "f": "Ferrol",
    };
    result = lookup[val];
    return result;
}
```

JavaScript Objects – Check existence

- It could be useful to check if the property of a given object exists or not.
- We can use the .hasOwnProperty(propname) method of objects to determine if that object has the given property name.
- .hasOwnProperty() returns true or false if the property is found or not.

```
var cat = {
    "name": "Whiskers",
    "legs": 4,
    "tails": 1,
    "enemies": ["Water", "Dogs"]
};

cat.hasOwnProperty("legs");  // true
cat.hasOwnProperty("color"); // false
```

JavaScript Objects - Add Properties

- You can add new properties to existing JavaScript objects the same way you would modify them.

```
var cat = {
    "name": "Whiskers",
    "legs": 4,
    "tails": 1,
    "enemies": ["Water", "Dogs"]
};

cat.color = "gray";
cat["color"] = "gray";
```

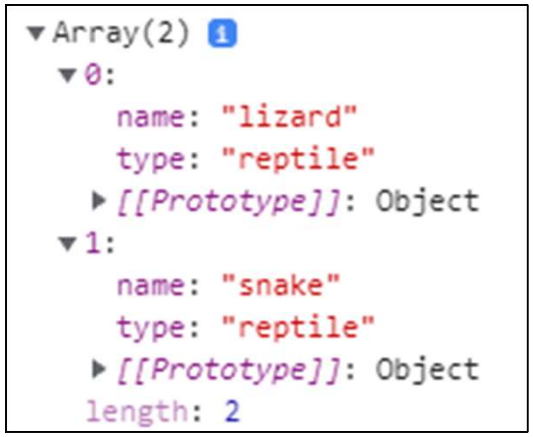
JavaScript Objects - Delete Properties

- We can also delete properties from objects.

```
var cat = {
    "name": "Whiskers",
    "legs": 4,
    "tails": 1,
    "enemies": ["Water", "Dogs"]
};

delete cat.tails;
delete cat["tails"];
```

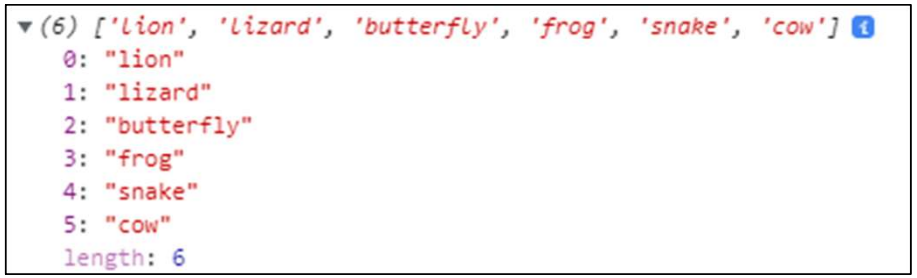
JavaScript Objects + Higher-order functions

```
let animals = [
    { name: "lion", type: "mammal" },
    { name: "lizard", type: "reptile" },
    { name: "butterfly", type: "insect" },
    { name: "frog", type: "amphibian" },
    { name: "snake", type: "reptile" },
    { name: "cow", type: "mammal" }
]

let isReptile = animal => animal.type === "reptile";

let reptiles = animals.filter(isReptile);
console.log(reptiles);
```



JavaScript Objects + Higher-order functions

```
let animals = [
    { name: "lion", type: "mammal" },
    { name: "lizard", type: "reptile" },
    { name: "butterfly", type: "insect" },
    { name: "frog", type: "amphibian" },
    { name: "snake", type: "reptile" },
    { name: "cow", type: "mammal" }
]

let names = animals.map( animal => animal.name );
console.log(names);
```



JavaScript Complex Objects

- You may want to store data in a flexible Data Structure.
- A JavaScript object is one way to handle flexible data.
- They allow combinations of strings, numbers, booleans, arrays, functions, and objects.
  - This array contains one object with various pieces of metadata about a film. It also has a nested “actors" array.
  - It’s possible to add more films to the top level array.

```
var myFilms = [
    {
      "title": "Titanic",
      "year": 1997,
      "director": "James Cameron",
      "country": "USA",
      "actors": [
        "Leonardo DiCaprio",
        "Kate Winslet",
        "Billy Zane"
      ],
      "oscars": true
    }
];
```

Nested Objects

- The sub-properties of objects can be accessed by chaining together the dot or bracket notation.

