Tutorato di:

Web Programming, Design & Usability

Tutor: Alessio Tudisco

Università di Catania

HTTP in pillole

Il protocollo su cui si basa il web

In breve...

01
È un protocollo del *application layer*

Permette la **comunicazione** in **ambienti distribuiti**: *fra vari host e vari client*.

02
È un protocollo stateless

Non conserva informazioni sullo stato dei client che effettuano la comunicazione.

03
La comunicazione avviene tramite URL

Un URL (**Uniform Resource Locator**) identifica una risorsa nel web

04
L'azione è descritta dai verbi HTTP

L'intento sulla risorsa è descritto dai 4 verbi HTTP: **GET**, **POST**, **PUT**, **DELETE**.

05
A una richiesta segue una risposta

Il client invia al server una **richiesta HTTP**, quest'ultimo replica con una **risposta HTTP** .

06
Le risposte hanno un codice di stato

Il codice di stato può aiutarci a capire **l'esito della richiesta**.

Piccoli dettagli

Struttura di un URL	$[PROTOCOL]://[DOMAIN]:[PORT]/[RESOURCE]?[QUERY]$
L'azione descritta dai verbi HTTP	❖ **GET:** per la richiesta di una risorsa esistente; ❖ **POST:** per la creazione di una nuova risorsa; ❖ **PUT:** per l'aggiornamento di risorsa esistente; ❖ **DELETE:** per la rimozione di una risorsa esistente;
Specifiche del metodo GET	Le informazioni vengono trasmesse **in chiaro** inserendole **nella sezione QUERY** dell'URL, non è adatto per comunicare informazioni sensibili. Le richieste GET **possono**: **essere cachate**, **creare cronologia** e **essere salvate nei preferiti.** Supportano **solo parametri testuali** (ma è possibile usare la base64 per dati non testuali). Si è limitati dalla **lunghezza massima degli URL** supportata dagli applicativi.
Specifiche del metodo PUT	Le informazioni vengono poste nel **body** della **richiesta HTTP**, può essere usato per comunicare informazioni sensibili. Le richieste POST **non possono**: **essere cachate**, **creare cronologia** e **essere salvate nei preferiti.** Supportano **vari tipi di parametri** (testuale, binario, numerico, ect...). Non si è limitati dalla **lunghezza massima degli URL** supportata dagli applicativi.
Classificazione degli Status Code:	**1XX**: per i messaggi informativi; **2XX**: per i messaggi di operazione riuscita; **3XX**: per i messaggi di avvenuto caching; **4XX**: per i messaggi di errore lato client; **5XX**: per i messaggi di errore lato server;

HTML

L'*Hyper-Text Markup Language* non è un linguaggio di programmazione!

In breve...

L'HTML è il linguaggio di markup standard per la creazione di pagine web: rappresenta la struttura di una pagina web attraverso dei ***<tag>***.

Deriva dal meta-linguaggio XML e appartiene alla famiglia degli ***Standard Generalized Markup Language*** *(SGML), è un linguaggio case-insensitive.*

Il ruolo di un **web browser** è quello di leggere un **documento HTML** e visualizzarne il contenuto.

I documenti HTML prevedono un **tag iniziale** che aiuta i web browser a visualizzare correttamente i siti web.

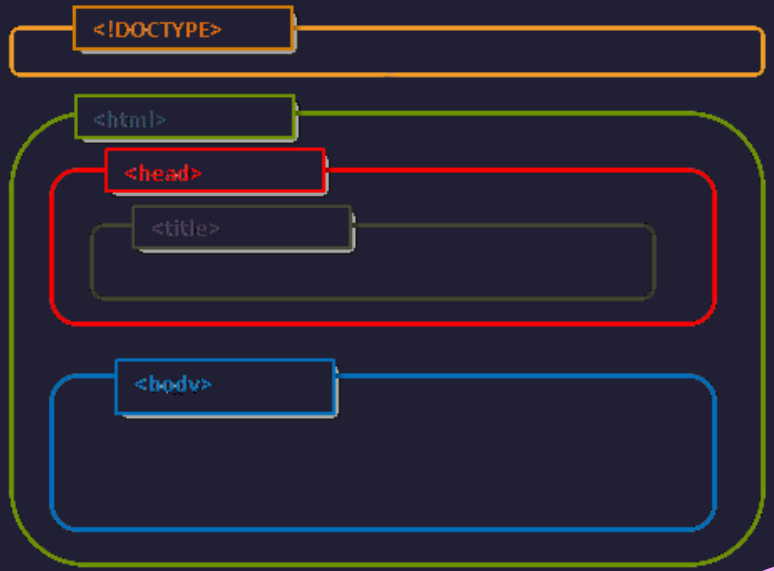
Tale tag prende il nome di **DOCTYPE** e per HTML5 ha la seguente forma:

