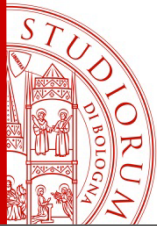


# JavaScript



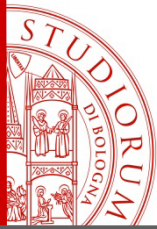
**2 domande  
bonus**  **BONUS**



# Agenda

---

- Venerdì 24 novembre:
  - turno unico, dalle 9 alle 13 (PHP)
- Martedì 28 novembre:
  - seminario su «*Elementi di Human-Computer Interaction e User interface per il Web in culture orientali*»
- Venerdì 1 dicembre:
  - Turno unico, dalle 9 alle 12 (BootStrap)
- Martedì 5 dicembre:
  - Seminario su «Search Engine Optimization e Accessibilità»



# Job Day

**job** day  
**2023**

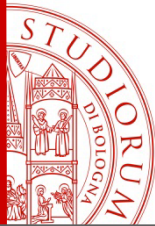
**22 NOVEMBRE**



**DALLE 13.30 ALLE 18.00**

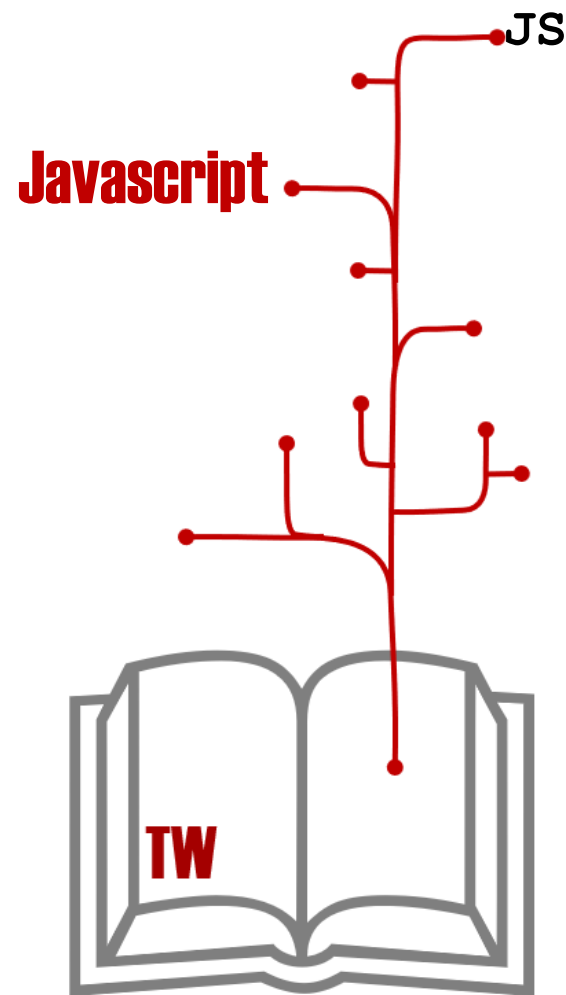
**LAB. 3.5 - LAB. 3.6 - POLIVALENTE**

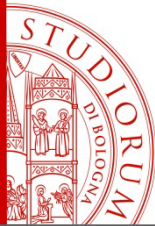
**[EVENTI.UNIBO.IT/JOBDAY](https://eventi.unibo.it/jobday)**



# Argomenti

- JavaScript:
  - Obiettivi e Standardizzazione
  - Caratteristiche generali
  - AJAX
  - JSON
  - DOM





# JavaScript

- JavaScript è un linguaggio di **scripting interpretato** dal browser e basato su una logica **weakly typed** e **prototype-based**
  - Insieme ad HTML and CSS, JavaScript è una delle tre core technologies nella produzione di contenuti Web

# JavaScript

- Fu originariamente sviluppato dalla Netscape come Mochan prima e come **LiveScript** poi.
- Il nome fu poi cambiato in **JavaScript** nel 1995 (attraverso un accordo Netscape con Sun) nello stesso periodo in cui Netscape iniziò a includere nel proprio browser il supporto per Java

When the world still made sense

JavaScript first appeared in 1995

Programmers in 1994 :

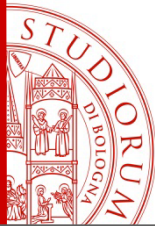


# JavaScript vs Java



- La scelta di questo nome è ancora oggi fonte di **grande** confusione:
  - **Non c'è una vera relazione tra Java e JavaScript**
  - Le somiglianze sono soprattutto sintattiche (causate dal fatto che entrambi i linguaggi si ispirano al C)
  - In particolare gli object model non hanno relazione e sono ampiamente incompatibili





# JavaScript vs Java

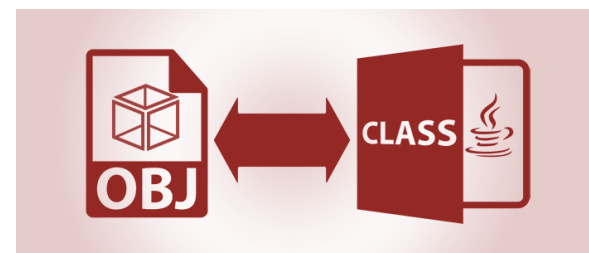
- PRINCIPALI differenze tra Java e Javascript
  - JavaScript è **interpretato** non compilato
  - JavaScript è **object-based** o meglio **prototype-based** ma **NON** class-based
    - Esiste il concetto di oggetto
    - Non esiste il concetto di classe
  - JavaScript è debolmente tipizzato (**weakly typed**): Non è necessario definire il tipo di una variabile
- **SI USANO IN CONTESTI DIVERSI!**





# Linguaggi class-based

- I linguaggi object-oriented che avete visto fino ad ora sono linguaggi **class-based** (C++, Java, C#, SmallTalk, ecc).
- Nei linguaggi class-based:  
la classe esiste come concetto esplicito e primario:
  - le classi formano una gerarchia di tipi, l'ereditarietà avviene tra classi, gli oggetti sono istanze pure delle classi (non hanno metodi propri).
  - Il design delle interfacce precede ed è strumentale alla creazione degli algoritmi per la esecuzione dei compiti dell'applicazione.

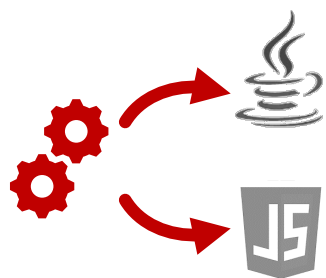




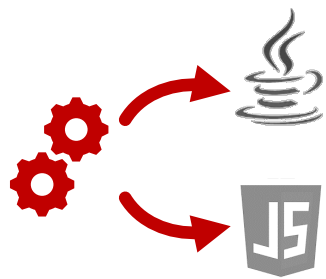
# Linguaggi prototype based

- JavaScript (e ovviamente anche ECMAScript e i linguaggi derivati) è **prototype-based**.
- Nei linguaggi prototype-based:
  - non esiste il concetto di classe, ma quello di **prototipo, una istanza primaria, astratta, sempre accessibile e modificabile, di cui le singole istanze clonano (e, se serve, modificano) sia membri sia metodi.**
  - Il design delle interfacce è contemporaneo e indipendente dalla creazione degli algoritmi, **e può essere modificato in qualunque momento**, anche a run-time per garantire massima flessibilità.

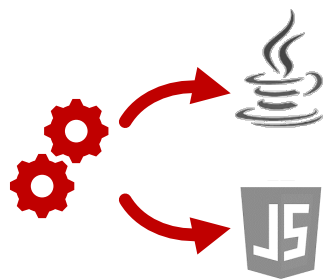
# Sostanzialmente...



