# JavaScript

..ed altre tecnologie clientside

# Per ora

Ci siamo prevalentemente concentrati sul back-end delle web apps.

Abbiamo visto con Django come implementare logica arbitraria lato server, scatenata tramite input degli utenti.

Applicazione tipo finora: l'utente raggiunge un url, interagisce con esso tramite gli *input* del file HTML generato dinamicamente lato server. Tale input viene rispedito lato server, il quale risponde con un'altra pagina web generata dinamicamente. Questo, poi accade iterativamente.

Abbiamo visto poco sul versante presentazione…

# Ma le web app/siti che conosciamo?

Si limitano a questo?

**No.**

Computazione arbitraria (e di conseguenza dinamicità) può esserci anche lato client.

Paradossalmente, se non avessi modo di avere dinamicità **anche** lato client, ogni singola virgola di cambiamento lato client deve essere scatenata da una coppia (request, response) del server con conseguente ricaricamento della pagina!

Come si ottiene dinamicità lato client? Usando linguaggi e strumenti che ci permettono di definire comportamenti arbitrari in esecuzione sul **client.**

# Javascript

Linguaggio di scripting lato client, nato per il web nel 1995

- Il documento HTML appare “statico” lato client.
- Il codice Javascript è **embeddato** nel documento HTML nei tag <script></script>.
- Perchè usarlo?

– manipolazione degli oggetti HTML;

– validazione (lato client) campi dei form;

– effetti grafici difficilmente implementabili con solo CSS

– … tanto altro…

https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages/

# The Most In-Demand Programming Languages for 2022

### 1. JavaScript

*What this language is used for:*

- Web development
- Game development
- Mobile apps
- Building web servers

According to Stack Overflow's 2020 Developer Survey, JavaScript currently stands as the most commonly-used language in the world (69.7%), followed by HTML/CSS (62.4%), SQL (56.9%), Python (41.6%) and Java (38.4%). It is also the most sought-out programming language by hiring managers in the Americas (PDF, 2.4 MB).

# Comparazione

| | Python | js |
|---|---|---|
| Indentazione | Strongly enforced | Uso di {} |
| Definizione Variabili | Implicita | Implicita/Esplicita |
| Notazione | snake_case | camelCase; |
| Variabili numeriche | int/float/complex | BigInt/Number/number |
| Strutture dati | Molte built-in (array, set, dict, tuple etc… | Molte meno… |
| Tipizzazione | Dinamica/Strong | Dinamica/Weak |
| Paradigma | Oggetti e funzionale | Oggetti, eventi e funzionale |

# Dichiarazioni variabili

```
var a = 5;
const COSTANTE = 5;
COSTANTE=4; //TypeError: Assignment to constant variable.
{
    let a = 6;
    console.log("a nel blocco vale " + a); //6
}
 console.log("a fuori blocco vale " + a); //5
```

# Dichiarazione implicita (omissione di var)

```
function funz(){  a = true; }


funz();

console.log("a esiste? " + a) //stampa true
```

**Assolutamente non consigliato.**
“a” esiste se e solo se la funzione “funz” è chiamata prima del suo utilizzo.
Essendo “a” dichiarato senza la keyword var, **essa viene posta nel global scope**, e potenzialmente può sovrascrivere altre variabili chiamate allo stesso modo.

# Tipi di dato fondamentali in js

- Number
- Boolean
- Null
- String
- Date
- Array
- BigInt

```
var v = 5;

console.log(“v è di tipo “ + typeof(v) + “ e vale “ + v);
```

**v è di tipo number e vale 5**

# Primitivi e Oggetti

```
var v = 5;

console.log("v di tipo " + typeof(v) + " e vale " + v);

v = new Number(5);

console.log("v è di tipo " + typeof(v) + " e vale " + v);
```

**v è di tipo number e vale 5**

**v è di tipo object e vale 5**

# Array in js

Oggetto di tipo “Array”.

```
var v = new Array(); //Array vuoto

var w = new Array(1, 2, 3);

var u = new Array(1, 2, “tre”);
```

• Non è necessario, ma è possibile specificarne la dimensione

```
u.push(‘quattro’); //aggiungo un elemento

console.log(u); //stampa nella console degli sviluppatori

[ 1, 2, 'tre', 'quattro' ]
```

# Array: notazione alternativa

Similmente a python:

```
//non possibile specificare la dimensione

var w = [1,2,"tre"];

w.push('quattro');

console.log(w);

/*List comprehension: non sempre supportata
(experimental/prototype feature)*/
```

# Operatori

“Ereditati” da Java/C++.

Quindi esiste “++”/”--”, diversamente da python.

Non esiste “//” diversamente da python.