$$<!DOCTYPE\ html>$$

La struttura HTML in pillole

A seguire del *DOCTYPE* vi è un tag contenitore principale denominato **<html>** che contiene 2 parti fondamentali:

- **Head**: è una sezione dedicata ai metadati od altre informazioni che non contribuiscono visivamente alla pagina web, come ad esempio il titolo della scheda del browser o i metadati di OpenGraph;
- **Body**: è la sezione dedicata agli elementi che contribuiscono visivamente alla pagina web, ovvero contiene tutti gli elementi che l'utente vedrà e con cui potrà interagire;

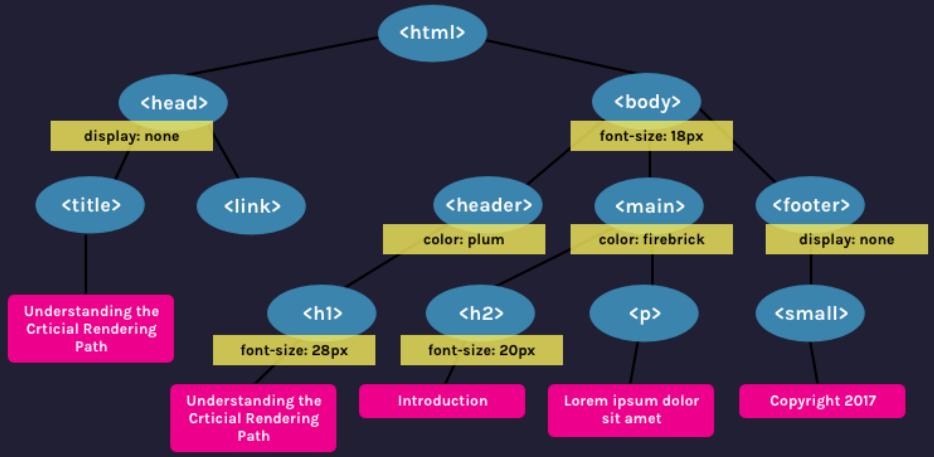


Una pagina HTML è rappresentabile da un **albero n-ario**, in cui tutti i nodi sono elementi HTML. Quest'ultimi possono essere annidati, ovvero contenere altri elementi HTML.

Generalmente, un elemento HTML inizia con un tag di apertura, e, se prevede un contenuto, un tag di chiusura.

Vi sono elementi HTML denominati void elements che non prevedono il tag di chiusura.

- $< H1 > Titolo <\text{/H1} >$
- $< br/>$

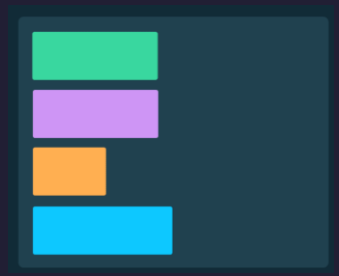


Classificazione elementi HTML

Elementi di tipo block

[DIV, P, H#, UL, OL, LI, HR]

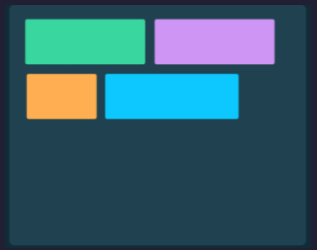
- Iniziano sempre a capo;
- Prendono tutta la larghezza del elemento parent;
- Hanno margini superiori e inferiori;



Elementi di tipo inline

[SPAN, A, IMG, I, B]

- Non iniziano a capo, si accostano agli altri elementi inline;
- Assumono la larghezza minima necessaria per il proprio contenuto;
- Non hanno margini superiori e inferiori;



Attributi nativi degli elementi

Gli attributi sono utilizzati per definire le **caratteristiche** di un elemento HTML, vengono inseriti all'interno del **tag di apertura** dell'elemento HTML. Hanno una forma di tipo **chiave-valore**.
Tutti gli elementi HTML possiedono almeno i **4 attributi nativi**: $[KEY = VALUE]$