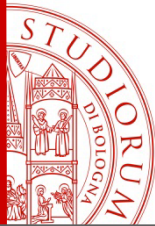
- In Java si definisce una classe e da questa si dichiara un oggetto
- In JavaScript ogni oggetto è creato direttamente, senza bisogno di definire prima una classe.



- In Java si possono avere sottoclassi definite a partire da una classe
- In JavaScript è possibile definire un prototipo con il quale un oggetto prende a modello un altro oggetto condividendone le caratteristiche



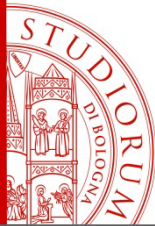
- In Java la classe definisce tutte le caratteristiche che può avere un oggetto
- In JavaScript è possibile specificare le caratteristiche di un oggetto anche aggiungendo proprietà e metodi a runtime



# JavaScript: Prototype

---

- Ogni oggetto in JavaScript è autonomo e si possono aggiungere tutti i metodi/proprietà che si vuole senza modificare gli altri.
- Per aggiungere proprietà/metodi a molti oggetti si deve usare l'oggetto **prototype**.
- Prototype si usa per creare o riusare librerie di oggetti e metodi:
  - Per estendere le proprietà di un oggetto built-in nel linguaggio
  - Per estendere le proprietà di oggetti creati in precedenza
  - Ogni oggetto JavaScript ha una proprietà prototype a cui si può aggiungere un membro ed associare una funzione
  - La modifica del prototipo può avvenire in qualunque momento nell'esecuzione del programma.



# Esempio di prototype

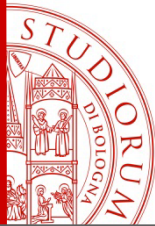
---

```
// costruttore oggetto Persona
function Persona(nome, cognome) {
    this.firstName = nome;
    this.lastName  = cognome;
}

var pinco = new Persona("Pinco", "Pallino");

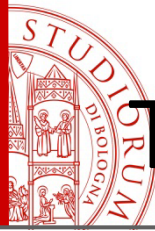
Persona.prototype.welcome = function(){
    alert("Benvenuto, "+this.firstName + "!");
} // Aggiunta di una funzione al prototipo Persona

pinco.welcome(); // Utilizzo della nuova funzione
```



# ECMAScript

- Nel 1997 una versione standardizzata di JavaScript alla specifica di **ECMAScript**, uno standard ECMA (European Computer Manufacturers Association) (ECMA-262)
  - È anche uno standard ISO (ISO/IEC 16262)
  - È arrivato alla versione 14 ECMAScript 2023 (ES2023) del Giugno 2023: <https://262.ecma-international.org/>
  - è implementato come linguaggio embedded nei browser da JavaScript o, nel caso di Internet Explorer, da JScript.
- Le implementazioni (a volte) introducono delle **estensioni del linguaggio** o della libreria standard e delle relative API.



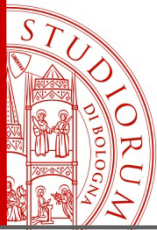
# The 2023 Top Programming Languages

## The 2023 Top Programming Languages

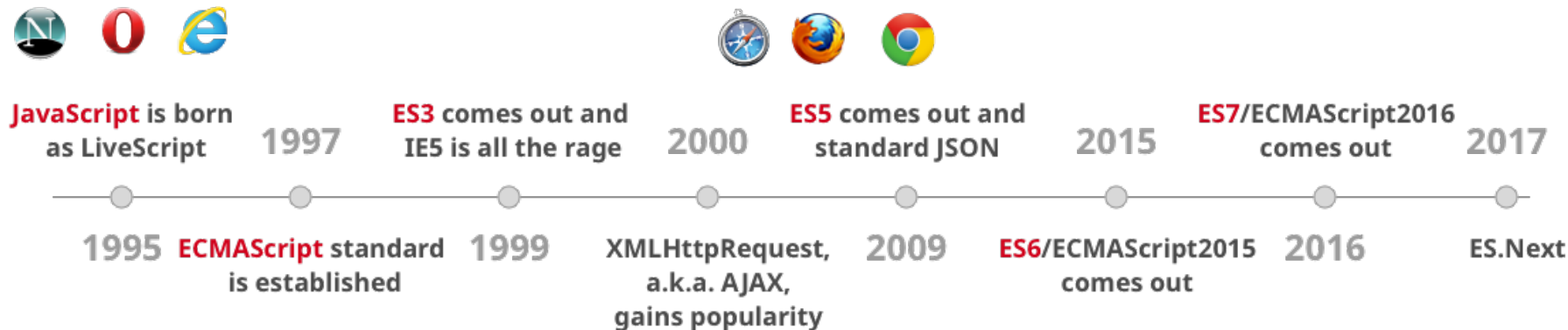
<https://spectrum.ieee.org/top-programming-languages/>







# L'evoluzione di Javascript

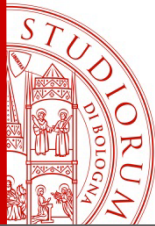


- Javascript nasce come linguaggio client side interpretato dal browser per realizzare semplici script nella pagina (in origine erano spesso usati per inserire effetti grafici e per dare dinamismo al contenuto web)

# L'evoluzione di Javascript

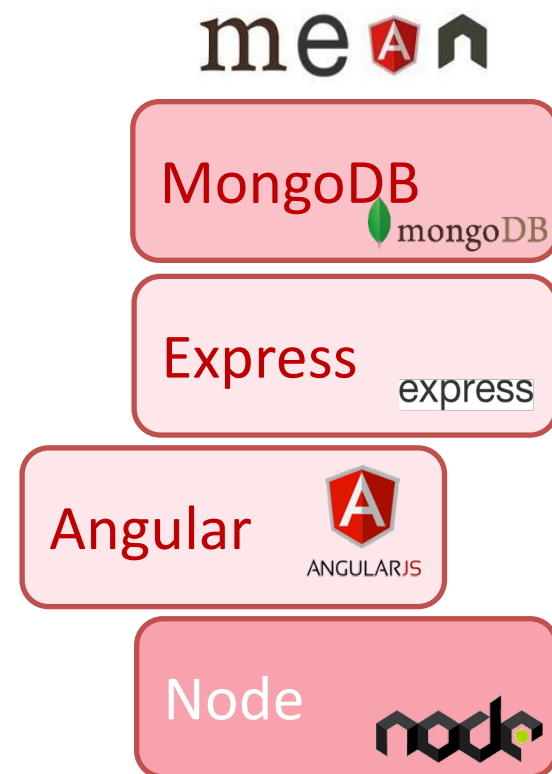
- Ma cosa è diventato Javascript ora?
  - ➊ – Un **linguaggio server side**, che concorre con PHP
  - ➋ – La tecnologia di base di una metodologia di sviluppo software per la realizzazione di applicazioni web interattive (**AJAX**)
  - ➌ – La tecnologia di base per un formato di interscambio di dati fra applicazioni client-server (**JSON**) che concorre con XML
    - La tecnologia alla base di DB no SQL come Mongo DB





