**INTRODUZIONE ALL’HTML** *27-09-23*

**HTML** è un linguaggio di markup, e cioè serve a dare una struttura al contenuto.

Un **documento HTML** è costituito da testo ed elementi HTML.

Un **elemento HTML** è caratterizzato da:

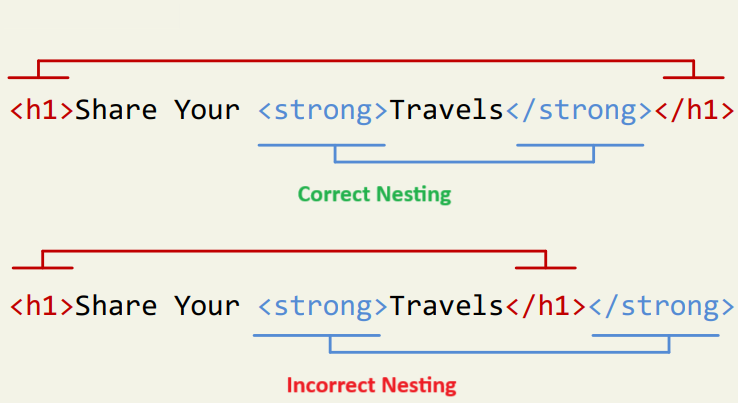
* Un *tag di apertura* e un *tag di chiusura*, tra parentesi angolari, contenenti il nome dell’elemento:
  + All’interno del tag di apertura si possono specificare eventuali attributi, che tipicamente hanno una struttura *name* = “*value*”;
  + Il nome dell’elemento all’interno del tag di chiusura deve essere preceduto da ‘/’.
* Del *contenuto* tra i due tag.



Alcuni elementi non prevedono alcun contenuto, in quanto sono solo delle istruzioni per il browser.   
Questi elementi sono detti *empty elements*, e hanno solo il tag di apertura.



Un elemento HTML può contenere altri elementi HTML. L’elemento figlio deve essere *interamente* contenuto nell’elemento padre, e cioè il tag di chiusura dell’elemento figlio deve trovarsi prima del tag di chiusura dell’elemento padre:



Ogni documento HTML presenta sempre la seguente struttura:

<!DOCTYPE html>

<html>

<head>

…

</head>

<body>

…

</body>

</html>

* **<!DOCTYPE html>:** dichiarazione (non elemento HTML) che serve a dire al browser che dovrà processare un documento HTML;
* **<html>…</html>:** elemento che contiene tutti gli altri elementi del documento;
* **<head>…</head>:** elemento che contiene informazioni sul documento;
* **<body>…</body>:** elemento che contiene il contenuto vero e proprio del documento.

**ELEMENTI BASE DEL BODY**

Vediamo alcuni elementi che si possono inserire nel body:

* **<h1>…</h1> fino a <h6>…</h6>:** servono a definire delle intestazioni. Un livello di intestazione maggiore (il numero all’interno dei tag) definisce un’intestazione di minore importanza.
* **<p>…</p>:** serve a definire un paragrafo. Può contenere solo testo ed elementi HTML inline;
* **<br>**:serve ad andare a capo (premendo semplicemente *invio*, infatti, non si va a capo);
* **<wbr>:** serve ad indicare dove in un testo andrebbe bene andare a capo (infatti, il testo occupa tutta la larghezza disponibile, e se una frase è troppo lunga, il browser la fa continuare alla riga successiva, provocando un effetto che non sempre è piacevole. Questo elemento serve dunque ad indicare al browser il punto migliore a partire dal quale far continuare la frase alla riga successiva).
* **<hr>**:serve a definire un cambiamento tematico tra un paragrafo e un altro;
* **<pre>…</pre>**: serve a definire un blocco di testo che verrà mostrato nel browser così come è scritto nel file HTML.
* **<blockquote>…</blockquote>:** serve a racchiudere del contenuto estratto da una fonte esterna;
* **<cite>…</cite>:** serve a indicare il titolo e/o l’autore di un contenuto esterno riportato nel documento;
* **<em>…</em>:** serve a racchiudere del testo da enfatizzare;
* **<strong>…</strong>:** serve a racchiudere del testo di grande importanza;
* **<small>…</small>:** serve a racchiudere del testo di poca importanza;
* **<dfn>…</dfn>:** serve a racchiudere un termine di cui verrà presto data la definizione;
* **<abbr>…</abbr>:** serve a racchiudere un termine che è l’abbreviazione di qualcos’altro.
  + L’attributo title = “*significato\_abbreviazione*” serve a specificare il significato dell’abbreviazione. Questo verrà visualizzato quando l’utente muove il cursore sopra l’abbreviazione.
* **<code>…</code>:** serve a racchiudere un frammento di codice;
* **<var>…</var>:** serve a racchiudere una variabile matematica;
* **<samp>…</samp>:** serve a racchiudere un esempio di output di un programma;
* **<kbd>…</kbd>:** serve a racchiudere un esempio di input da parte dell’utente;
* **<ul>…</ul>:** serve a definire una lista non ordinata;
* **<ol>…</ol>:** serve a definire una lista ordinata;
  + L’attributo type= *“tipo”* serve a definire il tipo di marker da usare nella lista.   
    Alcuni possibili valori di *tipo* sono:
    - 1: valore di default, verranno usati i numeri arabi;
    - a: verranno usate le lettere minuscole;
    - A: verranno usate le lettere maiuscole;
    - i: verranno usati i numeri romani, scritti in lettere minuscole;
    - I: verranno usati i numeri romani, scritti in lettere maiuscole;
  + L’attributo reversed, se presente, fa sì che i marker siano in ordine decrescente/  
    anti-alfabetico (di default, infatti, i marker sono in ordine crescente/alfabetico, dunque *es.* se la lista usa come marker i numeri arabi ed è costituita da 10 elementi, il primo avrà come marker “1”, il secondo “2”, il terzo “3”, …, il nono “9”, il decimo “10”. Mettendo l’attributo *reversed*, invece, succede che il primo elemento avrà come marker “10”, il secondo “9”, il terzo “8”, …, il nono “2”, il decimo “1”).
  + L’attributo start= *“valore”* serve ad impostare il valore del marker del primo elemento della lista. Gli elementi della lista successivi avranno come marker i valori a seguire.

*valore* può assumere solo valore intero, dunque se ad esempio la lista usa come marker le lettere maiuscole e si vuole che il primo elemento della lista abbia come marker “C”, va messo   
*start* = *“3”* (indicando così che si vuole come primo marker la *terza* lettera dell’alfabeto).

* **<li>…</li>:** elemento figlio di ul o ol, serve a definire un elemento della lista non ordinata o ordinata.
  + L’attributo value= *“valore”*, nel caso di lista ordinata, permette di impostare il valore del marker dell’elemento della lista. Gli elementi della lista successivi avranno come marker i valori a seguire.
* **<dl>…</dl>:** serve a definire una lista di definizioni, e cioè una lista di termini, ognuno con una descrizione.
* **<dt>…</dt>:** elemento figlio di dl, serve a definire un termine della lista di definizioni;
* **<dd>…</dd>:** elemento figlio di dt, serve a definire la descrizione del termine della lista di definizioni;
* **<ins>…</ins>:** serve a indicare un’aggiunta al documento;
  + L’attributo cite = “*URL\_spiegazione\_aggiunta*” serve ad indicare l’URL del documento che spiega il perché dell’aggiunta;
  + L’attributo datetime = “*timestamp\_aggiunta*” serve ad indicare il timestamp in cui è avvenuta l’aggiunta;
* **<del>…</del>:** serve a indicare una rimozione dal documento;
  + L’attributo cite = “*URL\_spiegazione\_rimozione*” serve ad indicare l’URL del documento che spiega il perché della rimozione;
  + L’attributo datetime = “*timestamp\_rimozione*” serve ad indicare il timestamp in cui è avvenuta la rimozione;
* **<a>…</a>:** serve a creare un link. Ogni link è costituito da due elementi:
  + L’attributo href *= “destinazione”* specifica la *destinazione* del link;
  + Tra i tag <a>, </a> va messo il *label* del link, e cioè ciò che, una volta cliccato, porta alla destinazione del link.