Operatori logici:

tornano i vari &&,|| etc…

# Stringhe

Come in java: quindi *indexOf, charAt* etc…

```
var string = "Stringa";


for (let i = 0; i<string.length; i++){
    console.log(string.charAt(i));
}
```

# Set in js

```
var insieme = new Set(["a","b","c","a"]);

insieme.add("d");

console.log(insieme);

Set { 'a', 'b', 'c', 'd' }
```

Diversamente da python, i Set in js presentano un’interfaccia molto basica; e.g. niente metodi specifici per intersezioni/unioni/etc…

# Dizionari

```
var fakeDict = { "IT":"Italia","ES":"Spagna", "FR":"Francia"
};
fakeDict["PIGS"] =
["Portogallo","Italia","Grecia","Spagna"];
console.log(fakeDict);
```

**{ IT: 'Italia',**

**ES: 'Spagna', FR: 'Francia', PIGS: [ 'Portogallo', 'Italia', 'Grecia', 'Spagna' ] }**

# Perchè “fake”?

**Non** esistono dizionari in js.

Quello che abbiamo appena visto si comporta come un dizionario in python, **ma in realtà è un oggetto**.

Del resto, in python potevamo fare *oggetto.__dict__* per ottenere una lista di metodi e attributi di un’istanza di una **classe**.

**Quindi possiamo definire oggetti in js, senza necessariamente definire la sua astrazione in una classe!**

# Oggetti/Classi in js

```
class Rectangle {
  constructor(h, w) {
    this.h = h;
    this.w = w;
  }
//metodi
  area(){return this.h * this.w;}
  toString(){return "Rettangolo";}
}
var r = new Rectangle(3,2);
console.log("Area del " + r + " = " + r.area());
```

# Evitando la definizione di classe…

```
var r0 = {
    w : 2,
    h : 3,
    area : function() { return this.w*this.h; },
    toString : function() { return "Rettangolo"; }
}
console.log("Area del " + r0 + " = " + r0.area());
console.log(“W ed H: ” + r0[“w”] + “, “ + r0.h);
```

# Metodo alternativo: tramite funzione

```
function Rectangle(h,w) {
    this.h = h;
    this.w = w;
     this.area = function() {return this.h*this.w;};
     this.toString = function() {return "Rettangolo";};
}

var r1 = new Rectangle(3,2);
console.log("Area del " + r1 + " = " + r1.area());
```

# Altro metodo alternativo: Object

```
var r = new Object();
r.h = 3;
r.w = 2;
r.area = function(){return this.h*this.w};
r.toString = function(){return "Rettangolo"};

console.log("Area del " + r+ " = " + r.area());
```

# Funzioni

In JavaScript è possibile definire funzioni all'interno di uno script

```
function name(arg0, arg1, …, argn-1) {
…
}
```

La funzione definita è identificata da **name** e dipende dagli argomenti arg0, arg1, …, argn-1

La funzione è un metodo se la keyword **function** è presente all'interno di un **oggetto. Come in python, le funzioni sono comunque entità che è possibile assegnare come fossero variabili.**

# Argomenti in ingresso

```
function stampaArgs(a, b){
    console.log(a+","+b);
}


stampaArgs(1,2); //1,2
stampaArgs(1); //1,undefined
stampaArgs(); //undefined,undefined
stampaArgs(b=1); //1,undefined
stampaArgs(null,null); //null,null
```

# undefined vs null

In JavaScript, **undefined** è un tipo; mentre **null** è un oggetto. Significa che una variabile è stata dichiarata, ma nessun valore le è stato assegnato. Diversamente, **null** in JavaScript **è un valore assegnato ad una variabile**. Tale valore può essere assegnato a qualsiasi oggetto.

```
var a;

if (a==undefined)
    console.log("a non è definita: " + a);

a = null;
console.log(a);
/*
stampa: a non è definita undefined
null
*/
```

# Istruzioni condizionali e di iterazione

Nessuna grossa differenza rispetto a Java, operatori ternari compresi.

inoltre rispetto a python tornano gli switch-case.

```
condizione ? funzioneA() : funzioneB();
```

Stesse considerazioni per quello che riguarda le istruzioni iterative

# js ed HTML

- La caratteristica principale di JavaScript è di essere “integrabile” all’interno delle pagine web

- In particolare consente di aggiungere una logica arbitraria agli elementi HTML, cambiandone le loro proprietà in maniera dinamica

- A differenza degli altri strumenti visti finora, JavaScript funziona completamente sul client

- JavaScript e la sua toolchain è installato di default nei browser più diffusi

# Come avviene l’integrazione?

- Codice js “embeddato” all’interno del mio documento HTML.
- In qualsiasi punto del mio codice esiste un oggetto “*document*” che simboleggia la struttura ad albero dei miei oggetti HTML.
- Da *document* ottengo un riferimento ai widget HTML così da essere in grado di cambiarne proprietà di funzionamento e aspetto.
- Sempre seguendo questo principio, il codice js riesce a catturare gli eventi inerenti l’input dell’utente sui widget dell HTML.