- **ID**: un identificatore **univoco** dell'elemento;
- **CLASS**: un identificatore di **relazione** che raggruppa più elementi HTML;
- **STYLE**: permette di definire uno stile per l'elemento HTML;
- **TITLE**: una descrizione che appare tramite tooltip, spesso utilizzata dai narratori per fornire maggiore accessibilità agli utenti non vedenti;

È possibile utilizzare anche dei **meta-attributi**, ovvero attributi non standard del HTML ma custom, che possono contenere informazioni utili per il corretto funzionamento della pagina web.
Per definire un meta-attributo si utilizza il prefisso **data-**: $[data - KEY = VALUE]$

Carrellata di elementi HTML

Titoli (H1-H6)

Elemento block:

$< H\# > Titolo </H\# >$

Paragrafi

Elemento block:

$< p > Testo </p >$

Link (O ancore)

Elemento inline:

Testo

Immagini

Elemento inline:

Elenchi Puntati

Elemento block:

$< ol >< li > \cdots </li $

Tabelle

Elemento block:

... Troppo lungo ...

Divisore

Elemento block:

 ...

iframe

Elemento inline:

<iframe src='...' ></iframe>

Form e Input

Elementi inline:

Prossima slide...

Form e Input

Form

È un elemento HTML di tipo **block** interattivo con cui l'utente interagisce al fine di inviarci delle informazioni. È un contenitore di **elementi input**.

È caratterizzato dai seguenti attributi:

- **action**: definisce l'azione da eseguire al submit, generalmente uno script su un server;
- **method**: specifica il verbo http da utilizzare al submit, un form può eseguire GET o POST;

Input

È un elemento HTML void di tipo **inline** che può raccogliere un dato di una particolare forma, può assume le sembianze di una: textfield, checkbox, radio button, combobox, etc.

È caratterizzato dai seguenti attributi:

- **type**: definisce la tipologia di dato che può raccogliere;
- **name**: specifica il nome dell'input, ovvero l'identificativo con cui il dato viene inviato. Se omesso l'input non verrà inviato dalla form;

Attributi opzionali degli Input

value	il valore iniziale dell'input
placeholder	un suggerimento su che tipo di valore inserire
readonly	specifica che l'input non è modificabile
disabled	specifica che l'input è disabilitato e non verrà incluso nella richiesta HTTP
required	specifica che l'input è necessario per poter inviare la richiesta HTTP, se non viene compilato tale input al submit si otterrà un avviso
checked	specifica che l'input di tipo checkbox o radio è selezionato in modo predefinito
pattern	un regex con il quale l'input può validare il dato che raccoglie
maxlength	la massima lunghezza del testo raccoglibile dalla textarea
min e max	il valore minimo e quello massimo accettabili dall'input, può essere un numero o una data

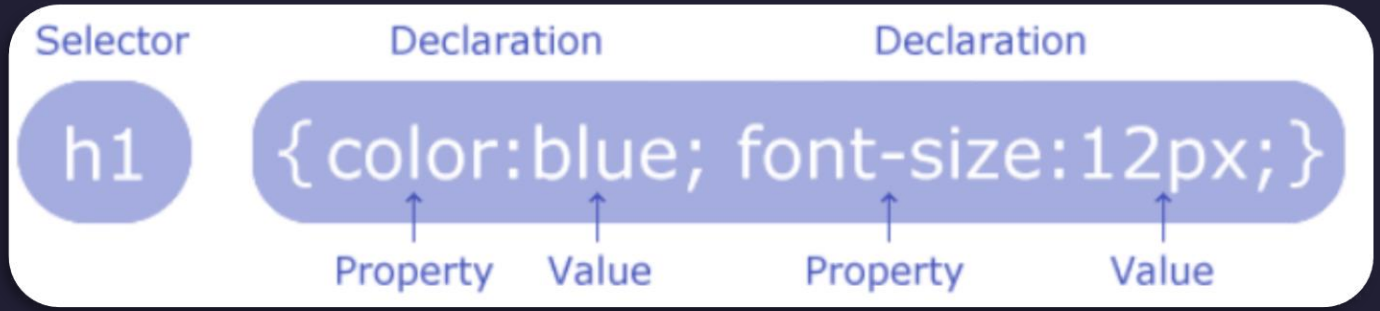
CSS

Il Cascading Style Sheet è un linguaggio di stile

CSS in pillole

È un **linguaggio di stile** che permette di descrivere il layout delle pagine web. Consiste in una serie di regole, aventi forma **chiave-valore**, che vengono applicate a cascata, ovvero dall'alto verso il basso in cui: se una regola viene definita più volte, prevale la sua ultima definizione.