Si possono creare vari tipi di link:

* + Link che portano al documento HTML principale di un sito esterno:

<a href=“http://www.centralpark.com”>Central Park</a>

* + Link che portano ad una risorsa specifica di un sito esterno:

<a href=“http://www.centralpark.com/logo.gif”>Central Park</a>

* + Link che portano ad una risorsa specifica dello stesso sito del documento corrente:

<a href=“index.html”>Home</a>

* + Link che portano ad un punto specifico del documento corrente:

<a href=”#top”>Go to Top of Document</a>

* + Link che portano ad un punto specifico di un altro documento dello stesso sito del documento corrente:

<a href=“productX.html#reviews">Reviews for product X</a>

* + Link che, quando cliccato, fa andare in esecuzione il client di posta elettronica che suggerisce di inviare un’e-mail all’indirizzo indicato nella destinazione del link stesso:

<a href=”mailto:person@somewhere.com">Someone</a>

* + Link che, quando cliccato, fa andare in esecuzione la function Javascript indicata nella destinazione del link stesso:

<a href="javascript:OpenAnnoyingPopup();">See This</a>

* + Link che, quando cliccato, fa andare in esecuzione l’app Telefono con già digitato il numero indicato nella destinazione del link stesso:

<a href="tel:+18009220579">Call toll free (800) 922-0579</a>

Nell’indicare una risorsa di un sito esterno va usato il suo URL assoluto, composto da:

* + Protocollo usato (tipicamente, http://);
  + Nome di dominio;
  + Path della risorsa interessata (non necessario se si vuole indicare il documento HTML principale).

Nell’indicare una risorsa dello stesso sito del documento corrente, invece, è possibile (e conviene) usare il suo URL relativo, costituito solo dal path della risorsa interessata, che può essere espresso:

* + A partire da una certa directory base (di default, quella in cui si trova il documento corrente) se non si fa iniziare il path con ‘/’;
  + A partire dalla directory root del web server se si fa iniziare il path con ‘/’.

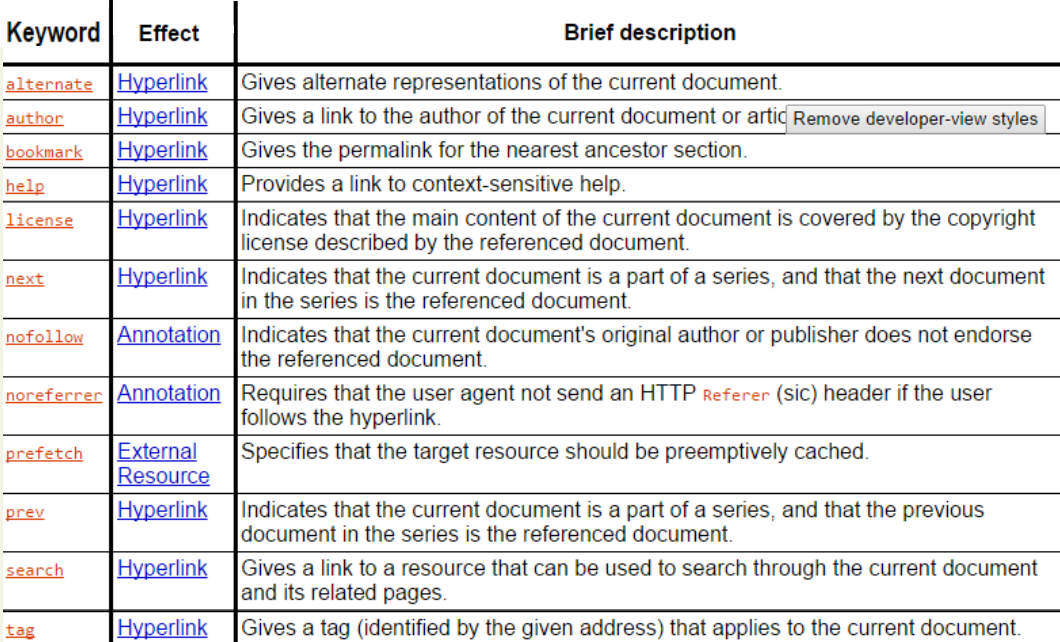
Vediamo ora altri attributi che può avere l’elemento *a*:

* + L’attributo target= “*contesto*” serve a specificare il contesto di navigazione in cui dovrà essere aperto il link. Alcuni possibili valori di *contesto* sono:

Immagine che contiene testo, numero, schermata, Carattere

Descrizione generata automaticamente

* + L’attributo rel= “*relazione*” serve a specificare la relazione tra il documento corrente e la destinazione del link. Alcuni possibili valori di *relazione* sono:



* + L’attributo download*,* se presente, serve a specificare che la destinazione del link deve essere scaricata nel dispositivo dell’utente quando questo clicca sul link.
* **<img>:** serve a includere un’immagine. Questo elemento va usato quando l’immagine rappresenta del contenuto (*es.* se è l’immagine di un prodotto) non quando è puramente decorativa (*es.* se è l’immagine di sfondo), perché per curare l’aspetto del contenuto, come già detto, si userà CSS.
  + L’attributo src = “*URL\_immagine*” serve a specificare l’URL dell’immagine da includere;
  + L’attributo alt = “*testo\_alternativo*” serve a specificare il testo alternativo dell’immagine.   
    E’ buona norma specificarlo sempre;
  + L’attributo title = “*titolo\_immagine*” serve a specificare il titolo dell’immagine. Questo verrà visualizzato quando l’utente muove il cursore sopra l’immagine;
  + Gli attributi width = “*larghezza\_immagine*” e height = “*altezza\_immagine*” servono a specificare rispettivamente la larghezza e l’altezza in pixel che l’immagine dovrà assumere nel documento;
* **<map>…</map>:** serve a definire una image map.
  + L’attributo name = “*nome\_mappa*” serve a specificare il nome della image map.   
    L’elemento img che userà la image map dovrà avere l’attributo usemap = “#*nome\_mappa*”;
* **<area>:** elemento figlio di map, serve a definire un’area della image map.
  + L’attributo alt = “*testo\_alternativo*” serve a specificare il testo alternativo dell’area;
  + L’attributo shape = “*forma\_area*” serve a specificare la forma dell’area. Alcuni possibili valori di *forma\_area* sono:
    - *rect*: l’area avrà forma rettangolare;
    - *circle:* l’area avrà forma circolare;
    - *poly:*l’area avrà forma poligonale.
  + L’attributo coords = “*coordinate\_area*” serve a specificare le coordinate dell’area.   
    Le *coordinate\_area* vanno espresse in pixel a partire dall’angolo in alto a sinistra dell’immagine che userà la image map, e vanno specificate in modo diverso a seconda della forma dell’area:
    - Se shape = “*rect*”, coords = “*top-left-x*, *top-left-y*, *bottom-right-x*, *bottom-right-y*”;
    - Se shape = “circle”, coords = “*center-x*, *center-y*, *radius*”;
    - Se shape = “poly”, coords = “*x1*, *y1*, *x2*, *y2*, …, *xN*, *yN*”.
* **<iframe>…</iframe>:** serve a definire un contesto di navigazione annidato.
  + L’attributo name = “*nome\_iframe*” serve a specificare il nome del contesto di navigazione annidato;
  + Ci sono due modi per specificare il documento HTML che dovrà essere visualizzato nel contesto di navigazione annidato:
    - Tramite l’attributo src = “*URL\_documento*”, che permette di specificare l’URL del documento HTML che dovrà essere visualizzato nel contesto di navigazione annidato;
    - Tramite l’attributo srcdoc = “*codice\_documento*”, che permette di specificare il codice del documento HTML che dovrà essere visualizzato nel contesto di navigazione annidato;
  + Gli attributi width = “*larghezza\_contesto*” e height = “*altezza\_contesto*” servono a specificare rispettivamente la larghezza e l’altezza in pixel del contesto di navigazione annidato;
  + Tra i tag <iframe>, </iframe> va messo il testo che si vuole venga visualizzato quando il browser non è in grado di caricare il contesto di navigazione annidato.
* **<object>…</object>:** serve a includere una risorsa esterna multimediale (tipicamente, qualcosa che fa uso di un qualche plug-in).
  + L’attributo data = “*URL\_risorsa*” serve a specificare l’URL della risorsa esterna multimediale da includere;
  + L’attributo type = “*tipo\_risorsa*” serve a specificare il tipo della risorsa esterna multimediale (leggere la documentazione per vedere i tipi possibili).
  + Gli attributi width = “*larghezza\_risorsa*” e height = “*altezza\_risorsa*” servono a specificare rispettivamente la larghezza e l’altezza in pixel che la risorsa esterna multimediale dovrà assumere nel documento;
  + Tra i tag <object>, </object> va messo il testo che si vuole venga visualizzato quando il browser non è in grado di caricare la risorsa esterna multimediale.
* **<video>…</video>:** serve a includere un video.
  + L’attributo src = “*URL\_video*” serve a specificare l’URL del video da includere;
  + L’attributo autoplay, se presente, serve a specificare che il video deve partire automaticamente non appena è pronto;
  + L’attributo loop, se presente, serve a specificare che il video, ogni volta che termina, deve ripartire automaticamente;
  + L’attributo controls, se presente, serve a specificare che devono essere visibili i controlli per gestire la riproduzione del video (bottone play, bottone pausa…);
  + Gli attributi width = “*larghezza\_video*” e height = “*altezza\_video*” servono a specificare rispettivamente la larghezza e l’altezza in pixel che il video dovrà assumere nel documento;
  + Tra i tag <video>, </video> va messo il testo che si vuole venga visualizzato quando il browser non è in grado di caricare il video.
* **<audio>…</audio>:** serve a includere un audio.
  + L’attributo src = “*URL\_audio*” serve a specificare l’URL dell’audio da includere;
  + L’attributo autoplay, se presente, serve a specificare che l’audio deve partire automaticamente non appena è pronto;
  + L’attributo loop, se presente, serve a specificare che l’audio, ogni volta che termina, deve ripartire automaticamente;
  + L’attributo controls, se presente, serve a specificare che devono essere visibili i controlli per gestire la riproduzione dell’audio (bottone play, bottone pausa…);
  + Tra i tag <audio>, </audio> va messo il testo che si vuole venga visualizzato quando il browser non è in grado di caricare l’audio.
* **<div>…</div>:** serve a definire un raggruppamento generico di contenuto;
* **<span>…</span>:** serve a definire un raggruppamento generico di contenuto (tipicamente,   
  solo testuale);
* **<aside>…</aside>:** serve a definire un raggruppamento di contenuto non essenziale che è correlato marginalmente al contenuto nelle vicinanze;
* **<figure>…</figure>:** serve a definire un raggruppamento di contenuto (non solo immagini) essenziale che può essere messo in qualsiasi posizione all’interno del documento e questo continua ad aver senso;
* **<figcaption>…</figcaption>:** elemento figlio di figure, serve ad aggiungere una didascalia alla figure;
* **<header>…</header>:** serve a definire un raggruppamento di contenuto introduttivo (dell’intero documento o solo di una parte di esso).
* **<footer>…</footer>:** serve a definire un raggruppamento di contenuto conclusivo (dell’intero documento o solo di una parte di esso).
* **<nav>…</nav>:** serve a definire un raggruppamento di link che portano ad altri documenti o ad altre parti dello stesso documento;
* **<main>…</main>:** serve a racchiudere il contenuto principale del documento, e cioè il contenuto che si trova solo in questo documento e in nessun’altro;
* **<section>…</section>:** serve a definire una sezione, e cioè un raggruppamento di contenuto che tratta dello stesso tema;
* **<article>…</article>:** serve a definire un articolo, e cioè un raggruppamento di contenuto indipendente dal resto del documento;