# L’oggetto *window*

Presente nel global scope del nostro script

- Modificare il titolo della tab corrente

```
window.title = “Titolo della pagina”;
```

- Accedere ad un nuovo documento

```
window.location = “https://www.unimore.it/”;
var w = window.open("https://www.unimore.it/", "unimore");
w.moveTo(0, 0); //nuova finestra
```

- Calcolare l’area in pixel della finestra

```
var area = window.innerWidth * window.innerHeight;
```

# Alert dialogs e navigator

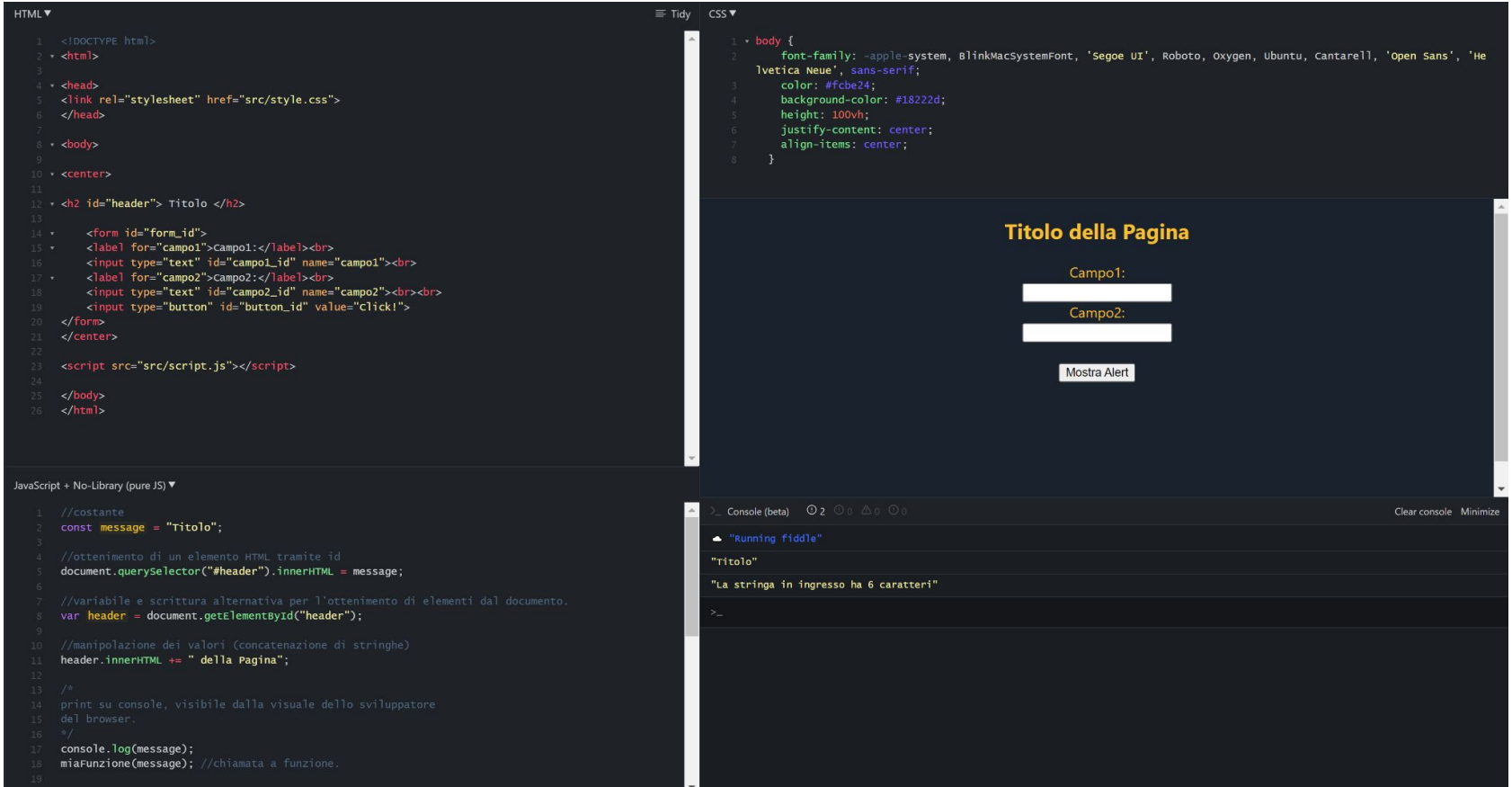
```
window.alert(“Messaggio da ” + window.title);
```

Tipicamente window è implicito. Questo significa che avremmo potuto chiamare quei metodi senza specificare window:

```
alert(“Sei connesso con ” + navigator.appName + “ versione ”
+ navigator.version);
```

# Esempi completi

Si consiglia di provare su jsfiddle.net.



HTML

```
<!DOCTYPE html>
<html>

<head>
<link rel="stylesheet" href="src/style.css">
</head>

<body>

<center>

<h2 id="header"> Titolo </h2>

    <form id="form_id">
    <label for="campo1">Campo1:</label><br>
    <input type="text" id="campo1_id" name="campo1"><br>
    <label for="campo2">Campo2:</label><br>
    <input type="text" id="campo2_id" name="campo2"><br><br>
    <input type="button" id="button_id" value="Click!">
</form>
</center>

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

</body>
</html>
```

CSS

```
body {
    font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, Oxygen, Ubuntu, Cantarell, 'Open Sans', 'Helvetica Neue', sans-serif;
    color: #fcbe24;
    background-color: #18222d;
    height: 100vh;
    justify-content: center;
    align-items: center;
  }
```

Titolo della Pagina

Campo1:

Campo2:

Mostra Alert

JavaScript + No-Library (pure JS)

```
//costante
const message = "Titolo";

//ottenimento di un elemento HTML tramite id
document.querySelector("#header").innerHTML = message;

//variabile e scrittura alternativa per l'ottenimento di elementi dal documento.
var header = document.getElementById("header");

//manipolazione dei valori (concatenazione di stringhe)
header.innerHTML += " della Pagina";

/*
print su console, visibile dalla visuale dello sviluppatore
del browser.
*/
console.log(message);
miaFunzione(message); //chiamata a funzione.
```

Console (beta)

```
"Running fiddle"
"Titolo"
"La stringa in ingresso ha 6 caratteri"
>_
```

# La pagina HTML:

```
<h2 id="header"> Titolo </h2>

    <form id="form_id">
    <label for="campo1">Campo1:</label><br>
    <input type="text" id="campo1_id" name="campo1"><br>
    <label for="campo2">Campo2:</label><br>
    <input type="text" id="campo2_id" name="campo2"><br><br>
    <input type="button" id="button_id" value="Click!">
</form>
</center>

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

Lo script è “importato”. Avremmo anche potuto inserire il nostro codice js direttamente tra i tag <script>. Stiamo disegnando un normalissimo form HTML, con 2 label, due campi di input testuali ed un pulsante. E’ importante definire almeno un attributo di “identificazione” a quegli elementi html con cui vogliamo interagire lato js. Per esempio, il bottone ed i campi di testo hanno l’attributo “id” specificato.

# Lo script

```
//costante
const message = "Titolo";

//ottenimento di un elemento HTML tramite id
document.querySelector("#header").innerHTML = message;

//variabile e scrittura alternativa per l'ottenimento di elementi dal documento.
var header = document.getElementById("header");

//manipolazione dei valori (concatenazione di stringhe)
header.innerHTML += " della Pagina";
```


```
<h2 id="header"> Titolo </h2>

    <form id="form_id">
    <label for="campo1">Campo1:</label><br>
    <input type="text" id="campo1_id" name="campo1"><br>
    <label for="campo2">Campo2:</label><br>
    <input type="text" id="campo2_id" name="campo2"><br><br>
    <input type="button" id="button_id" value="Click!">
</form>
</center>

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

# Modifichiamo il pulsante

```
//prima cosa, otteniamone un riferimento

var b = document.getElementById("button_id");

//modifichiamone il suo aspetto

b.value = "Mostra Alert"

/* E' diverso da innerHTML… il valore "mostrato" dal
pulsante si trova nella proprietà value */
```

# In generale

<tag attribs=....> innerHTML </tag>

<tag attribs=....>

In un caso modifichiamo il contenuto **tra i tag (innerHTML),** nell'altro invece ne modifichiamo gli **attributi**.

# Gestione eventi: click del pulsante

Due modi:

- Staticamente in HTML
- Dinamicamente in js

Modo statico:

```
<input type="button" id="button_id" onclick="onClick()"
value="Click!">
```

Presume l’esistenza di una funzione chiamata **onClick** nello script

# Dinamicamente + codice funzione

```
//gestione degli eventi:
var b = document.getElementById("button_id");
b.value = "Mostra Alert"
b.onclick = onClick;

function onClick(){
    window.alert("Data: " + Date())
}
```

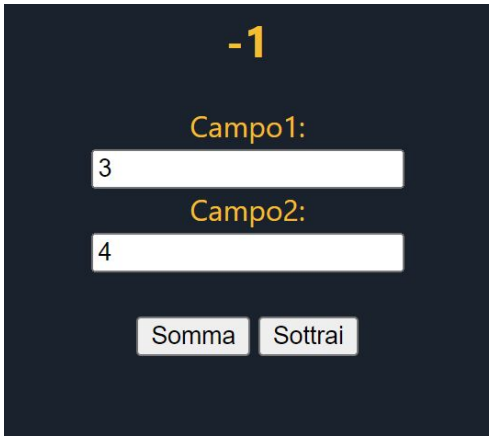
# Parametri nelle funzioni ascoltatrici

Si supponga di complicare il nostro form: i pulsanti sono ora due, uno per la somma, l'altro per la sottrazione dei valori immessi nei campi di testo editabili. All'evento *onclick* di entrambi i pulsanti deve corrispondere la funzione *operazione* che opererà una somma od una sottrazione in accordo al pulsante premuto.

```
<h2 id="header"> Titolo </h2>
    <form id="form_id">
    <label for="campo1">Campo1:</label><br>
    <input type="text" id="campo1_id" name="campo1"><br>
    <label for="campo2">Campo2:</label><br>
    <input type="text" id="campo2_id" name="campo2"><br><br>
    <input type="button" id="sum_id" onclick = "operazione(this.id)"
value="Somma">
    <input type="button" id="sub_id" value="Sottrai">
</form>
```

# Lo script

```
var subBtn = document.getElementById("sub_id");
subBtn.onclick = function (id) { operazione(subBtn.id); }

function operazione(id){

    const a = parseInt(document.getElementById("campo1_id").value);
    const b = parseInt(document.getElementById("campo2_id").value);

    switch(id){
    case "sum_id":
        header.innerHTML = a+b;
    break;
    default:
        header.innerHTML = a-b;
  }

}
```



# In generale:

Da HTML:

```
<elemento evento=”funzione(this…)” … >
```

Dinamicamente in js puro:

```
elemento.nomeEvento = funzione; //niente params

elemento.nomeEvento = //con parametri

function(param names …) { funzione(param values…);};
```

# Altri eventi?

Consideriamo che i widget HTML sono assimilabili a quello che abbiamo visto con java AWT-Swing. Diversi componenti scatenano diversi **eventi** tipicamente comandati dagli utenti.