Vengono utilizzati dei **selettori** per individuare gli elementi dell'albero HTML a cui applicare le regole contenute nel declaration block.



Tipi di Selettori

Element Selector

Match tag HTML

$p\ \{\dots\}$

ID Selector

Match per ID

$\#p1\ \{\dots\}$

Class Selector

Match per classe

$.yellow\ \{\dots\}$

Universal Selector

Match qualunque elemento

$*\ \{\dots\}$

Group Selector

Match multiplo separato da virgola

$.yellow\ ,\#p1\ \{\dots\}$

Position Selector

Match basato sulla annidamento posizione degli elementi:

$div\ p\ \{\dots\}; div > a\ \{\dots\}; ect\ \dots$

Attribute Selector

Match basato sugli attributi

$img[alt = "\dots"\ \{\dots\}$

Pseudo-Class Sel.

Match per pseudo-classi:

:active, :checked, ect...

Pseudo-Element Sel.

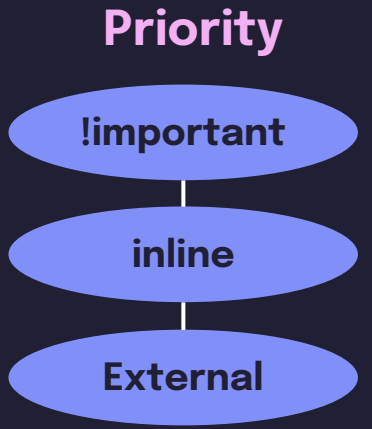
Match per pseudo-elementi:

::before, ::after, ect...

Definire regole CSS

Una regola CSS può essere definita inline, quando è definita attraverso l'attributo **style** dell'elemento HTML, oppure nel foglio di stile CSS , che può essere interno o esterno al documento HTML.

Date le molteplici locazioni in cui è possibile definire le regole CSS, vi è una **gerarchia o priorità di applicazione**: esiste la keyword **!important** che associa la massima priorità indipendentemente dalla posizione in cui si trova la regola.

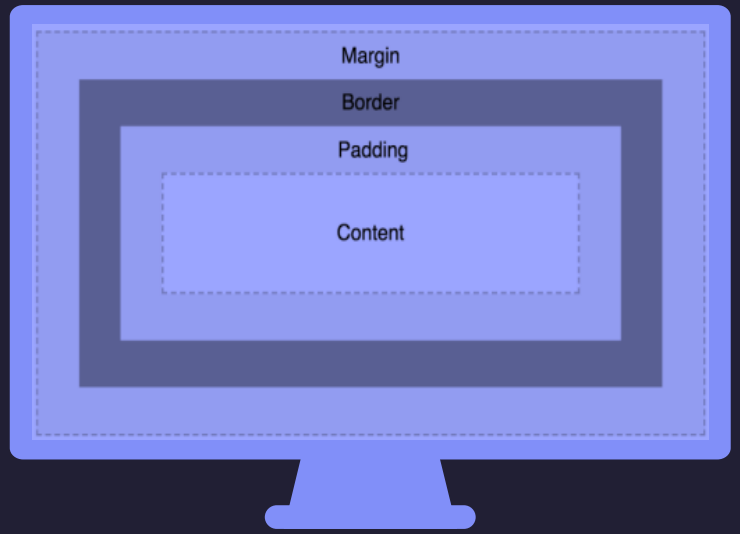


Il foglio di stile CSS può essere definito direttamente nell'**head** del documento HTML, tramite $< style >$, oppure in un file css esterno che dovrà essere caricato inserendo sempre nell'**head** il tag $< link >$

```
<link rel="stylesheet" type="text/css" href="/my website/styling.css">
```

Il Box Model

L'engine dei browser renderizza ogni elemento HTML come un rettangolo seguendo lo standard denominato **box model**. Ogni box (rettangolo) è costituito da quattro parti (o aree): il **margine**, il **bordo**, il **padding** e il **contenuto**.