```
var ourStorage = {
    "desk": {
        "drawer": "stapler"
    },
    "cabinet": {
        "top drawer": {
            "folder1": "a file",
            "folder2": "secrets"
        },
        "bottom drawer": "soda"
    }
};
ourStorage.cabinet["top drawer"].folder2;  // "secrets"
ourStorage.desk.drawer; // "stapler"
```

Nested Arrays

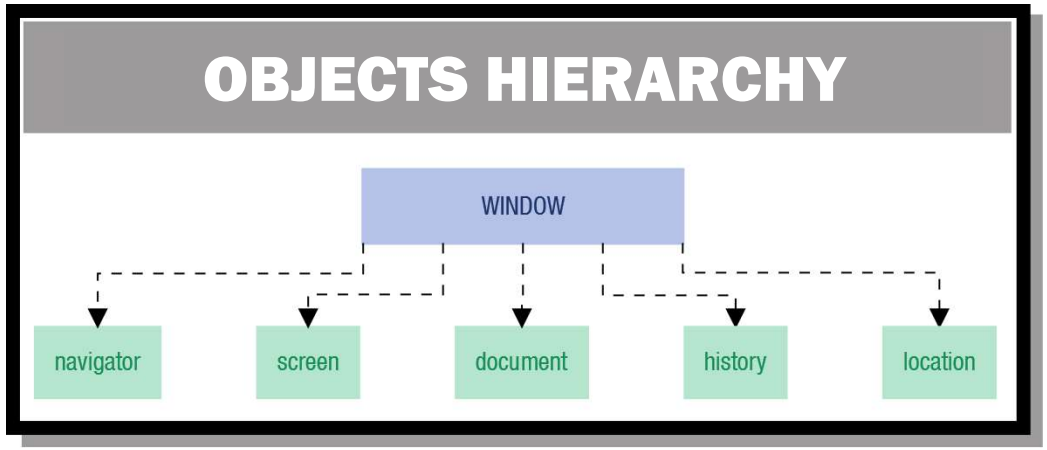
- Objects can contain both nested objects and nested arrays.
- Similar to accessing nested objects, Array bracket notation can be chained to access nested arrays.

```
var consoles = [
    {
        consoleName: "XBox",
        videogames: [
            "Forza Motorsport",
            "Halo",
            "Gears of War"
        ]
    },
    {
        consoleName: "PlayStation",
        videogames: [
            "God of War",
            "The Last of Us",
            "Uncharted"
        ]
    }
];
consoles[0].videogames[1]; // "Halo"
consoles[1].videogames[2]; // "Uncharted"
```

Surfing the DOM

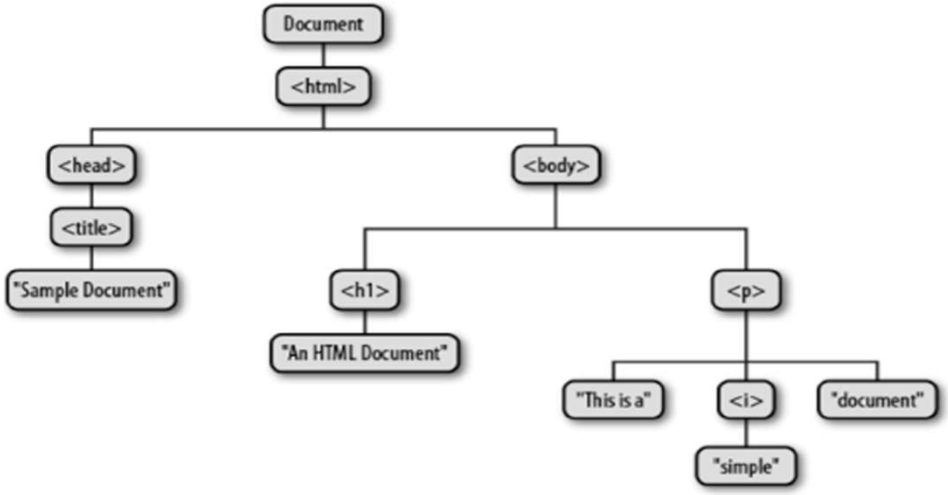
JavaScript High Level Objects

- Now that we've seen the basics of objects, let's go a little deeper into them.
- We can place Objects in most of our documents.
- This graphic represents the JavaScript high level object model…



What is the HTML DOM?

- When a web page is loaded, the browser creates a Document Object Model of the page.
- The DOM represents a document as a hierarchical tree of Objects.
- The HTML DOM is a standard for how to get, change, add, or delete HTML elements to a webpage via JavaScript.




```
<html>
    <head>
        <title>Sample Document</title>
    </head>
    <body>
        <h1>An HTML Document</h1>
        <p>This is a <i>simple</i> document</p>
    </body>
</html>
```

Window Object

- Represents an open window in a browser (for JavaScript, a browser tab is a window too).
- A Window is the main container for all the content that is displayed in the browser.
- It includes the dimensions of the window, scroll bars, toolbar, status bar, etc.
- It has properties and methods, but properties are barely used.
- All the properties and methods in W3Schools.
  - http://www.w3schools.com/jsref/obj_window.asp

Window Object

- We can access properties and methods by using the Dot notation (.):
  - window.propertyName
  - window.methodName (*[parameters]*)
- Other ways:
  - When we access a property from a document loaded on a Window:
    - self.propertyName
  - Since the Window object is always present, we can omit it when accessing objects of the same window:
    - propertyName
    - methodName

Window Object

- In fact, we're not aware that we have already used some methods of the Window object:
  - `alert()`
    - Displays an alert box with a message and an OK button
  - `confirm()`
    - Displays a dialog box with a message and an OK and a Cancel button
  - `prompt()`
    - Displays a dialog box that prompts the visitor for input

Window Object

- A script will never create the main window of a browser.
- It's the user who opens a URL in the browser or a file from the menu.
- Anyway, a script running in one of the main browser windows can create or open new sub-windows.
- The method `window.open()` generates a new window and contains up to three parameters:
  - the URL of the document to open
  - the name of the window
  - its physical appearance (size, color, etc.)

Window Object

```
var subWindow = window.open("new.html",
                            "new",
                            "height=800,width=600");
```

- We assign the new window to the *subWindow* variable.
- Doing this, we will be able to access the new window from the original script in the main window.