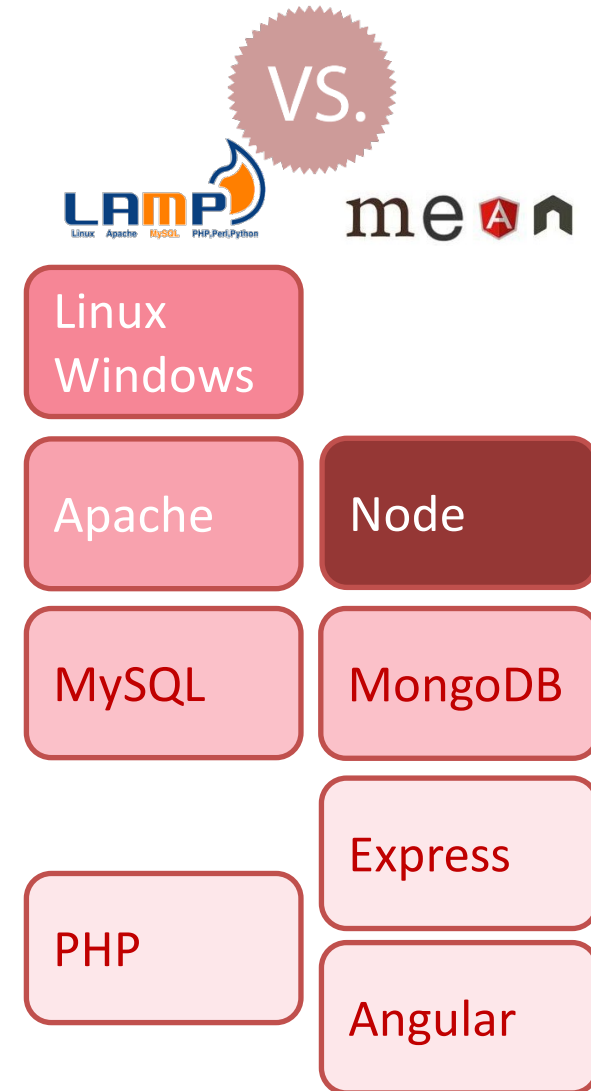
# Una architettura tutta JS: MEAN

- MEAN è un solution stack con una struttura tutta basata su JS. In particolare:
  - **M**ongoDB, come data Base (NoSQL database).
  - **E**xpress.js, come framework di sviluppo JavaScript lato server.
  - **A**ngular, come framework di sviluppo JavaScript lato **client**.
  - **N**ode.js, ambiente di esecuzione per applicazioni **server-side** (permette di eseguire codice Javascript server-side all'esterno del browser).
- MEAN si definisce come «piattaforma JavaScript fullstack per applicazioni web moderne»



# Confronto MEAN-L(W)AMP

- Il confronto tra **MEAN** e **L(W)AMP** evidenzia molte differenze:
  - LAMP e WAMP fanno riferimento a uno specifico sistema operativo. MEAN è più moderno, nasce **multiplatforma**.
  - MEAN usa Node.js come ambiente di esecuzione per applicazioni server-side, che permette di eseguire codice Javascript server-side all'esterno del browser.
  - MEAN usa un DB **non relazionale** (MongoDB) al posto di MySQL, DB relazionale di L(W)AMP.
  - MEAN fornisce **due supporti di programmazione** (uno **client** e uno **server**, Angular e Express) basati sullo stesso linguaggio **Javascript**. L(W)AMP si occupano invece di definire solo lo stack a lato server, usando per programmare PHP/Python e Perl.



- Nella programmazione Web «tradizionale» ogni passo dell'applicazione richiede di
  - consultare il server,
  - eseguire una funzione dell'application logic,
  - generare l'HTML finale,
  - riceverlo e visualizzarlo.
- Questo approccio è lento e poco dinamico perché ad ogni passo dell'applicazione corrisponde una interazione HTTP, ovvero un proprio URL
- Per superare questo modello è stato sviluppato **AJAX (Asynchronous Javascript And Xml)**, un approccio per sviluppare applicazioni Web che aumenta la dinamicità di pagine web, grazie allo scambio di piccole quantità di dati

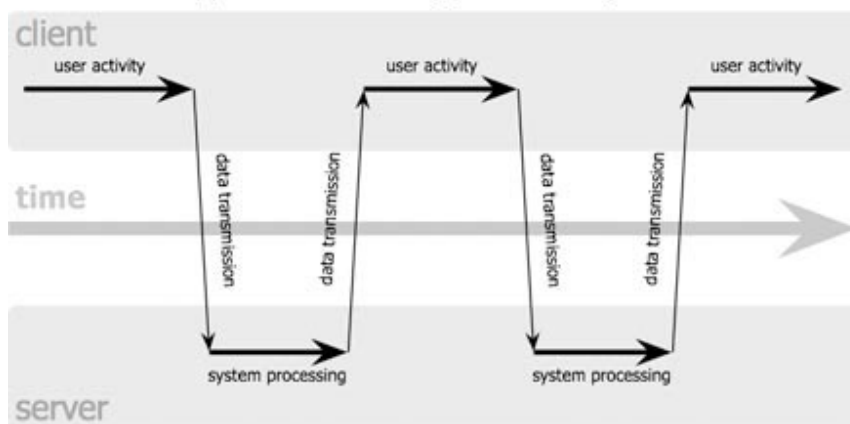
# AJAX

- Anche in questo caso, il nome rischia di essere fuorviante: le applicazioni AJAX **possono utilizzare, oltre a XML, anche testo normale o dati in formato JSON.**
- AJAX:
  - Permette alle pagine web di cambiare il proprio contenuto senza effettuare refresh dell'intera pagina
  - È una tecnologia web indipendente dal software del web server
  - È eseguito all'interno del browser
  - È basato sul protocollo HTTP
  - Trasferisce dati **in modo asincrono** tra il browser e il web server (attraverso **HTTP requests**)
  - Le HTTP requests sono inviate da chiamate a script di JavaScript **senza dover effettuare submit di form**

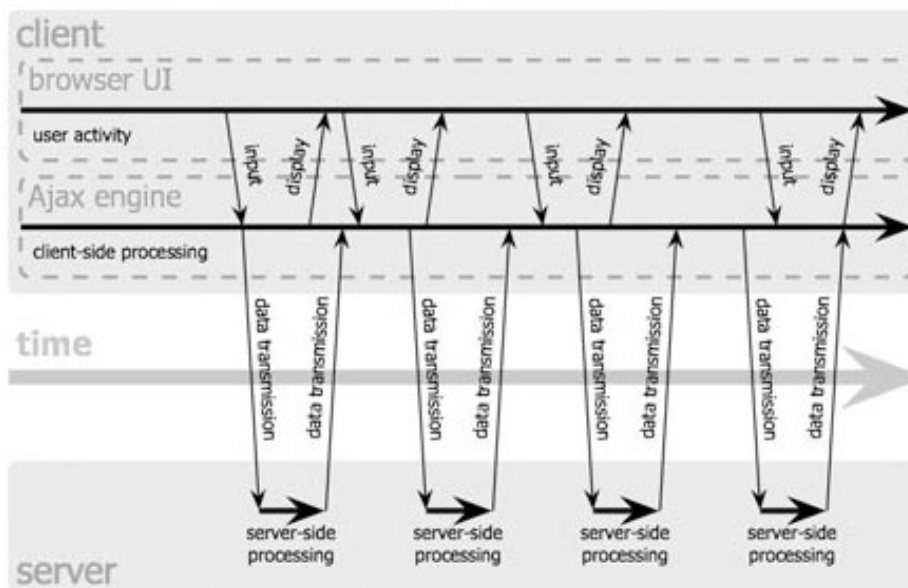
# AJAX

- Le differenze nel modello di interazione c-s

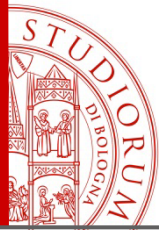
classic web application model (synchronous)