**NB:** la scelta degli elementi da inserire nel documento va fatta pensando alla struttura del contenuto desiderata, e non all’aspetto che il contenuto deve assumere (*es.* il livello di un’intestazione va scelto pensando all’importanza che si vuole dare all’intestazione nel documento, e non all’aspetto che l’intestazione deve assumere nel documento). Per definire poi l’aspetto del contenuto, come già detto, si userà CSS.

**ELEMENTI BASE DELL’HEAD**

Vediamo alcuni elementi che si possono inserire nell’head:

* **<meta>**: serve a indicare metadata riguardanti il documento.
  + L’attributo charset *= “codifica”* serve a indicare la codifica dei caratteri usata nel documento. E’ buona norma indicarlo sempre.
  + Scrivendo name = “author” content = “*nome\_autore*” si indica il nome dell’autore del documento;
  + Scrivendo name = “description” content = “*descrizione*” si indica la descrizione del documento;
  + Scrivendo name = “keywords” content = “*elenco\_keywords*” si indicano le keywords del documento (saranno usate dal motore di ricerca per l’indicizzazione).
* **<title>…</title>**: serve a indicare il titolo del documento. Questo verrà visualizzato nella tab del browser.
* **<base>**: serve a specificare la directory base che verrà usata per risolvere gli URL relativi che   
  non iniziano per ‘/’ presenti nel documento.
  + L’attributo href = “*URL\_assoluto\_directory\_base*” serve a specificare l’URL assoluto della directory base che verrà usata per risolvere gli URL relativi che non iniziano per ‘/’ presenti nel documento.
* **<link>…</link>**: serve a fare varie cose.
  + Scrivendo rel = “stylesheet” content = “*URL\_stylesheet\_CSS*”, si specifica lo stylesheet CSS da applicare al documento;
  + Scrivendo rel = “stylesheet” content = “*URL\_favicon*”, si specifica la favicon del documento.