```
subWindow.close();
```

- We cannot use `window.close()`, `self.close()` or `close()`, since we would be trying to close the main window.

Location Object

- The Location object contains information about the current URL.
- The Location object is part of the window object and is accessed through the window.location property.
- This object has some useful methods:
  - `location.reload()`: Reloads the current document
  - `location.replace()`: Replaces the current document with a new one
- All the properties and methods in W3Schools.
  - https://www.w3schools.com/jsref/obj_location.asp

Navigator Object

- The Navigator object contains information about the client device and browser .
- This object has some useful properties:
  - `navigator.deviceMemory`: Returns the amount of RAM memory
  - `navigator.hardwareConcurrency`: Returns the number of processors
  - `navigator.language`: Returns the browser language
  - `navigator.onLine`: Returns the online status of the browser (T/F)
  - `navigator.userAgent`: Returns the browser identification, although it's not very reliable

Navigator Object

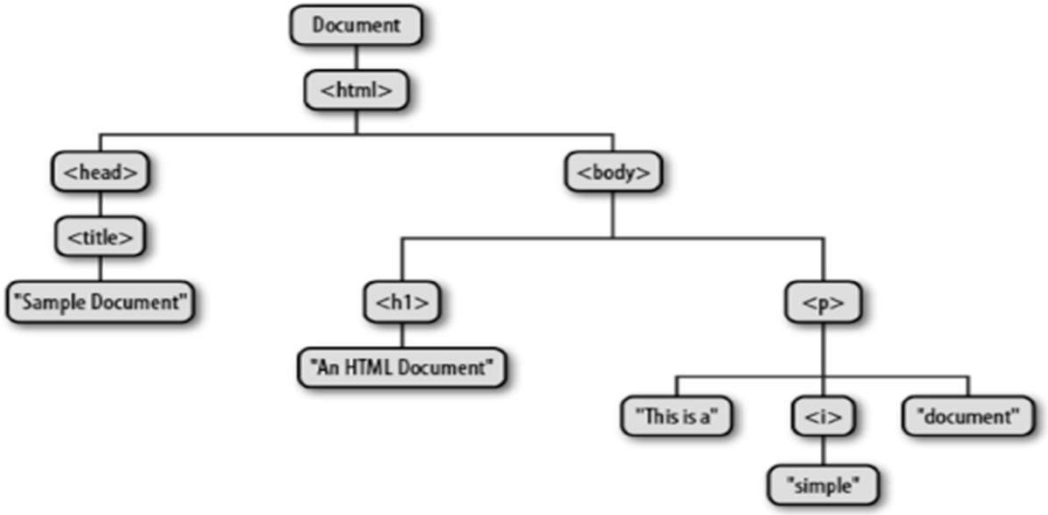
- The Navigator object has some other useful properties:
  - `navigator.geolocation`: Returns the user's global position

```
var loc = navigator.geolocation.getCurrentPosition(
    function (position) {
        console.log("Latitude: " + position.coords.latitude);
        console.log("Longitude: " + position.coords.longitude);
    })
```

- All the properties and methods in W3Schools.
  - https://www.w3schools.com/jsref/obj_navigator.asp

Document Object

- The Document object is the highest object in the DOM (Document Objects Model) structure:



Document Object

- When an HTML document is loaded into a web browser, it becomes a Document object.
- It provides properties and methods to access all node objects from within JavaScript.
- The document object is the root node of the HTML document.
- It's the object that lets you work with DOM.

Document Object

- Some very useful methods (`document.xxxxxxxxxx`):
  - `getElementById("introduction")` //returns the element with this id
  - `getElementsByName("name")` //returns an array with named elements
  - `getElementsByTagName("li")` //returns an array with all li elements
  - `getElementsByClassName("class")` //returns an array with all .class elements
  - `querySelector(".class")` //returns the first .class element
  - `querySelectorAll(".class")` //returns an array with all .class elements
  - `write() - writeln()` //writes text directly to the HTML document
  - `createElement("button")` //creates a button element

Selecting elements

```
//get all the elements named animal
    var animal = document.getElementsByName("animal");
    for (let x = 0; x < animal.length; x++) {
        console.log(animal[x]);
    }
//get the body element
    var body = document.getElementsByTagName("body");
    body[0].style.backgroundColor = "red";
//get the element with the ID "myDiv"
    var myDiv = document.getElementById("myDiv");
//get all the elements with class "selected"
    var selected = document.getElementsByClassName("selected");
```

Selecting elements

```
//get the body element
    var body = document.querySelector("body");
//get the element with the ID "myDiv"
    var myDiv = document.querySelector("#myDiv");
//get the first element with class "selected"
    var selected = document.querySelector(".selected");
//get the first image with class "button"
    var img = document.querySelector("img.button");
//same function but returning all the elements found
    var matches = document.querySelectorAll("div.note, div.comment");
    for (let x = 0; x < matches.length; x++) {
        matches[x].style.backgroundColor = "blue";
    }
```

- All the properties and methods in W3Schools.
  - http://www.w3schools.com/jsref/dom_obj_document.asp

Element Object

- It represents an HTML element like P, DIV, A, TABLE…
- It can have child nodes, siblings and father.
- It's the object that lets you work with DOM.

Element Object

- The Element object has some very useful methods and properties:
  - `innerHTML //Access the HTML content of an element`
  - `innerText //Access the inner text of an element`
  - `click() //Simulates a mouse-click on an element`
  - `blur() //Removes focus from an element`
  - `appendChild() //Appends a node as the last child of a node`
  - `parentElement() //Returns the parent element of the specified element`
  - `remove() //Removes the specified element from the document`

- All the properties and methods in W3Schools.
  - https://www.w3schools.com/jsref/dom_obj_all.asp

Element attributes

- We can also access the attributes of a DOM element.
- To reference the attributes of a DOM element, such as an `img` element, we will use the `attributes` collection.