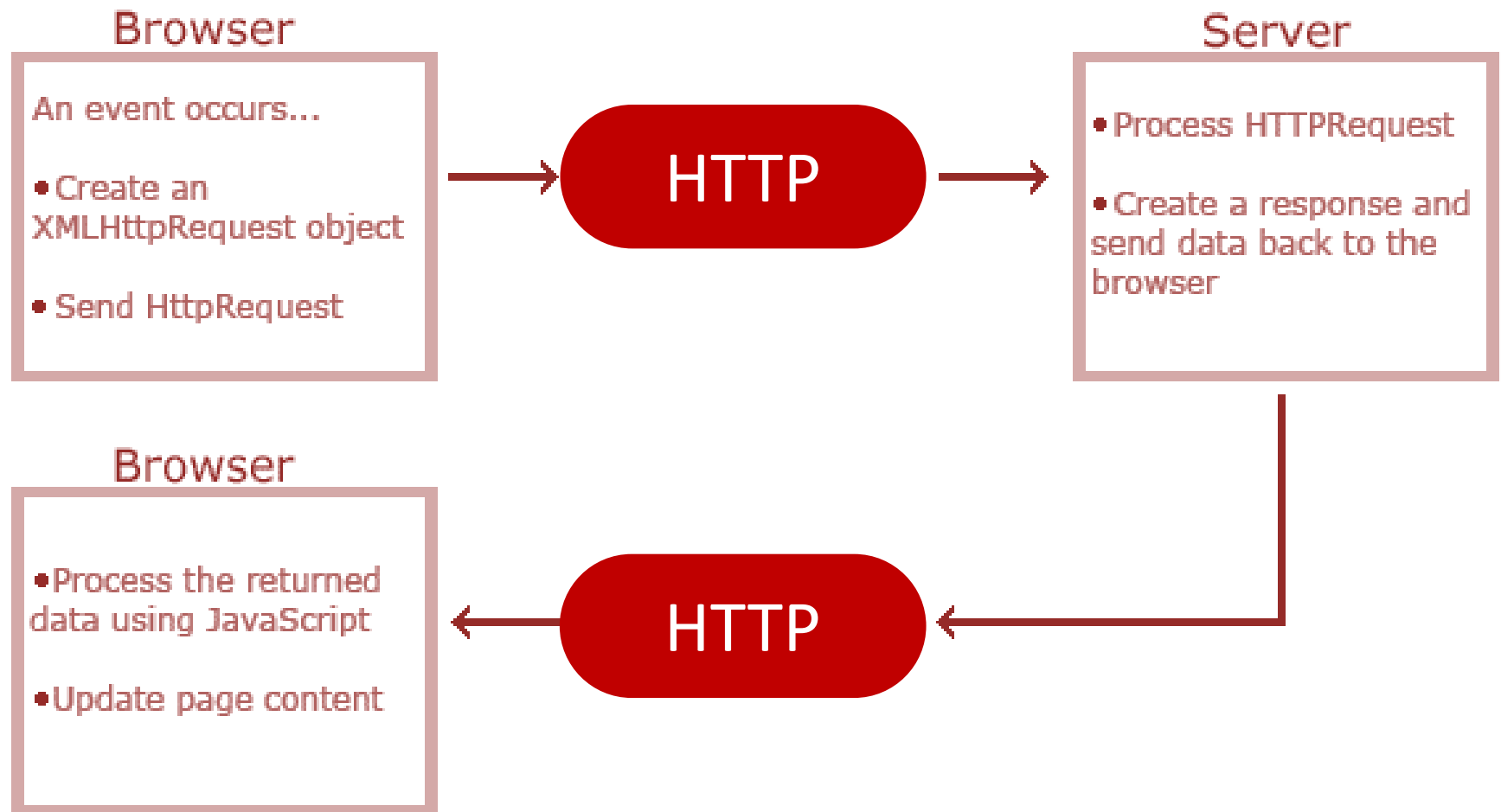
Ajax web application model (asynchronous)







# AJAX



# Tecnologie di Base

- AJAX usa:
  - HTML e CSS per la presentazione
  - Aggiorna il DOM dinamicamente
  - XML con XSLT, oppure HTML preformattato, testo semplice, **oppure JSON** per lo scambio di dati
  - JavaScript, per la programmazione lato client
  - Un oggetto **XMLHttpRequest**, che consente al browser di dialogare in modo asincrono con il server

# AJAX

- **AJAX in JavaScript**, occorre:
  - Definire un oggetto per poter inviare una opportuna HTTP request
  - Ottenere l'oggetto richiesto
  - Definire una opportuna funzione per gestire la response
  - Effettuare una request GET o POST
  - Inviare i dati
  - Gestire la response
- In realtà è più semplice usare AJAX in un **framework JS-AJAX** che preveda già una gestione semplice della comunicazione e della modifica conseguente alla pagina.

# Framework JS-AJAX

- Sono librerie Javascript che semplificano la vita nella creazione di applicazioni Ajax anche complesse.
- Hanno tre scopi fondamentali
  - **Astrazione:** gestiscono le differenze tra un browser e l'altro e forniscono un modello di programmazione unico (o quasi) che funziona MOLTO PROBABILMENTE su tutti o molti browser.
  - **Struttura dell'applicazione:** forniscono un modello di progetto dell'applicazione omogeneo, indicando con esattezza come e dove fornire le caratteristiche individuali dell'applicazione
  - **Libreria di widget:** forniscono una (più o meno) ricca collezione di elementi di interfaccia liberamente assemblabili per creare velocemente interfacce sofisticate e modulari

# I framework JS-AJAX

- Forniscono:
  - Accesso al DOM
    - Navigazione sull'albero del documento
    - Selezione di elementi
    - Modifiche dinamiche al contenuto degli elementi
  - AJAX: Comunicazioni asincrone con il server
    - Gestione asincronicità e callback
    - Gestione successo e errori
    - Conversione da e per JSON
  - Gestione eventi
    - Selezione dell'evento e associazione ad elementi arbitrari
    - Bubbling degli eventi
    - Ricca struttura dati dell'evento
  - Libreria di widget
    - Layout complessivo della pagina
    - Elementi dei form
    - Animazioni
- Modifiche dinamiche allo stile CSS degli elementi
  - Posizione
  - Comparsa / sparizione
  - Caratteristiche tipografiche
- Tabelle intelligenti e alberi
- Templating

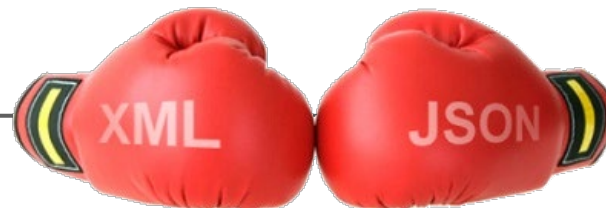
# JSON

- **JSON** (JavaScript Object Notation) è un formato adatto all'interscambio di dati fra applicazioni client-server.
  - Viene usato, a partire da AJAX, come alternativa a XML/XSLT.
  - Nativo in Javascript (è basato sul ECMA-262 3<sup>a</sup> edizione dicembre 1999) esistono però librerie JSON per la maggior parte dei linguaggi di programmazione.
  - Come XML mantiene insieme il dato e l'etichetta descrittiva del dato, permettendo di creare strutture dati auto-descrittive e auto-contenute.
- JSON è progettato per essere minimale, testuale e integrato in JavaScript.