La mia logica può **ascoltare** tali eventi e legarci una **function di callback.**

Esistono eventi **temporizzati**, i.e. **threads\timer** che periodicamente mi restituiscono un evento.

Infine, esistono anche gli eventi dati dalle **operazioni asincrone:** lancia una richiesta ad una risorsa locale o remota (lettura di un file, risposta da sito web) in background e **segnalami un evento** quando finisci…

# Eventi comuni agli elementi HTML

lista completa: https://www.w3schools.com/jsref/dom_obj_event.asp

| Event | Description |
|---|---|
| onchange | An HTML element has been changed |
| onclick | The user clicks an HTML element |
| onmouseover | The user moves the mouse over an HTML element |
| onmouseout | The user moves the mouse away from an HTML element |
| onkeydown | The user pushes a keyboard key |
| onload | The browser has finished loading the page |

Esistono poi eventi specifici per elementi di tipo specifico…

# onkeyup di input type=”text”

Sempre con il nostro sommatore\sottrattore, proibiamo all’utente di inserire testo non traducibile in **intero**.

*onkeyup* lancia un evento ogni qual volta la  modifica all’input text è stata completata.

```
    <input type="text" id="campo1_id" name="campo1" onkeyup="check(this)"><br>
    <label for="campo2">Campo2:</label><br>
    <input type="text" id="campo2_id" name="campo2",
onkeyup="check(this)"><br><br>

const re = new RegExp(/^[-+]{0,1}\d+$/);
function check(e){
    if(!re.test(e.value))
        e.value = e.value.substring(0,e.value.length-1);
}
```

# Eventi periodici


*<window.>setInterval(funzione, periodo in ms)*

Esempio:

```
header = document.getElementById(“header”);

setInterval(orologio, 500);

function orologio(){

    var d = new Date();

    header.innerHTML =
    d.getHours()+":"+d.getMinutes()+":"+d.getSeconds();

}
```

# Altri esempi js/css/HTML

sono nel git del corso a partire dalla cartella *js/*

In particolare:

- Tris, con due giocatori umani.
  - js/tris/
- Wordle clone
  - js/wordleclone/
  - Ultra semplificato, database piccolo di parole inglesi,
  - Non è necessario inserire parole esistenti
  - Non è possibile cancellare le lettere inserite

# Il Tris (.html)

```html
<!DOCTYPE html>
<html>
  <style>
    <link rel="stylesheet"href="src/style.css">

  </style>

  <body>

    <center>
      <h2 id="header">Tris!</h2>

      <table onclick="click()">
        <tr>
          <td bgcolor="oldlace" id="c00" onclick="onClick(this)"> </td>
          <td bgcolor="oldlace" id="c01" onclick="onClick(this)"> </td>
          <td bgcolor="oldlace" id="c02" onclick="onClick(this)"> </td>
        </tr>
        <tr>
          <td bgcolor="oldlace" id="c10" onclick="onClick(this)"> </td>
          <td bgcolor="oldlace" id="c11" onclick="onClick(this)"> </td>
          <td bgcolor="oldlace" id="c12" onclick="onClick(this)"> </td>
        </tr>
        <tr>
          <td bgcolor="oldlace" id="c20" onclick="onClick(this)"> </td>
          <td bgcolor="oldlace" id="c21" onclick="onClick(this)"> </td>
          <td bgcolor="oldlace" id="c22" onclick="onClick(this)"> </td>
        </tr>
      </table>

    </center>
  </body>

  <script scr="src/script.js"></script>

</html>
```

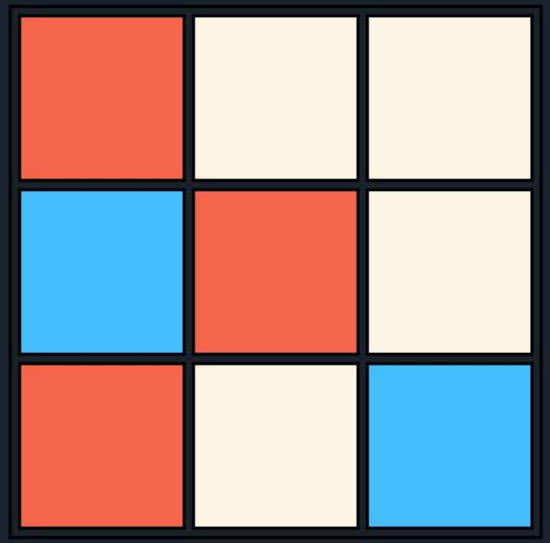
Una tabella <table>:
3 colonne <td> per 3 righe <tr>
Attributo id in funzione delle “coordinate”
Stessa funzione ascoltatrice per tutti