Overview su alcune regole CSS

Vi sono tante regole CSS, che si possono studiare in autonomia. Vedremo solo regole particolari...

display	**specifica il tipo di rendering da applicare ad un elemento HTML:** **block, inline, none, inline-block, flex, grid** Il rendering **none** rimuove l'elemento dalla pagina web, inibendone pure gli eventi collegati
position	**definisce come un elemento è posizionato nel documento HTML** ❖ **static:** valore predefinito, rappresenta il naturale posizionamento; ❖ **relative:** definisce un offset rispetto al posizionamento naturale; ❖ **absolute:** definisce un offset rispetto al primo antenato **non-static**, inoltre l'elemento viene rimosso dal flusso di rendering ed renderizzato a parte (l'elemento non influenza più il layout); ❖ **fixed:** simile all'absolute con la differenza che l'offset è sempre rispetto alla **viewport**; ❖ **Sticky**: un posizionamento legato allo scrolling, esegue uno switch fra posizionamento **relativo** e **fixed**;
Float e **clear**	**specifica che un elemento debba "fluttuare", permettendo agli elementi inline di posizionarsi attorno ad esso.** L'elemento fluttuante viene rimosso dal flusso di rendering ma influenza ancora il layout. La regola **clear** viene utilizzata per «pulire» l'effetto fluttuante: googlate **clearfix** !

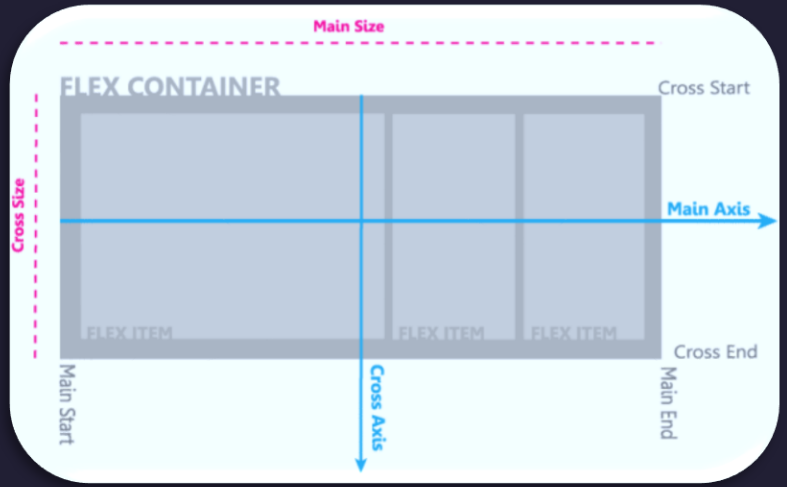
CSS Responsive

Flexbox & Grid System

FlexBox in pillole

Il flexbox è un layout responsive per disporre elementi in un ambiente unidimensionale, o in riga o in colonna. *Permette di centrare verticalmente i figli rispetto al padre*.

Vi è un **flex-container** che contiene **flex-items**. Tutti i figli diretti di un flex-container sono considerati flex-items.



Un contenitore è definito tramite «$display: flex$»

- **Asse principale**: asse che va per la direzione con la quale vengono visualizzati gli ottetti (Es. da sinistra a destra per le righe);
- **Asse trasversale**: asse perpendicolare a quella principale;

Regole CSS per Flexbox

Regole del container

Flex-direction	**specifica la direzione in cui impilare i flex-item.** Può assumere: **row, row-reverse, column e column-reverse** **La direzione colonna inverte le assi**
Flex-wrap	**specifica il comportamento in caso di overflow dei flex-item rispetto alle dimensioni del flex-container.** Può assumere: **nowrap, wrap e wrap-reverse** **Il wrap permette ai flex-item in overflow di andare a capo**
Justify-content	**specifica la disposizione dei flex-item per l'asse principale.** Può assumere: **flex-start, flex-end, center, space-between, space-around, space-evenly**
Align-items	**specifica la disposizione dei flex-item per l'asse trasversale.** Può assumere: **flex-start, flex-end, center, stretch**