# JSON: esempio

```
{  
  "name": "Pinco",  
  "surname": "Pallino",  
  "address": {  
    "street": "Via San Crispino, 12",  
    "city": "Cesena",  
    "country": "Italy"  
  },  
  "phones": [  
    { "sede": "Cesena", "num": "0547 338892" },  
    { "sede": "Bologna", "num": "051 2094880" }  
  ]  
}
```





- **<XML> vs {JSON}**
  - È più semplice ed è immediatamente integrato con la programmazione in Javascript
  - è più leggero: è meno verboso e quindi usa file più piccoli e ha performance migliori nel trasferimento dati.

```
{
  "impiegati": [
    {
      "nome": "Tony Stark",
      "genere": "M"
    },
    {
      "nome": "Bruce Wayne",
      "genere": "M"
    }
  ]
}
```

```
<impiegati>
  <impiegato>
    <nome>Tony Stark</nome>
    <genere>M</genere>
  </impiegato>
  <impiegato>
    <nome>Bruce Wayne</nome>
    <genere>M</genere>
  </impiegato>
</impiegati>
```

# JSON

- Usare **JSON** in **Javascript** è particolarmente semplice:
  - l'interprete è in grado di eseguirne il parsing da stringa ad oggetto JSON tramite una semplice chiamata alla funzione **parse()** e una funzione **stringify()** per il parsing da oggetto JSON a stringa.
  - È basato sul concetto di array associativo e di oggetto di Javascript. Poiché un array associativo può avere al suo interno un array o anche un altro array associativo, posso creare strutture gerarchiche arbitrariamente complesse.
- Anche nel caso di JSON l'utilizzo attuale va molto oltre il suo obiettivo originale, per esempio **MongoDB** supera la struttura relazionale a favore di documenti in stile **JSON con schema dinamico** (che MongoDB chiama il formato BSON).

# DOM



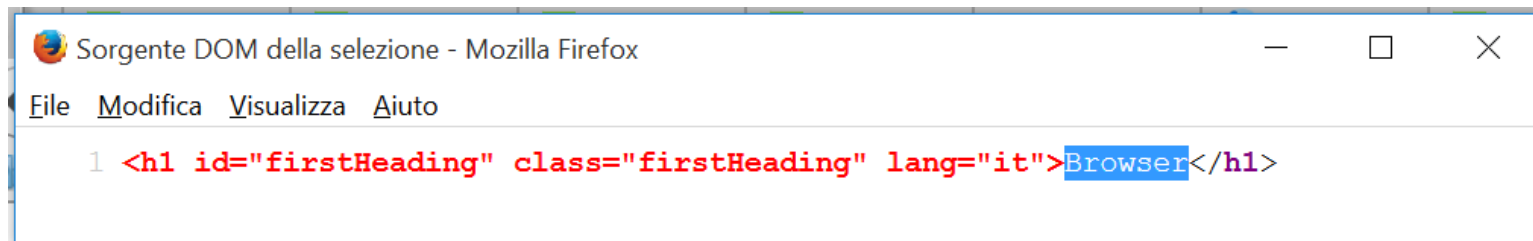
DOM

DOCUMENT OBJECT MODEL

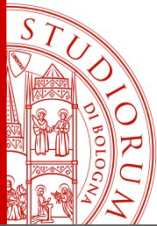
- Abbiamo citato in vari momenti delle lezioni il **DOM**.
- Il **Document Object Model** è rappresentazione dei documenti strutturati come modello orientato agli oggetti.
- Ogni documento caricato dal browser genera un DOM che specifica sottoforma di gerarchia di oggetti, tutti gli elementi di quel documento:
  - Se un elemento ne contiene un altro, allora il secondo è figlio del primo nella gerarchia del DOM
  - Gli attributi dell'elemento sono attributi dell'oggetto corrispondente

# DOM

- Il DOM definisce sostanzialmente un'interfaccia di programmazione (API) per documenti sia HTML sia XML.
  - Definisce la struttura logica dei documenti ed il modo in cui si accede e si manipola un documento.
  - Utilizzando DOM i programmatori possono costruire documenti, navigare attraverso la loro struttura, e aggiungere, modificare o cancellare elementi.
  - Ogni componente di un documento HTML o XML può essere letto, modificato, cancellato o aggiunto utilizzando il Document Object Model.



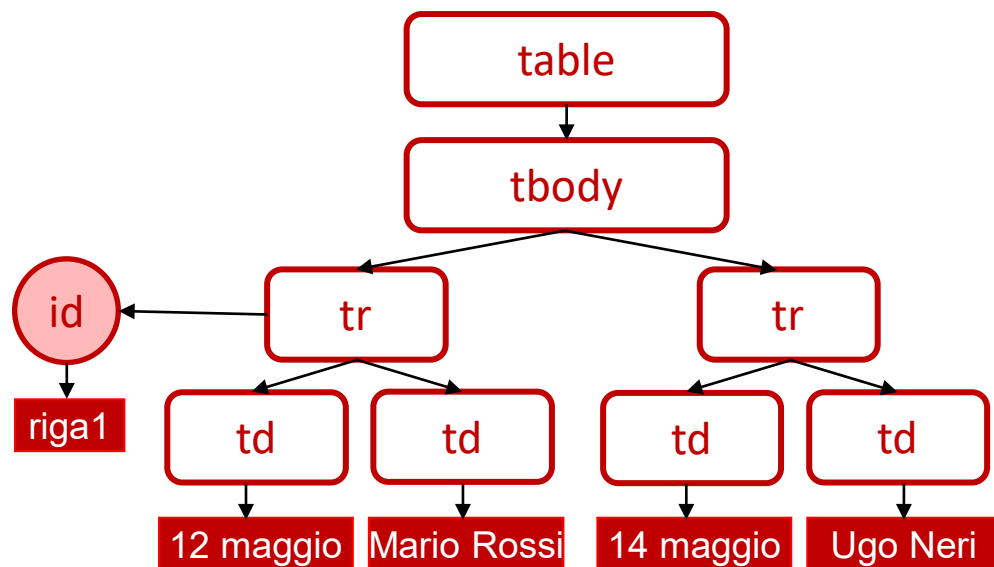
```
1 <h1 id="firstHeading" class="firstHeading" lang="it">Browser</h1>
```



# DOM

- Per ogni documento (e sua porzione) viene creato un DOM:

```
<table>
  <tbody>
    <tr id="riga1">
      <td>12 maggio</td>
      <td>Mario Rossi</td>
    </tr>
    <tr>
      <td>14 maggio</td>
      <td>Ugo Neri</td>
    </tr>
  </tbody>
</table>
```



# DOM

## HTML



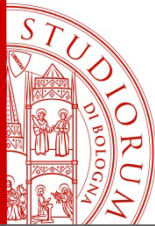
The **HTML DOM** is an **Object Model** for **HTML**. It defines:

- HTML elements as **objects**
- **Properties** for all HTML elements
- **Methods** for all HTML elements
- **Events** for all HTML elements



The **HTML DOM** is an **API** (Programming Interface) for **JavaScript**:

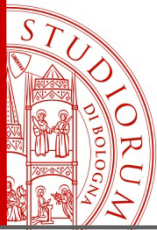
- JavaScript can add/change/remove HTML elements
- JavaScript can add/change/remove HTML attributes
- JavaScript can add/change/remove CSS styles
- JavaScript can react to HTML events
- JavaScript can add/change/remove HTML events



# Oggetti del DOM

- L'oggetto principale di DOM è **DOMNode**, che però è una interfaccia (cioè viene solo usata per crearne classi)
- Il core del DOM definisce alcune **classi fondamentali** per i documenti HTML e XML, e ne specifica proprietà e metodi.
- Gli oggetti principali definiti nel DOM sono:
  - **DOMDocument**: il documento di cui si sta parlando
  - **DOMElement**: ogni singolo elemento del documento
  - **DOMAttr**: ogni singolo attributo del documento
  - **DOMText**: ogni singolo nodo di testo del documento
  - **DOMComment**, **DOMProcessingInstruction**, **DOMCDATASection**, **DOMDocumentType**, ecc.