# Il Tris (.js)

```
var winningIndices = [
  new Set([0, 1, 2]),
  new Set([3, 4, 5]),
  new Set([6, 7, 8]),
  new Set([0, 3, 6]),
  new Set([1, 4, 7]),
  new Set([2, 5, 8]),
  new Set([0, 4, 8]),
  new Set([2, 4, 6])
];

curPlayer = 0;
playerColor = ["Tomato", "DeepSkyBlue"]

var moves = [new Set(), new Set()];

const title = "Tris: tocca a te player ";
var header = document.getElementById("header");
var clicked = 0;
header.innerHTML = title + (curPlayer + 1);

function onClick(e) {

  console.log("clicked " + e.id);
  row = parseInt(e.id.charAt(1));
  col = parseInt(e.id.charAt(2));
  e.style.backgroundColor = playerColor[curPlayer]
  e.onclick = nope;
  moves[curPlayer].add(col + row * 3);
  checkWin();
}
```

```
function checkWin() {

  if (moves[curPlayer].size >= 3) {
    for (var i = 0; i < winningIndices.length; i++) {
      var wmov = winningIndices[i];
      var intersect = new Set();
      for (var x of wmov)
        if (moves[curPlayer].has(x)) intersect.add(x);
      if (intersect.size == 3){
          alert("Player " + (curPlayer + 1) + " you Win!");
        }
    }
  }

  curPlayer = curPlayer == 0 ? 1 : 0;
  header.innerHTML = title + (curPlayer + 1);
  clicked++;
  if (clicked == 9){
    alert("Parità!")
  }

}

function nope() {
  alert("Già Cliccato!");
}
```

# Tris: tocca a te player 2



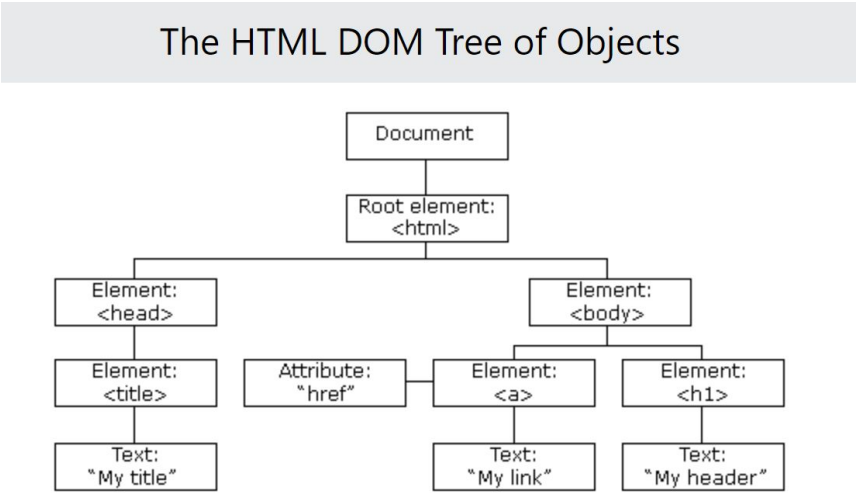
# Wordleclone

Anche in questo caso, la visualizzazione può essere basata su delle tabelle.

In particolare, una **tabella di gioco** in cui si visualizzano le lettere delle parole da indovinare per i vari tentativi ed una tabella “**tastiera**” per visualizzare le lettere dell’alfabeto a disposizione.

E’ uno scenario in cui possiamo focalizzarci su come usare javaScript per manipolare il documento non solo in termini di eventi e proprietà dinamiche, ma anche per **aggiungere\togliere** elementi HTML in maniera programmatica.

# DOM manipulation




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

DOM: Document Object Model.
E' una rappresentazione ad albero degli elementi presenti all'interno del documento HTML.

Questa struttura può essere "lunga" e "larga" in maniera arbitraria.

Ogni nodo rappresenta un elemento che estraiamo dal documento per poi modificarne le proprietà tramite lo script in js.

Questo lo abbiamo visto…

L'idea ora è quella di estrarre un nodo da questo albero ed aggiungerne dei figli. In questo modo possiamo comporre in maniera dinamica e programmatica una pagina web tramite JavaScript.

# Nel wordleclone

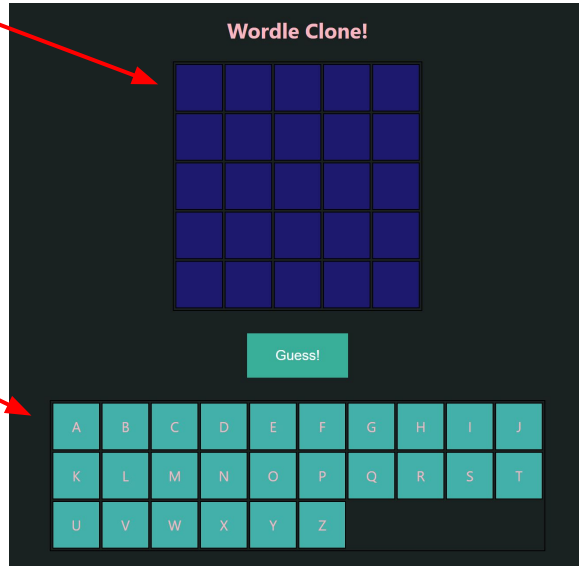
IDEA:

anzichè “hardcodare” in HTML tutte le tabelle e le loro rispettive celle (sono tante!), usiamo la DOM manipulation per estendere il grafo di elementi della nostra pagina HTML usando js.