Regole degli oggetti

Flex	**regola aggregatore per flex-grow, flex-shrink e flex-basis** ❖ **Flex-grow:** specifica una crescita proporzionale rispetto agli altri elementi del container; ❖ **Flex-shrink:** specifica un rimpicciolimento proporzionale rispetto agli altri elementi del container; ❖ **Flex-basis**: specifica la lunghezza iniziale di un oggetto;
Align-self	**Sovrascrive la regola align-items del container. Assume gli stessi valori di quest'ultima**
order	**specifica la posizione dell'item rispetto agli altri item del flex-container**

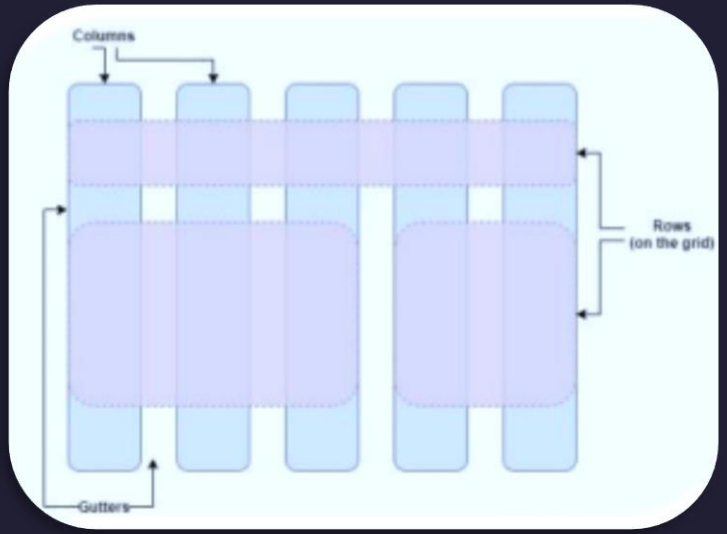
Grid System in pillole

Il grid system è un layout responsive per disporre elementi in un ambiente bidimensionale (matrici).

Vi è un **grid-container** che contiene **grid-items**. Tutti i figli diretti di un grid-container sono considerati grid-items.

Un contenitore è definito tramite «*display*: *grid*»

- Definiamo righe e colonna della griglia;
- Le spaziature fra righe e colonne prendono il nome di **gap**;



Regole CSS per Grid System

Regole del container

grid-template-[columns|rows]	**specifica il numero di [columns|rows] della griglia** Possiamo fornire come parametro le lunghezze per ogni [column|row]
gap	**specifica la dimensione della spaziatura fra righe e della spaziatura fra colonne**
Grid-template-areas	**specifica una descrizione a pattern della griglia, ci permette di assegnare etichette e span alle celle della griglia**
[Justity|Align]-items	**specificano la disposizione dei grid-item rispettivamente nella l'asse delle righe e nell'asse delle colonne.** Può assumere: **start, end, center, stretch**

Regole degli oggetti

grid-[column|row]	**specifica il range di [column|row] che l'item ricopre.** Accetta due parametri separati da un forward slash: il primo indica la [column|row] di inizio (inclusa) e il secondo quella di fine (esclusa) È possibile **span** dopo il forward slash per utilizzare il secondo parametro come numero di estensioni
Grid-area	**specifica il posizionamento dell'item in un'area definita nel grid-template-areas del container** Accetta come parametro una stringa che equivale all'etichetta dell'area
[Justity|Align]-self	**Sovrascrivono le regole [justify|align]-items del container. Assumono gli stessi valori di quest'ultime**

JavaScript

Un linguaggio di programmazione interpretato, non è imparentato con Java!

In breve...

JavaScript è un **linguaggio di programmazione interpretato multi-paradigma** (programmazione funzionale, imperativa, asincronicità**),** **debolmente tipizzato** e **a singolo thread di esecuzione**, ovvero può eseguire solo un'istruzione alla volta, non bloccante.

Ogni browser ha un engine JS che esegue il codice, tale ***esecuzione*** *è relativamente* ***sicura*** *poiché il browser esegue ogni finestra dentro una* ***sandbox****, che non permette l'accesso diretto al sistema host dell'utente.*
Fanno eccezione le estensioni dei browser...