**INTRODUZIONE AL CSS**

**CSS** è un linguaggio che permettedi specificare come deve apparire il contenuto.

In CSS, ogni elemento HTML viene visto come un box (riquadro) rettangolare, costituito da varie parti:

Immagine che contiene testo, schermata, linea, Carattere

Descrizione generata automaticamente

* **Element content area**: la zona in cui si trova il contenuto dell’elemento;
* **Padding**: uno spazio vuoto attorno al contenuto dell’elemento;
* **Border**: il bordo dell’elemento;
* **Margin**: uno spazio vuoto attorno al bordo dell’elemento, che lo distanzia dagli altri elementi.

Uno **stylesheet CSS** è costituito da una o più regole di stile.

Ogni **regola di stile** è costituita da:

* Un *selettore*, che seleziona gli elementi HTML di cui si vuole modificare l’aspetto;
* Un *blocco di dichiarazioni*, tra parentesi graffe. Ogni dichiarazione termina per ‘;’ ed ha una struttura *property* : *value*,dove:
  + *property* indica quale proprietà si vuole modificare degli elementi selezionati dal selettore;
  + *value* indica il modo in cui si vuole modificare tale proprietà.

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

I *value* vanno indicati in modo diverso a seconda della proprietà da modificare:

* I *value* che esprimono un colore possono essere indicati con:
  + Un nome (*es.* blue, yellow…)
  + Un codice RGB (*es.* rgb(255, 105, 0));
  + Un codice RGBa (*es.* rgb(255,0,0, 0.5));
  + Un codice esadecimale (*es*. #FF0000).
* I *value* che esprimono un URL vanno indicati con url(“*valore\_URL*”);
* I *value* che esprimono una dimensione vanno indicati con un numero seguito da un’unità di misura.   
  Alcune unità di misura sono:
  + *em*: relativa alla dimensione del font di un elemento;
  + *%:* relativa al valore di un’altra proprietà;
  + *in:* pollici;
  + *pt:* punti (1/72 di un pollice);
  + *px:* pixel (1/96 di un pollice).

L’unità di misura va scritta attaccata al numero (*es.* va scritto “50px”, non “50 px”).

Ci sono tre modi per dotare un documento HTML di codice CSS:

* **Inline**: si scrive il codice CSS all’interno dell’elemento di cui si vuole cambiare l’aspetto, inserendolo come valore dell’attributo style:

<h2 style=”font-size: 24pt”>Description</h2>

* **Embedded**: si scrive il codice CSS all’interno di un elemento style, da mettere dentro l’head:

<head>

…

<style>

h2 {

font-size: 24pt;

}

</style>

…

</head>

* **External**: si scrive il codice CSS in un file .css separato, e lo si applica al documento tramite un elemento <link>:

<head>

…

<link rel=”stylesheet” href=”styles.css>

…

</head>

**SELETTORI**

I selettori si dividono in varie categorie.

**TYPE SELECTORS**

Un **type selector** seleziona gli elementi di un certo tipo (e cioè, di un certo nome).

La sintassi è *nome\_elemento*

*es.*

p {

margin: 5px 0 10px 0;

Seleziono gli elementi <p>

font-weight: bold;

font-family: Arial, Helvetica, sans-serif;

}

**CLASS SELECTORS**

Un **class selector** seleziona gli elementi di una certa classe.

La sintassi è .*nome\_classe*

*es.*

Seleziono gli elementi di classe *second*

.second {

font-style: oblique;

color: yellow;

}

**ID SELECTORS**

Un **id selector** seleziona l’elemento che ha un certo id.

La sintassi è #*nome\_id*

*es.*

Seleziono l’elemento di id *negro*

#negro {

font-style: italic;

color: red;

}

**ATTRIBUTE SELECTORS**

Un **attribute selector** seleziona gli elementi sulla base della presenza di un certo attributo o sul valore di un certo attributo. La sintassi è:

* [*attribute*] seleziona gli elementi che presentano l’attributo *attribute*;
* [*attribute* = “*value*”] seleziona gli elementi che presentano l’attributo *attribute* con valore uguale a *value*;
* [*attribute* ^= “*value*”] seleziona gli elementi che presentano l’attributo *attribute* con valore che comincia per *value*;
* [*attribute* $= “*value*”] seleziona gli elementi che presentano l’attributo *attribute* con valore che termina per *value*;
* [attribute ~= “*value*”] seleziona gli elementi che presentano l’attributo *attribute* con valore che contiene la parola *value*;
* [attribute \*= “*value*”] seleziona gli elementi che presentano l’attributo *attribute* con valore che contiene la sottostringa *value*;