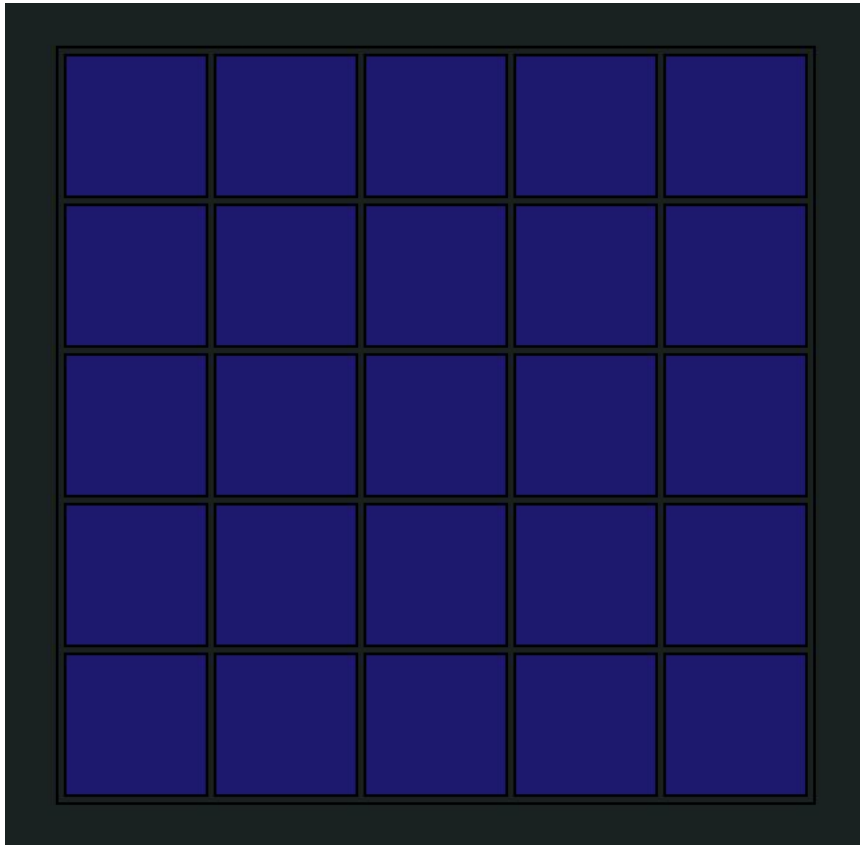
In HTML puro, possiamo partire con 2 tag <table>, uno per la tabella di gioco, l’altro per la tabella tastiera. Una volta dato un id a ciascuna di esse, possiamo aggiungerne i **figli** tramite js usando **`document.createElement` ed `element.appendChild`**

# HTML

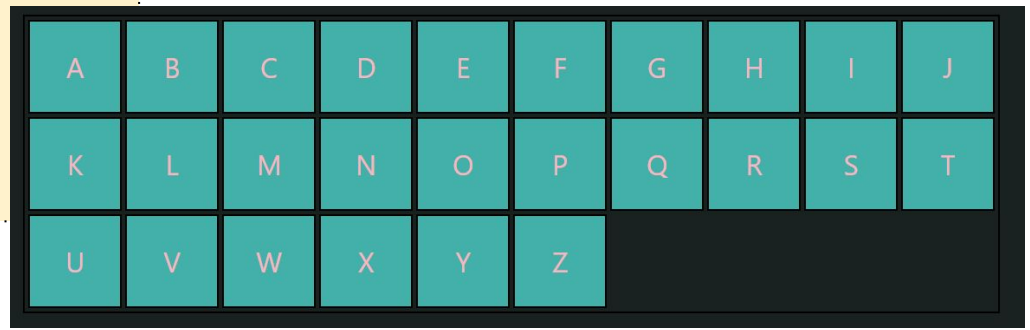
```
<h2>Wordle Clone!</h2>
<table id="tableGame_id">
</table>
<br>
<input type="button" class="button" onclick="guess()" value="Guess!">
<br><br>
<table id="tableA_id">
</table>
```



```
  //costruzione tabella di gioco
var gameTable =
document.getElementById("tableGame_id");
var tr = null;
  // 5 tentativi * 5 lettere
for (var i = 0; i < 25; i++) {

    if (i % 5 == 0) { //una riga ogni 5
      tr = document.createElement("tr");
      tr.id = "tr" + i + "_id";
      gameTable.appendChild(tr);
    }

    var node = document.createElement("td");
    node.id = i + "_id";
    node.style.backgroundColor = "MidnightBlue";
    node.innerHTML = " ";
    tr.appendChild(node);
  }
```



```
  //costruzione tabella tastiera
const ENG_ALPHA = ["A", "B", …
var table = document.getElementById("tableA_id");
tr = null;

  for (var i = 0; i < ENG_ALPHA.length; i++) {

     if (i % 10 == 0) {
       tr = document.createElement("tr");
       tr.id = "trA" + i + "_id";
       table.appendChild(tr);
     }

     var node = document.createElement("td");
     node.id = ENG_ALPHA[i] + "_id";
     node.style.backgroundColor =
     "LightSeaGreen";
     node.onclick = function()
     { onClick(this); };
     node.innerHTML = ENG_ALPHA[i];
     tr.appendChild(node);

  }
```



# Possibile esercizio

Fare un progetto in django con due app.