Un codice JS può essere aggiunto a una pagina web in modo analogo al CSS: **codice embeded** o **esterno.**

Possiamo aggiungerlo direttamente nel documento HTML, tramite $< script >$, oppure in un **file js esterno** che dovrà essere caricato usando l'attributo **src** del tag $< script >$

È **consigliato** l'inserimento del codice tramite un tag appena prima la chiusura del body per assicurare il **completo caricamento del DOM** prima dell'esecuzione del codice

Variabili e Hoisting

Variabili

In JS le variabili **devono iniziare** con una lettera, un underscore (_) o un segno del dollaro ($).

Possiamo dichiararle tramite:

- **var**: può definire variabili globali o locali, quest'ultime aventi un **function-scope**;
- **let**: può definire variabili locali aventi un **block-scope**, ovvero sopravvivono fino alla chiusura del blocco più vicino;
- **const**: può definire costanti locali aventi anch'esse un **block-scope**;

Le variabili non inizializzate hanno come valore predefinito il valore **undefined**.

Hoisting

È un **meccanismo** che consiste nello **spostare "logicamente"** le **dichiarazioni** di **variabili** (**dichiarate con var**) e **funzioni** nella parte superiore del codice al fine di potervi accedere in qualunque punto del codice.

```
var a;
console.log('The value of a is ' + a); // The value of a is undefined

console.log('The value of b is ' + b); // The value of b is undefined
var b; Hoisting :)

console.log('The value of c is ' + c); // Uncaught ReferenceError: c is not defined

let x;
console.log('The value of x is ' + x); // The value of x is undefined

console.log('The value of y is ' + y); // Uncaught ReferenceError: y is not defined
let y; No hoisting :(
```

JavaScript 101

Tipi di variabili

String, Number (int e float), boolean, undefined, null, object (collezione chiave-valore), function

Costrutti

If, switch, while, for analoghi agli altri linguaggi comuni... tranne Python con lo swtich

Array

Gli array object sono liste di alto livello (iterabili). Definizione analoga agli altri linguaggi comuni.

Funzioni

*Definizione mediante la keyword **function**, analoga agli altri linguaggi comuni.*

Funzioni anonime

*Definite tramite **espressioni di funzione**, salvabili in una variabile e passabili come argomento ad altre funzioni (callbacks?)*

Funzioni Arrow

Definite tramite una **variante più compatta dell'espressione di funzione**. In particolare il **this** si riferisce sempre all'oggetto che l'ha definita.

Clousure

Annidamento di funzioni, la funzione figlia ha accesso a tutte le variabili del padre

JS & DOM

Manipolazione delle pagine web tramite JS

Il Document Object Model (DOM)

È un'interfaccia di programmazione per i documenti HTML. Rappresenta la pagina, sotto forma di nodi di un albero ed oggetti, al fine di permettere ai programmi, scritti in JavaScript, di manipolare la struttura, stile e contenuto del documento HTML.

Il DOM fornisce una serie di oggetti:

- **document**: rappresenta il documento HTML;
- **window**: rappresenta la finestra del browser relativa alla pagina web;
- **element**: interfaccia che rappresenta un nodo dell'albero. Espone le sue funzioni e proprietà come: **innerHTML, style, setAttribute, getAttribute, dataset, ect...**;

Il DOM fornisce anche una serie di metodi, come:

- *Metodi di ricerca di un nodo*: **getElementByID, getElementsByClass, querySelector, ect...**;
- *Metodi di manipolazione*: **appendChild, insertBefore, prepend, cretaeElement, ect...**

Event Loop in pillole

JavaScript è un linguaggio a singolo thread di esecuzione, può eseguire solo un'istruzione alla volta ma abbiamo sicuramente visto o navigato siti in cui venivano eseguiti più compiti alla volta...

JavaScript ha un **modello di concorrenza** basato su un **event loop.** Quando viene eseguito del codice JS vengono usate due regioni di memoria:

- **stack**: usata per eseguire le funzioni e salvare una copia del loro frame;
- **head**: memoria non strutturata contenente oggetti e dati delle closure, soggetta a GC;

Il runtime JavaScript usa una **coda di messaggi**. Ad ogni messaggio è associata una funzione che deve essere richiamata per gestirlo.
I messaggi vengono processati dal più vecchio al più recente: il runtime rimuove il messaggio dalla coda ed esegue la funzione associata, creando un nuovo stack frame per la sua esecuzione. Quest'ultima continua fin tanto che lo stack non è vuoto, quando lo sarà l'event loop passerà all'esecuzione del prossimo messaggio.