# DOMNode

- **DOMNode** specifica i metodi per accedere a tutti gli elementi di un nodo di un documento, inclusi il nodo radice, il nodo documento, i nodi elemento, i nodi attributo, i nodi testo, ecc.
- Esempi:

## **membri**

- nodeName
- nodeValue
- nodeType
- parentNode
- childNodes
- attributes

## **metodi**

- insertBefore()
- replaceChild()
- removeChild()
- appendChild()
- hasChildNodes()
- hasAttributes()



# DOMDocument

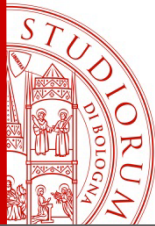
- **DOMDocument** specifica i metodi per accedere al documento principale, tutto compreso
- Esempi:

## **membri**

- docType
- documentElement

## **metodi**

- createElement()
- createAttribute()
- createTextNode()
- getElementsByTagName()
- getElementById()



# DOMElement

- **DOMElement** specifica i metodi e i membri per accedere a qualunque elemento del documento.
- Esempi:

<b>membri</b>	<b>metodi</b>
– tagName	getAttribute() setAttribute() removeAttribute() getElementsByTagName() getElementById()
- ... e analogamente per le altre classi ed interfaccia del DOM.

# Prima domanda

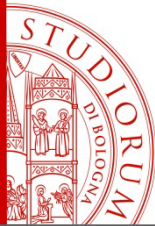
**BONUS**

- **DOMANDA 1:**

Quali sono gli elementi discendenti (**TUTTI**) di tbody?

```
<table>
  <tbody>
    <tr id="riga1">
      <td>12 maggio</td>
      <td>Mario Rossi</td>
    </tr>
    <tr>
      <td>14 maggio</td>
      <td>Ugo Neri</td>
    </tr>
  </tbody>
</table>
```

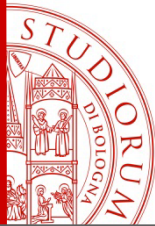
- ☐ Tutti gli elementi <tr> e tutti gli elementi <td>
- ☐ Nessuno
- ☐ Tutti gli elementi <td>
- ☐ Tutti gli elementi <tr>



# Come proseguiamo

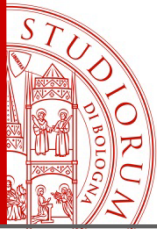
---

- Introdotti tutti i concetti generali, passiamo alla parte operativa:
  - introduciamo **Javascript** come linguaggio **CLIENT SIDE** sapendo che è C-based, ovvero per differenza con Java e il C
  - Seguirà laboratorio dedicato a **Javascript**



# Javascript client side

- Noi vediamo **JS client-side**, il cui codice è:
  - Interpretato dal browser
  - Incluso nella pagina HTML
- Il codice JavaScript può comparire **in più parti del documento HTML** (con uno qualunque dei tre metodi possibili che ora vedremo)
- Tutti i frammenti di codice presenti nello stesso documento HTML fanno parte di **un unico programma**
  - una variabile o una funzione definiti all'interno di un tag `<script>` possono essere usate da tutti i frammenti di codice eseguiti successivamente



# Inserire JS nella pagina

- Per inserire codice JavaScript in un documento HTML esistono tre modi:

- 1 elemento **<script>** contenente il codice JavaScript

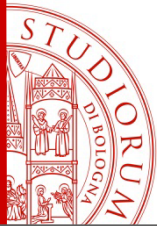
```
<script type="text/javascript">  
    ...codice JS  
</script>
```

- 2 Riferimento a **script contenuti in file .js esterni**

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

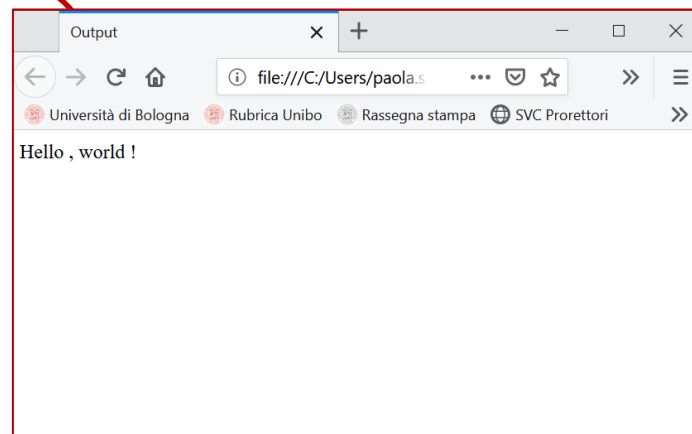
- 3 Direttamente nel codice HTML in risposta ad eventi

```
<input type="button" onclick="alert('Ciao!')".../>  
<a href="javascript:nome_funzione()"> Clicca  
qui!</a>
```

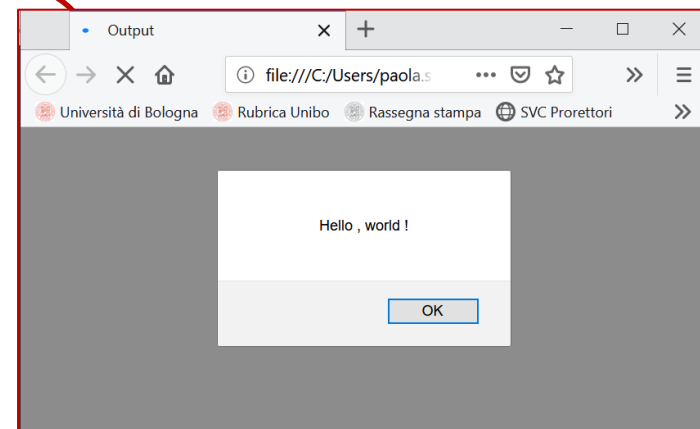


# Hello world!

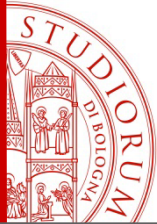
```
<html>
<head>
  <title>Output</title>
</head>
<body>
  <script>
    document.writeln ("Hello , world !")
  </script>
</body>
</html>
```



```
<html>
<head>
  <title>Output</title>
</head>
<body>
  <script>
    alert ("Hello , world !");
  </script>
</body>
</html>
```







# Sintassi

- La sintassi di JavaScript è **modellata su quella del C** con alcune varianti significative
  - In particolare: È un linguaggio **case-sensitive**
  - Le istruzioni sono **terminate** da ';' ma il terminatore può essere omesso se si va a capo
  - Sono ammessi sia commenti multilinea (delimitati da /\* e \*/) che mono-linea (iniziano con //)
  - Gli **identificatori** possono contenere lettere, cifre e i caratteri '\_' e '\$' ma non possono iniziare con una cifra



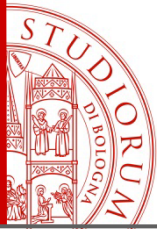
# Controllo

- Costrutti di controllo (uguali a C, C++ e Java):

- `if (test) { ... } else { ... }`
- `switch (v) {`
  - `case c1: ... ; break;`
  - `case c2: ... ; break;`
  - `default: ...;``}`
- `while (test) { ... }`
- `for (i=0; i<max; i++) { ... }`

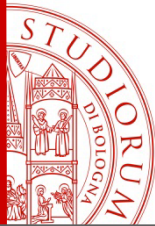
- Funzioni:

- `function funzione1(par1, par2, ...) {`
  - `...`
  - `return v;``}`



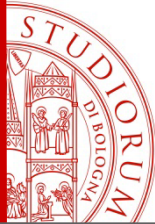
# Operatori relazionali

- Gli **operatori relazionali** sono i soliti (`==`, `!=`, `>`, `<`, `>=`, `<=`) più due nuovi (`===`, `!==`), combinabili con gli usuali operatori logici AND (`&&`), OR (`||`), NOT (`!`)
  - **Nella valutazione di condizioni si considera falso non solo false, ma ogni valore falsy ovvero anche null, undefined, la stringa vuota (''), il valore 0 e NaN (Not a Number)**
  - Ogni altro oggetto, inclusa la stringa 'false', è vero.
- Se i tipi dei due operandi sono diversi, `==` e `!=` applicano type coercion secondo regole molto discutibili:
  - `0 == '' true`, perché sono entrambi falsy values
  - `0 == '0' true`, perché '0' è coercibile a 0
- I due operatori `===` e `!==` offrono una alternativa più stabile al comportamento discutibile dei due `==`, `!=` perché non applicano mai alcuna type coercion, risultando quindi più affidabili e predicibili



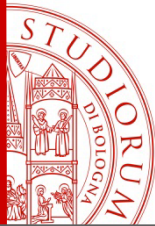
# Variabili

- Le variabili vengono dichiarate usando la parola chiave **var**:  
***var nomevariabile;***
  - **Non hanno un tipo** possono contenere valori di qualunque tipo
  - È prevista la possibilità di inizializzare una variabile contestualmente alla dichiarazione  
***var f = 15.8***
  - Possono essere dichiarate in linea:  
***for (var i = 1; i<10; i++)***
- Esiste lo scope *globale* e quello *locale* (ovvero dentro una funzione) ma, a differenza di Java, *non* esiste lo scope di blocco.