```
<img id="solvamLogo" src="solvam.jpg" width="120" height="150">

let attrs = document.getElementById("solvamLogo").attributes;
console.log(attrs.length); // we can loop through the attributes
for (let i = 0; i < attrs.length; i++) {
    console.log(attrs[i].name + " = " + attrs[i].value + "\n");
}
```

```
4
id = solvamLogo
src = solvam.jpg
width = 120
height = 150
```

Get, Set and Remove attributes

- The `setAttribute()` method will allow us to create or modify attributes of an element.

```
let img = document.getElementById("solvamLogo");
img.setAttribute("width", "200"); // changes the width
img.setAttribute("alt", "Logo"); // creates and sets alt
```

- To check the value of the attribute and not modify it, we can use `getAttribute()` or the dot notation:

```
var height = img.getAttribute("height"); // gets the height
var width = img.width; // gets the width
```

- Finally, to remove an attribute, we can do it with `removeAttribute()`:

```
img.removeAttribute("width"); // removes the width
```

Creating an Element Object

- To add a new Element object we have to follow these steps:
  - Create the Element object:

```
var link = document.createElement("a");
```

  - Set their properties:

```
link.innerText="Go to Solvam";
link.href="https://www.solvam.es";
link.target="_blank";
```

  - Select the parent element that will contain it:

```
var body = document.querySelector("body");
```

  - Append it into its parent as a child:

```
body.appendChild(link);
```

User-defined objects

- As we already know, we can create our own objects with properties and functions.

```
var person = new Object();
person.name = "Lola";
person.age = 29;
person.job = "Software Engineer";
person.sayName = function () {
    alert(this.name);
};
```

```
var person = {
    name = "Lola",
    age = 29,
    job = "Software Engineer",
    sayName = function () {
        alert(this.name);
    }
};
```

More objects

- Location: Redirections

```
alert("Going to amazon.es");
location.assign("https://www.amazon.es");
```

- Date: Managing dates

```
var today = new Date(); //creates Date object with current date
alert ( today.toDateString() ); //displays Sun Oct 17 2021
alert ( today.getFullYear() ); //displays 2021
alert ( today.getDate() ); //displays 17
alert ( today.getMonth() ); //displays 9, not 10 for October
```

- And much more…

Nodes relationships

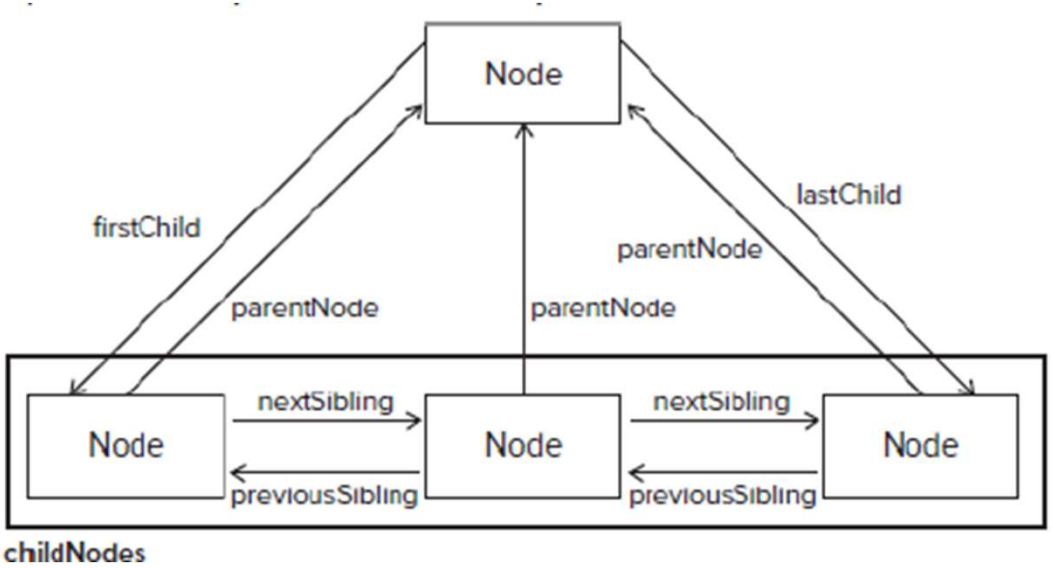
- All nodes in a document have relationships to other nodes.
  - Family like (father, sibling, children, etc.)
  - Each node has the following properties:
    - `parentNode/parentElement`
    - `childNodes`
    - `firstElementChild`, `lastElementChild`
      - `firstChild`, `lastChild (useful with texts or comments)`
    - `nextElementSibling`, `previousElementSibling`
      - `nextSibling`, `previousSibling (just like child properties)`
    - `nodeType`
    - `nodeValue`
    - `nodeName`

Nodes relationships

- Difference between `firstChild` and `firstElementChild`:

```
<ul><!--This is a comment node-->
    <li> A </li>
    <li> B </li>
    <li> C </li>
</ul>

const ul = document.querySelector('ul');
console.log(ul.firstChild);
    // returns the comment node <!--This is a comment node-->
console.log(ul.firstElementChild);
    // returns the element child <li> A </li> RECOMMENDED!
```

Nodes relationships



Nodes relationships

Changing the DOM

- Creating elements
  - A HTML link element:

```
<link rel="stylesheet" type="text/css" href="styles.css">
```

  - Using JS:

```
var link = document.createElement("link");
link.rel="stylesheet";
link.type="text/css";
link.href="styles.css";
var head = document.getElementsByTagName("head")[0];
head.appendChild(link);
```

Changing the DOM

- Inserting and removing elements
  - As the last child
    - `someNode.appendChild(someNode);`
  - In a specific location
    - `parentNode.insertBefore(node, positionNode);`
  - Replacing node
    - `someNode.replaceChild(newNode, oldNode);`
  - Removing node
    - `someNode.remove();`
      `parentNode.removeChild(childNode);`

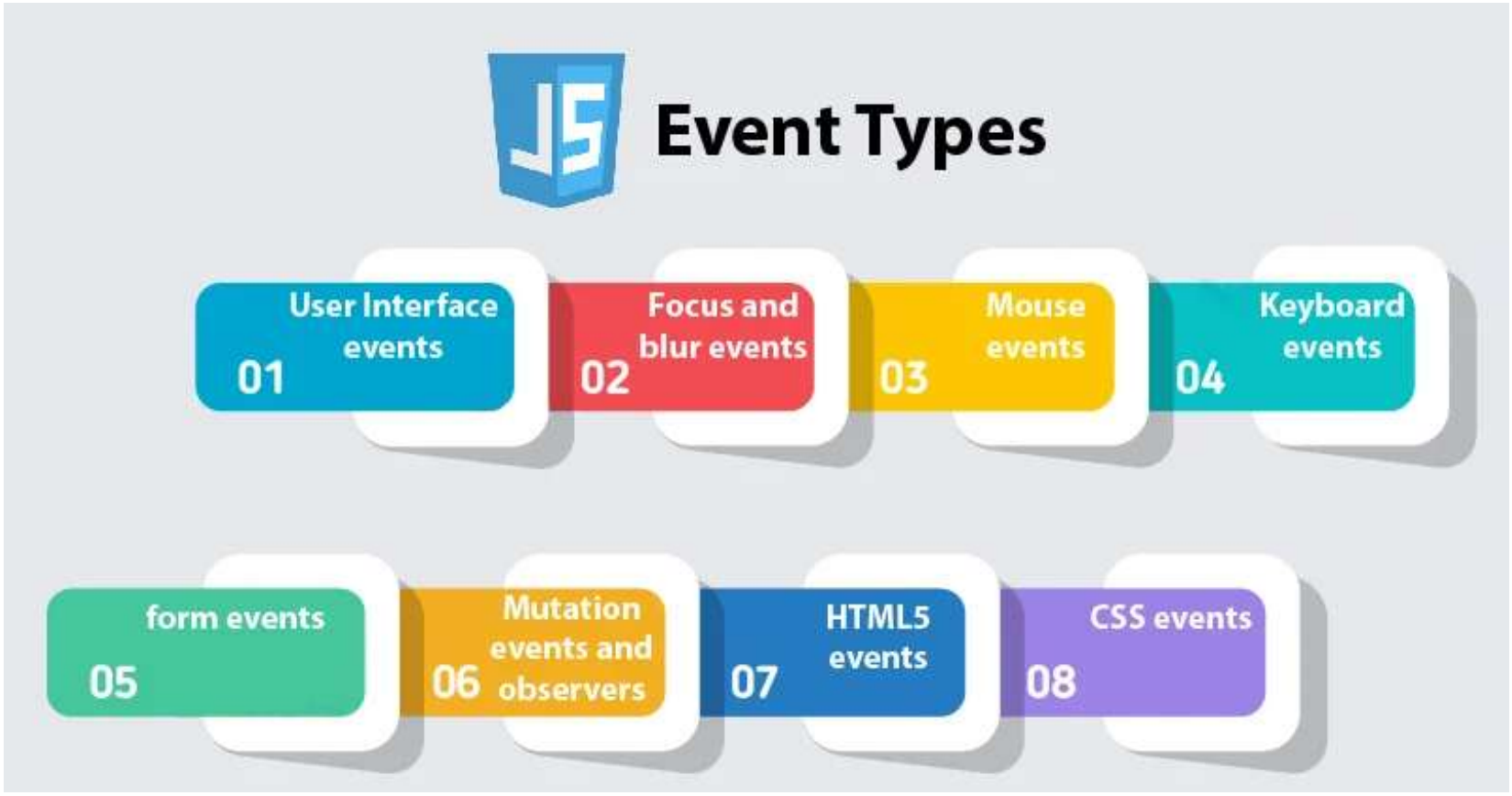
Managing events

Events

- JavaScript applications commonly respond to user actions like clicking on a button.
- These actions are called events, and the anonymous functions that handle the events are called event handlers.
- To make that happen, you have to attach the function to the events.

Events

- There are several event types that can occur in a web browser.
  - **User interface events:** interactions with the browser window.
  - **Focus and blur events:** when HTML elements gain or lose focus.
  - **Mouse events:** fired by the mouse (buttons, wheel or movement)
  - **Keyboard events:** when the user interacts with a keyboard.
  - **Form events:** the user interacts with a form.
  - **Mutation events and observers:** addition or removal of a DOM node through code.
  - **HTML5 events:** page-level events such as gestures and movements.
  - **CSS events:** when the script encounters a CSS element.



http://www.w3schools.com/jsref/dom_obj_event.asp

Events

- Some of the most useful events are related to forms.
- They're triggered by actions within a form, and although they apply to almost all HTML elements, they're primarily used in forms.
  - `onfocus`
  - `onblur`
  - `oninput`
  - `onsubmit`
  - `onchange`

Events

- Another set of useful events are those triggered by the mouse and the keyboard:

Mouse Events

- `onclick`
- `ondblclick`
- `ondrag`
- `ondrop`
- `onmousedown`
- `onmouseup`
- `onmouseover`
- `onmouseout`
- `onmousemove`

Keyboard Events

- `onkeydown`
- `onkeypress`
- `onkeyup`

Inline/HTML Event Handlers