Registrazione eventi

addEventListener() / removeEventListener()	assegna/rimuove una funzione da eseguire qualora l'evento indicato avvenga sul target; `myButton.addEventListener('click', greet);`
Inline attribute	Un attributo $on_{event} = "\ldots"$ nel tag HTML dell'elemento che associa una funzione da eseguire all'avvenimento dell'evento indicato; `<button onclick="alert('Hello world!')">`
Element event function	Analogo all'inline attributo, solo eseguito tramite JavaScript. `myButton.onclick = function(event){alert('Hello world');};`

JS & Asincronicità

Callbacks, Promise, Async/Await

CallBack in breve

Si definiscono **callback** quelle funzioni che vengono passate come parametro a un'altra funzione eseguita in background (come un evento o un task a lunga durata), quest'ultima al completamente della sua esecuzione eseguirà la funzione callback.

Un esempio di callback è l'addEventListener(), che prende un callback come parametro!

Promise in pillole

Una Promise fa da tramite per un valore che non si conosce al momento della creazione della promise, permettendoci di associare handler asincroni per il successo o meno. Una promise può trovarsi in uno dei seguenti stati:

- ***Pending***: lo stato iniziale, non è né completata né rifiutata;
- ***Fulfilled***: l'operazione è stata completata con successo;
- ***Rejected***: l'operazione è fallita per qualche motivo;

```
const prom =
    new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('done');
        }, 1000);
    });
```

Si crea una promessa passando come parametro al costruttore della classe Promise una funzione avente come argomenti i callback ***resolve***, *associato al fulfilled*, e **reject**, associato al rejected. il cui corpo costituisce la logica da eseguire in background che invoca **resolve** o **reject** a seconda del risultato ottenuto;

Metodi delle Promise

Then()	assegna i callback per gli stati di fulfilled e rejected. Ritorna una promise;
Catch()	assegna un callback per lo stato di rejected. Ritorna una promise
Finally()	assegna il callback da eseguire alla fine della promise indipendentemente dallo stato

Async/Await

Tramite la keyword **async** si definiscono funzioni asincrone che nel loro corpo fanno uso della keyword **await**.

Le funzioni asincrone possono essere viste come un livello di **zucchero sintattico** al di sopra del sistema delle promise: si dice che una funzione asincrona è basata su promise poiché **ha come valore di ritorno una promise**.

All'interno di una funzione asincrona, la keyword **await** può essere usata davanti al richiamo di un'altra funzione asincrona per interrompere l'esecuzione del codice fino al completamento della promise: **rende sincrona una parte della funzione asincrona**.

```
async function hello() { return greeting = await Promise.resolve("Hello"); }
hello().then(alert);
```

JS & HTTP Request

Fetch

Fetch in breve

Il **fetch** è una **API** che permette di eseguire **richieste HTTP** basandosi su un **sistema di promise**. Il metodo fetch() accetta una serie di parametri:

- **Endpoint**: l'url su cui eseguire la richiesta;
- **Options**: è un array opzionale che contiene le opzioni della richiesta. Di default il fetch esegue richieste GET, ma è possibile specificare in questo array il verbo HTTP , degli headers o il body contenente i dati di un POST;

Le Promise ottenute da un fetch() non assumeranno lo stato di Rejected qualora la risposta http contenga un error status code. **La Promise sarà rifiutata solo in caso di errori di rete**.

La Promise viene "risolta" non appena il server risponde con degli header, per controllare l'esito della richiesta si può usare la property **ok** dell'oggetto **response**, la quale sarà posta a **true** solo qualora lo status code ricevuto sia nel range 200-299.

```
fetch(url).then(r => r.json()).then(data => {console.log(data)})
```

Exercise Time

Il Quiz

Struttura

Un tipico quiz a risposte multiple. Si utilizzi il flexbox per visualizzare 2 domande per riga. Riducendo la finestra deve essere visualizzata una domanda per riga.

JavaScript

Il question-set in formato json deve essere ottenuto mediante richiesta GET e le domande devono essere inserite nel DOM dinamicamente.

Logica

Una volta fatto il submit dovete sommare gli score delle risposte e disabilitare I controlli. Il tasto reset riporta lo stato a quello iniziale.

Tips

Si prediliga un codice che non dipenda da un numero fisso di domande... usate l'iterabilità!