# Dichiarazione implicita o esplicita

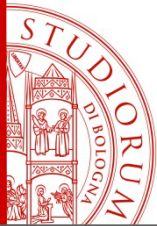
- La dichiarazione di una variabile può essere:
  - **implicita** (la si usa e basta)  
`pluto = 18 // dichiarazione implicita`
  - **esplicita** (con la parola chiave var)  
`var pippo = 19 // dichiarazione esplicita`
- **La dichiarazione implicita introduce sempre e solo variabili globali**
- La dichiarazione esplicita può introdurre variabili globali o locali, a seconda di dove compare.



# Javascript Hoisting

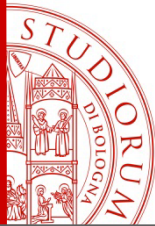
---

- **Hoisting** (sollevamento) indica il sistema con cui Javascript muove le dichiarazioni delle variabili in testa al codice (le solleva)
- Grazie all'hoisting una variabile Javascript:
  - **può essere usata prima di essere dichiarata**
  - Può essere dichiarata dopo essere stata usata



# Javascript Hoisting

- *«La dichiarazione implicita introduce sempre e solo variabili globali»*
- ```
function miaFunzione() {  
    var a = 20;  
    b = 100;  
}
```
- ```
miaFunzione();
```
- ```
console.log(a);
```
- ```
console.log(b);
```

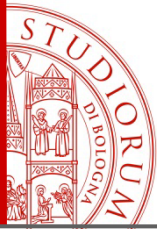


# Javascript Hoisting

- «La dichiarazione implicita introduce sempre e solo variabili globali»
- ```
function miaFunzione() {  
    var a = 20;  
    b = 100;  
}
```

*Diagram: A red arrow points from `var a = 20;` to the text "Locale a miaFunzione". Another red arrow points from `b = 100;` to the text "Globale".*
- ```
miaFunzione();
```
- ```
console.log(a); // ReferenceError: a  
is not defined
```
- ```
console.log(b); // 100
```






# Javascript Hoisting

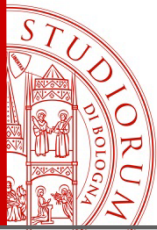
---

- «non esiste lo scope di blocco»
- `for (var i = 1; i<10; i++) { ... }`
- `console.log(i);`



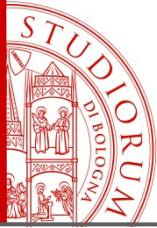
# Javascript Hoisting

- «non esiste lo scope di blocco»
  - `for (var i = 1; i<10; i++) { ... }`
  - `console.log(i); // 10`
-  **Globale**



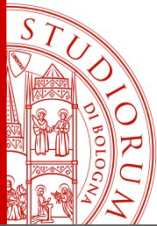
# ES6 – Dichiarazione Variabili

- In ES6, sono state inserite due keyword per la dichiarazione di variabili: **let** e **const**.
- **let** consente di definire variabili con scope di blocco.
- `for (let i = 1; i<10; i++) { ... }`
- `console.log(i); // ReferenceError: i is not defined`
- **note:** Le variabili dichiarate con **let** NON sono accessibili dall'oggetto Window (al contrario di quelle definite con **var**). Inoltre, usando **let**, non posso ri-dichiarare una variabile con lo stesso nome (posso farlo con **var**)



# ES6 – Dichiarazione Variabili

- **const** consente di definire variabili che non possono essere riassegnate.
- `const Prova = 10;`
- `Prova = 11 //TypeError: Assignment to constant variable`
- È obbligatorio assegnare un valore in fase di dichiarazione della variabile:
- `const Prova2 //SyntaxError: Missing initializer in const declaration`



# ES6 – Dichiarazione Variabili

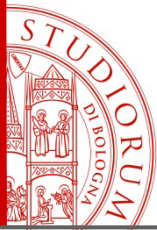
---

**Negli elaborati, negli esercizi e nel compito**

**si usano**

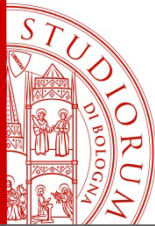
**SEMPRE**

**let e const**



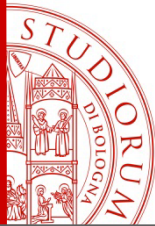
# ES6 – Dichiarazione Variabili

- Con **dati non primitivi**, come array e oggetti, sebbene non sia possibile riassegnare un valore diverso, è comunque possibile modificare il valore di un elemento dell'array o di una proprietà, oppure aggiungerne di nuovi.
- ```
const auto = [ 'Fiat', 'Lancia',  
  'Maserati' ] ;  
auto[0] = 'Lamborghini' ;  
auto.push( 'Ferrari' ) ;
```



# ES5 – Strict Mode

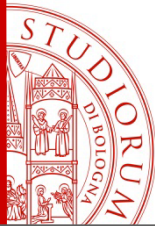
- Lo **strict mode** è una direttiva ES5 che impone regole stringenti in fase di dichiarazione di una variabile. Non è possibile assegnare un valore ad una variabile se questa non è stata dichiarata.
- È sufficiente aggiungere '**use strict**'; all'inizio dello script.
- **Vantaggio:** se sbaglio a scrivere il nome di una variabile normalmente viene creata una nuova variabile globale. In strict mode, invece, mi sarà **segnalato un errore**.



# Tipi

- Javascript prevede pochi tipi primitivi: numeri, booleani e stringhe (forse!)
- Numeri (**number**): Sono rappresentati in formato floating point a 8 byte
  - Non c'è distinzione fra interi e reali
  - Esiste il valore speciale **NaN** (not a number) per le operazioni non ammesse (ad esempio, radice di un numero negativo)
  - Esiste il valore **infinite** (ad esempio, per la divisione per zero)
- Booleani (**boolean**): ammettono i valori **true** e **false**





# Array

- Un **array** Javascript è un'entità a metà strada fra un array "classico" e una lista
  - come in Java, gli elementi si numerano da 0, `length` dà la lunghezza dell'array, si usa la notazione parentesi quadre
  - a differenza di Java,
    - `length` dà la lunghezza dinamica (attuale) dell'array
    - non c'è il vincolo di omogeneità in tipo: le celle contengono oggetti - cioè qualunque cosa
- È costruito sulla base del costruttore `Array`
  - si parte con un dato contenuto iniziale (anche vuoto):  
`colori = new Array("rosso", "verde", "blu")`
  - o anche, per array di costanti, con la notazione `[..]`:  
`varie = ["ciao", 13, Math.sin]`
  - si aggiungono poi nuovi elementi dinamicamente  
`colori[3] = "giallo"`



# Stringhe

- **Stringhe** in JavaScript:
  - Il tipo **string** denota stringhe di caratteri **Unicode**
  - non esiste il tipo **char**: **un carattere è una stringa lunga 1**
- Ogni stringa è un oggetto IMMUTABILE dotato di proprietà, tra cui **length**, e di metodi, tra cui **substring** (indici **first**, **last** come in Java – **last** indica il primo *escluso*)
- Le *costanti stringa* possono essere delimitate sia da *virgolette* sia da *apici singoli*, se occorre annidare virgolette e apici, occorre *alternarli*:

```
document.write('')  
document.write("<img src='image.gif'>")
```
- Le stringhe si possono **concatenare** con l'operatore +

# Seconda domanda

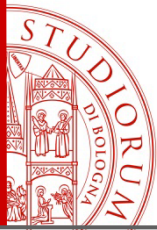
**BONUS**

- **DOMANDA 2:**

In JavaScript cosa stampa a console il seguente codice?