```
function myFunction() {
    alert("Hello World");
}
```


```
<button type="button" id="btn1" onclick="myFunction()">Button</button>
```

- This is bad-practice, but since we see it in older browsers, we need to be aware of it.
- This method is no longer in use; it's always better to separate the JavaScript from the HTML.

Traditional DOM Event Handlers

```
function myFunction() {
    alert("Hello World");
}
```

To keep our HTML clean, and our JS easier to maintain and scale, we can use event handlers.

```
var btn = document.getElementById("btn1");
btn.onclick = myFunction; //no parentheses
```

We can call an existing function…

```
btn.onclick = null;
```

We can remove the handler…

```
var btn = document.getElementById("btn1");
btn.onclick = function () {
    alert("Hello World");
}
```

Or create the function inside the call itself if it's a one-time function. It's called ***anonymous function*** since it has no name

DOM2 Event Handlers

```
function myFunction() {
    alert("Hello World");
}
```

```
function myFunction2(msg) {
    alert(msg);
}
```

Even better…

```
var btn = document.getElementById("btn1");
btn.addEventListener("click", myFunction);
```

We can add an event listener to our event

```
btn.removeEventListener("click", myFunction);
```

Or remove it

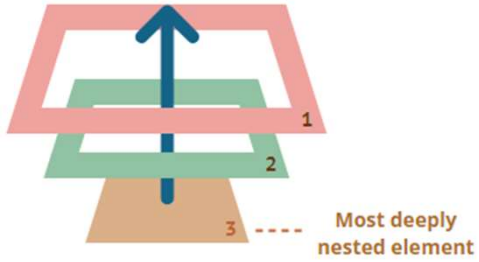
```
btn.addEventListener("click", function(){
    myFunction2("Hello World");
});
```

If our function accepts parameters, we can wrap the function call in an anonymous function

Event bubbling

- The bubbling principle is simple: when an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.

```
<form>FORM
    <div>DIV
        <p>P</p>
    </div>
</form>
```




```
var formElem = document.querySelector("form");
formElem.onclick= function() {
    alert("You clicked the form");
}
var divElem = document.querySelector("div");
divElem.onclick = function() {
    alert("You clicked the div");
}
var pElem = document.querySelector("p");
pElem.onclick = function() {
    alert("You clicked the p");
}
```

Event Target vs This

- If we want to know where did the event actually happened, we can use an object called `event`.
- The most deeply nested element that caused the event is called a target element, accessible as `event.target`.

```
var formElem = document.querySelector("form");
formElem.onclick= function(event) {
    alert("Type: " + event.type                    // click
          + " Target: " + event.target.tagName     // P-DIV-FORM
          + " This: " + this.tagName);             // FORM
}
```

  - `event.target` – is the "target" element that initiated the event. It doesn't change through the bubbling process.
  - `this` – is the "current" element, the one that has a currently running handler on it.
- It will be very useful when working with dynamic elements.

The DOMContentLoaded Event

- Until now, when we need to access HTML elements that are already written in the *body* section of the document from JavaScript, we have used the `window.onload` event.
- Now we have another useful event: the document DOMContentLoaded event.
  - The window `load` event is fired when the entire web page has been loaded, including the page DOM and all dependent resources (scripts, stylesheets, and images).
  - The document `DOMContentLoaded` event is fired when the page DOM has been loaded and all deferred scripts have been loaded and executed.
- If our JavaScript code only needs to interact with the DOM (modify elements, set up event listeners…), we should use the DOMContentLoaded event.
- If our JavaScript code depends on resources like stylesheets and images, then using the load event is a better option.