Una per il tris, l’altra per il wordleclone.

Da un url “home” del root prj si può selezionare quale gioco.

Servire opportunamente le risorse *static* utilizzate, i.e. file js e css…

# Quindi con JavaScript?

Possiamo avere logica lato client.

Però, quello che abbiamo visto finora è **completamente contenuto** lato client.

Possiamo fare interagire il nostro client in maniera dinamica e tramite js ad un server? **Magari gestito da noi ed implementato in Django?**

**Terza categoria di eventi:**

*Esistono anche gli eventi dati dalle **operazioni asincrone:** lancia una richiesta ad una risorsa locale o remota (lettura di un file, risposta da sito web) in background e **segnalami un evento** quando finisci…*

# Web requests asincrone

AJAX: asynchronous JavaScript and XML.

E’ lo strumento che ci serve.

Appare allo sviluppatore js come un oggetto.

Su tale oggetto possiamo chiamare metodi, che (di default) in maniera asincrona possono eseguire richieste dal client verso un url remoto gestito da un server.

Riceveremo una callback per l’evento di completamento della richiesta.

Trivialmente, tale richiesta si porta dietro una risposta, che noi possiamo usare nel nostro script…

# AJAX: come si usa

```
const xhttp = new XMLHttpRequest();

xhttp.onload = function() { … } //funzione di callback

//definisci i parametri della richiesta

xhttp.open(metodo_richiesta, url_richiesta);

xhttp.send(); //inoltra la richiesta
```

# Osservazioni

La “X” che sta per “XML” non deve trarre in inganno. I dati scambiati dal server verso i client possono anche **non** essere formattati in XML. Tipicamente si usano stringhe normali o **JSON.**

Il metodo **open** ci permette di definire il metodo (e.g. “POST” o “GET”), l’url da contattare ed un terzo parametro opzionale (booleano) che indica se la richiesta è da intendersi asincrona. Di default è settato a **true.** L’asincronicità implica che il nostro script continua ad eseguire dopo aver mandato la richiesta, per poi interrompersi per eseguire la callback indicata con **onload.**

All’interno della callback, gli attributi **status** e **responseText** ci permettono di avere la risposta del server e soprattutto il codice HTTP di tale risposta.

# GET / POST in AJAX

```
xhttp.open("GET", “url/?param1=value1&param2=value2”);
xhttp.send();
```

```
xhttp.open("POST", url);
xhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
xhttp.send("param1=value1&param2=value2");
```

# Esempio [biblio3]

Possiamo prova ad integrare js+AJAX in un progetto Django già esistente.

Per esempio la terza versione della biblioteca che (tra le altre cose) aveva una funzione di ricerca per i libri.

In particolare, potevo scegliere una keyword di ricerca per titolo di un libro o per autore ed django mi redirezionava in un'altra pagina con il risultato della ricerca.

Con una combinazione di tecniche che abbiamo imparato in questa slide, possiamo aiutare l'utente a fare questa ricerca tramite **autocompletamento.**

In altre parole, **mentre l'utente scrive la keyword, una richiesta AJAX contatta una view di django che autocompleterà la stringa che si sta scrivendo**.

# ricerca_ajax.html

(Nuova aggiunta al) template utilizzato dalla view *search* in gestione/views.py.

```
<script type="text/javascript">
  var stringText = document.getElementById("id_search_string");
  stringText.onkeyup = autoComplete;
  stringText.autocomplete = "off";

  function autoComplete(){
    var s = stringText.value;
    if(s.length<3){ return;}
    else{
      const w = document.getElementById("id_search_where").value;
      const xhttp = new XMLHttpRequest();
      xhttp.onload = function() {
          if(xhttp.status == 200)
            stringText.value = this.responseText;
      }
      xhttp.open("GET", "gethint/?w=" + w + "&q=" + s);
      xhttp.send();
    }
  }
</script>
```

# Raggiungibilità e logica lato server

in gestione/urls.py in urlpatterns

```
path("ricerca/gethint/", get_hint, name="get_hint")
```

in gestione/views.py

```
def get_hint(request):

    response = request.GET["q"]

    if(request.GET["w"]=="Titolo"):
        q = Libro.objects.filter(titolo__icontains=response)
        if len(q) > 0:
            response = q[0].titolo
    else:
        q = Libro.objects.filter(autore__icontains=response)
        if len(q) > 0:
            response = q[0].autore

    return HttpResponse(response)
```