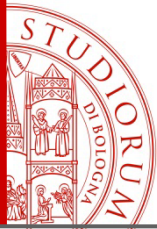
```
var stringa1 = "21";  
var stringa2 = "2";  
console.log(stringa1 * stringa2);
```

- ☐ String("42")
- ☐ Number(42)
- ☐ NaN
- ☐ Undefined



# JavaScript: Classi e oggetti

- JavaScript è un linguaggio object-oriented anche se non è tipato come Java.
  - Non esiste il concetto esplicito di Classe, ma si usa **function()** anche per definire un costruttore.
  - **new** istanzia un nuovo oggetto.
  - I membri di un oggetto possono essere acceduti equivalentemente con **obj.field** o con **obj["field"]**
- Gli oggetti possono essere considerati come liste non ordinate di proprietà:
  - Ogni proprietà può essere o un dato primitivo (una stringa, un numero, ecc.), oppure un dato reference (un altro oggetto, un array, o una funzione).



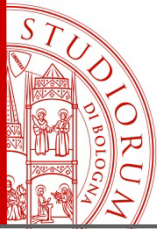
# Gli oggetti in Javascript

```
giuseppe = {  
  nome: 'giuseppe',  
  altezza: 180,  
  nascita: new Date(1995, 3, 12),  
  salta: function() {  
    return 'hop!'  
  }  
}
```

- Per chiamare le proprietà dell'oggetto:

```
> giuseppe.nome      ---- giuseppe['nome']
```

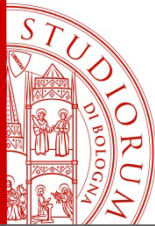
```
> giuseppe.salta()   ---- (giuseppe.salta)()  
                        giuseppe['salta']()
```



# Oggetti - Costruttore

---

```
function Persona(nome, altezza, nascita) {  
    this.nome = nome;  
    this.altezza = altezza;  
    this.nascita = nascita;  
    this.salta = function() {return "hop!";};  
}  
  
giuseppe = new Persona('giuseppe', 180, new  
Date(1995,3,12));
```



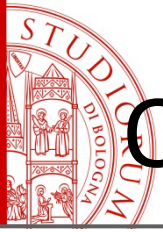
# ES6 Oggetti - Class

- In ES6 è stata introdotta la keyword class per la definizione delle classi.

```
class Persona{
  constructor(nome, altezza, nascita)
  {
    this.nome = nome;
    this.altezza = altezza;
    this.nascita = nascita;
  }
  salta() {
    return 'hop!';
  }
}

giuseppe = new Persona('giuseppe', 180, new
Date(1995,3,12));
```

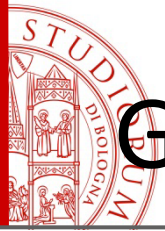
- NB: Si tratta semplicemente di uno zucchero sintattico! Il risultato è lo stesso del precedente! Non è stato introdotto il concetto di classe!**



# Oggetti principali: Window e Navigator

- **window**: è l'oggetto top-level con le proprietà e i metodi della finestra principale:
  - **posizione**: `moveBy (x, y)`, `moveTo (x, y)`, ecc.
  - **dimensioni**: `resizeBy (x, y)`, `resizeTo (x, y)`, ecc.
  - **altre finestre**:  
`open ("URLname", "Windowname", ["opt"])`
  - **tempo e intervalli**  
`setTimeout (function () , millisecs , ["opt"])`
- **navigator**: è l'oggetto con le proprietà del client come nome, numero di versione, plug-in installati, supporto per i cookie, etc.





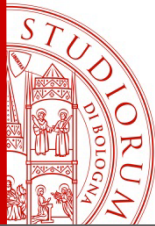
# Gli oggetti principali: Location e History

- **location**: l'URL del documento corrente. Modificando questa proprietà il client accede a un nuovo URL (redirect):
  - `window.location = "http://www.unibo.it/";`
  - `window.location.href = "http://www.unibo.it/";`
- **history**: l'array degli URL acceduti durante la navigazione. Possibile creare applicazioni client-side dinamiche che 'navigano la cronologia':
  - Proprietà: `length`, `current`, `next`
  - Metodi: `back()`, `forward()`, `go(int)`



# L'oggetto Document

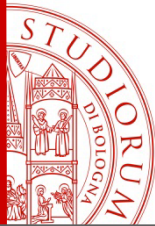
- **document** rappresenta il DOM ovvero il contenuto e la struttura del documento, ed ha proprietà e metodi per accedere ad ogni elemento nella gerarchia:
  - **window.document.title**: titolo del documento
  - **window.document.forms[0]**: il primo form
  - **window.document.forms[0].checkbox[0]**: la prima checkbox del primo form
  - **window.document.forms[0].check1**: l'oggetto con nome "check1" nel primo form (non per forza una checkbox!)
  - **window.document.myform**: l'oggetto "myform"
  - **window.document.images[0]**: la prima immagine



# Modello di documento

- Ogni oggetto nella gerarchia è caratterizzato da un insieme di proprietà, metodi ed eventi che permettono di accedervi, controllarlo, modificarlo.

```
function Verify() {  
    if (document.forms[0].elements[0].value == "") {  
        alert("Il nome è obbligatorio!")  
        document.forms[0].elements[0].focus();  
        return false;  
    }  
    return true;  
}  
  
<form action= "... " onSubmit="return Verify()">  
<label>Name: <input type="text" name="nome" ...> </label>
```



# Javascript e DOM

---

- Javascript implementa i metodi standard per accedere al DOM del documento, ma il supporto sui vari browser non è uniforme

```
var c = document.getElementById('c35');
```

```
c.setAttribute('class', 'prova1');
```

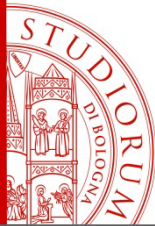
```
c.removeAttribute('align');
```

```
var newP = document.createElement('p');
```

```
var text = document.createTextNode('Ciao Mamma.');
```

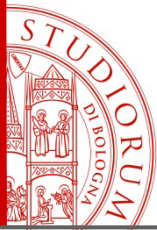
```
newP.appendChild(text);
```

```
c.appendChild(newP);
```



# Eccezioni

- JavaScript fornisce meccanismi nativi per sollevare e catturare **eccezioni**:
  - **try**: il blocco di comandi da eseguire e su cui controllare se si verifica un'eccezione
  - **catch**: il blocco di istruzioni da eseguire se si cattura una data eccezione
  - **throw**: istruzione per sollevare un'eccezione
- Alternativamente (e per compatibilità con i browser molto datati, es. Netscape Navigator 3.0) si può usare l'handler di eventi **onError()**



# Esempio di uso delle eccezioni

```
var x=prompt("Enter a number between 0 and 9:","");

try {
    if (x=="5") throw "line"
    var el = document.getElementById("menu"+x)
    var address = el.attributes["href"].value
    return address ;
} catch(er) {
    if(er=="line")
        return "Errore Linea";
    else
        return "Errore valore non adeguato" ;
}
```

# Domande?