```
document.addEventListener('DOMContentLoaded', () => {
    ...
}
```

Data Attributes

Data Attributes

- It's very usual to use element class names to store fragments of metadata for the sole purpose of making our JS simpler.
- For instance…
  - We have a list of different restaurants on a webpage.
  - We want to store information about the type of food offered by restaurants or their distance from the visitor.
  - We usually use the HTML class attribute to store that info:

```
<li class="chinese">Li Feng</li>
<li class="pizza">Don Mario</li>
<li class="burger">Meat & Greet</li>
<li class="pizza">Pasta e Pomodoro</li>
```

Data Attributes

- Thanks to HTML5, we now have the ability to embed custom data attributes on all HTML elements.

Data Attributes

- Custom data attributes consists of two parts:
  - Attribute name: must be prefixed with data- and has to be at least one character long.
  - Attribute value: can be any String (except for capital letters).

```
<li data-food-type="chinese" data-id="3781">Li Feng</li>
<li data-food-type="pizza" data-id="8104">Don Mario</li>
<li data-food-type="burger" data-id="6382">Meat & Greet</li>
<li data-food-type="pizza" data-id="9267">Pasta e Pomodoro</li>
```

Data Attributes

- An element can have any number of data attributes with any value you want.
- Be aware that users can see these data so don't add sensitive information in the attribute.
- In the browser, you won't see any difference in the webpage using these attributes.

JavaScript Access to Data Attributes

- We could use `getAttribute()` to read them.
- Instead, we can use the `dataset` object, get the property by the part of the attribute name after `data-` (if any dashes are present they will be converted to camel case).

```
const restaurants = document.querySelectorAll("li");
restaurants.forEach(restaurant => {
    console.log(restaurant.dataset.foodType);
    console.log(restaurant.dataset.id);
});
```

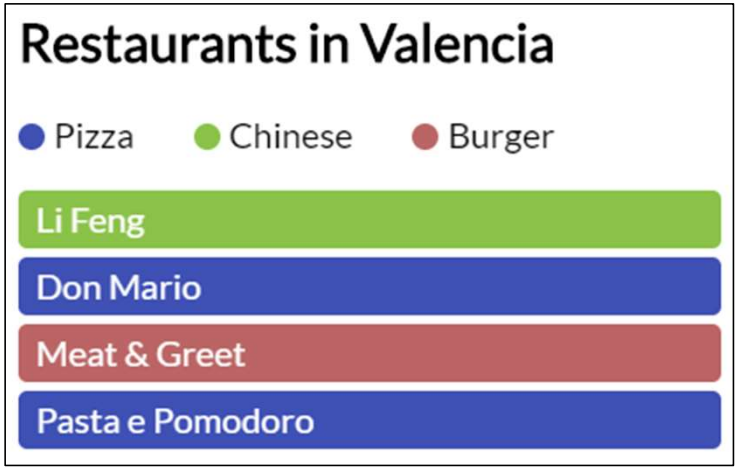
- Each property is a string and can be read and written.

```
restaurants[0].dataset.foodType = "sushi";
```

CSS Access to Data Attributes

- In CSS, it's posible to use the attribute selectors to change styles according to the data
- We can give users a cue about the type of restaurant using attribute selectors to style the restaurants differently.

```css
li[data-food-type="chinese"] {
    color: #8BC34A;
}
li[data-food-type="pizza"] {
    color: #3F51B5;
}
li[data-food-type="burger"] {
    color: #BB6666;
}
```



Use Data Attributes with Event Handlers

- You can use data attributes in event handlers to pass specific information related to the clicked element.

```html
<button data-action="edit" data-id="8104">Save</button>
<button data-action="delete" data-id="8104">Delete</button>

<button data-action="edit" data-id="3781">Save</button>
<button data-action="delete" data-id="3781">Delete</button>
```

```javascript
buttons.forEach(button => {
    button.addEventListener('click', function () {
        const action = this.dataset.action; // Get action (edit or delete)
        const id = this.dataset.id; // Get the id

        console.log(`Action: ${action}, ID: ${id}`);
    });
});
```

Remove Data Attributes

- We can remove any attribute, including data-* attributes, using the removeAttribute() method:

```
const restaurants = document.getElementsByTagName("li");
restaurants[0].removeAttribute("data-food-type");
```

- We can also remove a data-* attribute by using the `delete` keyword on the `dataset` property:

```
const restaurants = document.getElementsByTagName("li");
delete restaurants[0].dataset.foodType;
```

Forms and Validation

Forms in JavaScript

- Forms are the main means of data input in a web application, and a source of interactivity with the user.
- Forms and their controls (input, select, textarea…) are DOM objects that have unique properties that other objects do not (action, selected…).
- JavaScript adds two very interesting features to forms:
  - It allows us to **examine** and **validate** user input directly, on the client side.
  - It allows us to give **instant messages**, with information from the user's input.

Form selection

- Within a document we will have several ways to select a form:

```
<div id="sidemenu">
    <form id="contact" name="contactform" action="...">...</form>
</div>

var myForm = document.getElementById("contact");
var myForm = document.getElementsByName("contactform")[0];
var myForm = document.getElementsByTagName("form")[0];
var myForm = document.getElementsByTagName("form")["contactform"];
var myForm = document.forms[0];
var myForm = document.forms["contactform"];
```

Form Elements selection

- We can access the elements of a form in different ways:

HTML

```html
<form name="loginForm" action="#" method="post">
    Username: <input type="text" name="uname" id="username" >
    Password: <input type="password" name="pw" id="password" >
    <input type="submit" value="Submit">
</form>
```

JS

```js
var user = document.getElementById("username");
var user = document.getElementsByTagName("input")[0];
var user = document.querySelector("#username");
var user = document.loginForm.uname;
var myForm = document.forms[0]; // or any other way of selecting the form
var user = myForm.uname;
var user = myForm.elements[0];
```

Form Elements properties

- Once an element is selected, we can access its HTML properties to read or modify them:

HTML

```
<form name="loginForm" action="#" method="post">
    Amount: <input type="number" name="amount" id="amount" size="6">
    <input type="submit" value="Submit">
</form>
```

JS

```
var amount = document.getElementById("amount");
console.log(amount.size); // size property
console.log(amount.name); // name property
console.log(amount.value); // value property = entered number
```

Form Validation

- Data validation is the process of ensuring that user input is clean, correct, and useful.
- Typical validation tasks are:
  - has the user filled in all required fields?
  - has the user entered a valid date?
  - has the user entered text in a numeric field?
- Most often, the purpose of data validation is to ensure correct user input.

Form Validation - Example

- When we click the "Submit" button in a form, a JS function can be called to validate the input data:

HTML

```
<form name="loginForm" action="#" onsubmit="return valForm()" method="post">
    Username: <input type="text" name="uname" id="username">
    <input type="submit" value="Submit">
</form>
```

JS

```
function validateForm() {
    var x = document.getElementById("username").value;
    if (x == "") {
        alert("You must enter a username");
        return false;
    }
}
```

Form Validation

- How to check different form elements:
  - input and text elements (text, password, button, email, textarea…):
    - **HTML:**
```
<input type="text" id="myText">
<input type="password" id="myPW">
<textarea id="myTextArea">
```
    - **JS:**
```
document.getElementById("myText").value
document.getElementById("myPW").value
document.getElementById("myTextArea").value
```

Form Validation

- How to check different form elements:
  - checkbox/radio elements:
    - **HTML:**

```
<input type="radio" name="shift" value="D">Day
<input type="radio" name="shift" value="N">Night
<input type="checkbox" name="sauce" value="S">Spicy
<input type="checkbox" name="sauce" value="B">BBQ
```

    - **JS:**

```
document.getElementsByName("shift")
    .forEach(radio => {
        if (radio.checked) {
            console.log(radio.value);
        }
    });
```

```
document.getElementsByName("sauce")
    .forEach(check => {
        if (check.checked) {
            console.log(check.value);
        }
    });
```

Form Validation

- How to check different form elements:
  - select-options elements:
    - **HTML:**
```
<select id="mySelect">
  <option value="1">Jose</option>
  <option value="2" selected>Lola</option>
  <option value="3">Lorenzo</option>
</select>
```
    - **JS:**
```
var x = document.getElementById("mySelect");
x.value; // 2
x.options[x.selectedIndex].text; // Lola
```

Use RegEx In JavaScript

- Regular expressions (RegEx) are patterns used to match character combinations in strings.
- We can use regular expressions in our code to validate the entered values in a form.
- These patterns are used with the JavaScript test() method.

```
const lowerOnly = /^[a-z]+$/g;
console.log(lowerOnly.test(string));
```

- `[a-z]`: only lowercase characters are permitted
- `^`: start the string with something from [a-z]
- `+`: there can be one or more occurrences from the range [a-z].
- `$`: the string must end with something from [a-z]
- `test()` returns `true` or `false`

Use RegEx In JavaScript

```
function ValidateEmail() {
    var emailReg = /^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/;
    var email = document.getElementById("email").value;

    if (email == "" || !emailReg.test(email)) {
        alert("You have entered an invalid email address!");
        return false;
    } else {
        alert("Valid email address!");
        return true;
    }
}
```

Form Validation

- HTML5 introduced a new HTML validation concept called constraint validation.
  - Constraint validation HTML Input Attributes:
    `max, min, required, type …`
  - Constraint validation CSS Pseudo Selectors:
    `:disabled, :optional, :valid …`
  - Constraint validation DOM Properties and Methods:
    `validity, checkValidity() …`

https://www.w3schools.com/js/js_validation_api.asp

Form Validation – DOM Methods

- If an input field contains invalid data, display a message:

```
<input id="age" type="number" min="18" max="65" required>
<button id="btnCheck">OK</button>
<p id="val"></p>
<script>
   function checkAge() {
      var ageValue = document.getElementById("age");
      if (!ageValue.checkValidity()) {
         document.getElementById("val").innerHTML = ageValue.validationMessage;
      }
   }
   var btn = document.getElementById("btnCheck");
   btn.onclick = checkAge;
</script>
```

Form Validation – DOM Properties

- If an input field contains invalid data, display a message:

```
<input id="num" type="number" max="100" min="0">
<button onclick="check()">OK</button>
<p id="val"></p>
<script>
    function check() {
        var txt = "";
        var numElement = document.getElementById("num");
        if (numElement.validity.rangeOverflow) {
            txt = "Value too high";
        } else if (numElement.validity.rangeUnderflow) {
            txt = " Value too low";
        }
        document.getElementById("val").innerHTML = txt;
    }
</script>
```

Questions?