*es.*

[title] {

cursor: help;

padding-bottom: 3px;

Seleziono gli elementi che presentano l’attributo title

border-bottom: 2px dotted blue;

text-decoration: none;

}

**PSEUDO-ELEMENT SELECTORS**

Uno **pseudo-element** **selector** seleziona parti specifiche degli elementi. La sintassi è:

* ::first-line: seleziona la prima riga degli elementi;
* ::first-letter: seleziona la prima lettera degli elementi;
* ::before: serve a inserire qualcosa prima degli elementi;
* ::after: serve a inserire qualcosa dopo gli elementi;

**PSEUDO-CLASS SELECTORS**

Uno **pseudo-class** **selector** seleziona gli elementi che si trovano in un certo stato. La sintassi è:

* :link: seleziona gli elementi che non sono stati visitati (si applica ad elementi <area> ed <a>);
* :visited: seleziona gli elementi che sono stati visitati (si applica ad elementi <area> ed <a>);
* :hover: seleziona l’elemento che si trova sotto il cursore del mouse;
* :active: seleziona l’elemento *attivo*;
* :focus: seleziona l’elemento che ha il *focus*;
* :first-child: seleziona gli elementi che sono primi figli del loro padre;
* :nth-child(*n*): seleziona gli elementi che sono *n*-esimi figli del loro padre;
* :nth-of-type(*n*): seleziona gli elementi che sono *n*-esimi del loro tipo tra i loro fratelli.

**COMPOUND SELECTORS**

E’ possibile combinare gli effetti dei selettori delle 6 categorie precedenti, scrivendoli uno dopo l’altro tutti attaccati.

*es.*

Seleziono gli elementi <p> di classe *first*

p.first {

font-style: italic;

color: red;

}

Seleziono gli elementi <a> che presentano l’attributo hrefcon valore che comincia per *#* (e quindi, i link che portano ad un punto specifico del documento corrente)

a[href ^= "#"] {

background-color: gold;

}

a.second:hover {

Seleziono l’elemento <a> di classe *second* che si trova sotto il cursore del mouse.

text-decoration: none;

font-weight: bold;

}

**CONTEXTUAL SELECTORS**

Un **contextual selector** seleziona gli elementi sulla base dei loro antenati, discendenti o fratelli. La sintassi è:

* *selector1* *selector2* seleziona gli elementi selezionati da *selector2* che sono discendenti di un elemento selezionato da s*elector1*;
* *selector1* > *selector2* seleziona gli elementi selezionati da *selector2* che sono figli di un elemento selezionato da *selector1*;
* *selector1* + *selector2* seleziona gli elementi selezionati da *selector2* che sono immediati fratelli successivi di un elemento selezionato da *selector1*;
* *selector1* ~ *selector2* seleziona gli elementi selezionati da selector2 che sono fratelli successivi di un elemento selezionato da *selector1*.

*es.*

div p {

Seleziono gli elementi <p> che sono discendenti di un elemento <div>

color: blue;

}

#main>abbr {

Seleziono gli elementi <p> che sono figli di un elemento di id *main*

color: purple;

}

.myClass + p {

Seleziono gli elementi <p> che sono immediati fratelli successivi di un elemento di classe *myClass*

color: red;

}

Seleziono gli elementi <p> che sono fratelli successivi di un elemento <p> che è immediato fratello successivo di un elemento <h2>

Combinando più contextual selector, dunque, il selettore risultante va letto da destra verso sinistra.

h2 + p ~ p {

color: green;

}

**CONFLITTI TRA REGOLE**

Col cazzo che la faccio

**PROPRIETA’**

Vediamo alcune proprietà degli elementi HTML.

**PROPRIETA’ DEL BACKGROUND**

Il background riempie l’elemento fino al border escluso.   
Alcune proprietà del background sono:

* **background-color: *colore\_sfondo*** serve a specificare il colore del background;
* **background-image: *URL\_immagine*** serve a specificare l’immagine di background;
* **background-position: *posizione\_background*** serve a specificare la posizione di partenza dell’immagine di background. *posizione\_background* si può indicare con:
  + Una delle seguenti coppie di keyword:
    - left top
    - left center
    - left bottom
    - right top
    - right center
    - right bottom
    - center top
    - center center
    - center bottom
  + Due valori *x* *y*, dove:
    - *x* è la distanza dell’estremo sinistro del background dall’estremo sinistro dell’elemento;
    - *y* èla distanza dell’estremo superiore del background dall’estremo superiore dell’elemento;
* **background-attachment: *comportamento\_background*** serve a specificare come deve comportarsi l’immagine di background quando si scrolla il documento. Alcuni possibili valori di *comportamento\_background* sono:
  + scroll: l’immagine di background scrolla insieme al documento;
  + fixed: l’immagine di background rimane fissa mentre si scrolla il documento.
* **background-repeat: *ripetizione\_background*** serve a specificare se e come deve essere ripetuta l’immagine di background. Alcuni possibili valori di *ripetizione\_background* sono:
  + repeat: l’immagine di background viene ripetuta in orizzontale e in verticale;
  + repeat-x: l’immagine di background viene ripetuta in orizzontale
  + repeat-y: l’immagine di background viene ripetuta in verticale;
  + no-repeat: l’immagine di background non viene ripetuta.

**PROPRIETA’ DEL PADDING**

* **padding: *larghezza\_padding*** serve a specificare la larghezza di tutti e quattro i lati del padding.

E’ possibile specificare la larghezza di un singolo lato del padding modificando le proprietà   
padding-top, padding-right, padding-bottom, padding-left.

**PROPRIETA’ DEL BORDER**

* **border-style = *stile\_border*** serve a specificare lo stile di tutti e quattro i lati del border. Alcuni possibili valori di *stile\_border* sono:
  + solid: il border sarà una linea continua;
  + dotted: il border sarà a puntini;
  + dashed: il border sarà tratteggiato;

E’ possibile specificare lo stile dei singoli lati del border modificando le proprietà   
border-top-style, border-right-style, border-bottom-style, border-left-style.

* **border-width = *larghezza\_border*** serve a specificare la larghezza di tutti e quattro i lati del border.

E’ possibile specificare la larghezza dei singoli lati del border modificando le proprietà   
border-top-width, border-right-width, border-bottom-width, border-left-width

* **border-color = *colore\_border*** serve a specificare il colore di tutti e quattro i lati del border.

E’ possibile specificare il colore dei singoli lati del border modificando le proprietà   
border-top-color, border-right-color, border-bottom-color, border-left-color

* **border-radius = *raggio\_border*** serve a specificare il raggio di curvatura di tutti gli angoli del border.

E’ possibile specificare il raggio di curvatura dei singoli angoli del border modificando le proprietà border-top-right-radius, border-bottom-right-radius, border-bottom-left-radius, border-top-left-radius.

* **border-image = *URL\_immagine*** serve a specificare un’immagine da usare come border.

**PROPRIETA’ DEL PADDING**

* **padding: *larghezza\_padding*** serve a specificare la larghezza di tutti e quattro i lati del padding.

E’ possibile specificare la larghezza dei singoli lati del padding modificando le proprietà   
padding-top, padding-right, padding-bottom, padding-left.

**PROPRIETA’ DEL MARGIN**

* **margin: *larghezza\_margin*** serve a specificare la larghezza di tutti e quattro i lati del margin.

E’ possibile specificare la larghezza dei singoli lati del margin modificando le proprietà   
margin-top, margin-right, margin-bottom, margin-left.

Quando i margin verticali di due elementi si toccano, questi collassano in un unico margin, di dimensione pari al margin di massima dimensione.

I margins orizzontali, invece, non collassano mai.

**WIDTH E HEIGHT**

Le proprietà **width: *dimensione\_orizzontale\_content\_area*** e **height: *dimensione\_verticale\_content\_area*** servono a specificare la dimensione dell’element content area.

Per rendere il design del documento scalabile con la dimensione del browser, conviene specificare queste proprietà usando delle unità di misura relative. In particolare:

* Usando gli *em*, la dimensione è relativa alla dimensione del testo all’interno dell’elemento stesso;
* Usando le *%,* la dimensione è relativa alla dimensione dell’element content area dell’elemento padre;

**OVERFLOW**

La proprietà **overflow: *comportamento\_overflow*** serve a specificare il comportamento quando il contenuto è troppo grande per entrare nell’element content area. Alcuni possibili valori di *comportamento\_overflow* sono:

* visible: il contenuto in eccesso viene visualizzato all’esterno dell’element content area;
* hidden: il contenuto in eccesso non viene visualizzato;
* scroll: viene aggiunta una barra di scorrimento per visualizzare l’eventuale contenuto in eccesso;
* auto: viene aggiunta una barra di scorrimento solo se è presente del contenuto in eccesso;

**PROPRIETA’ DEL TESTO**

* **font-family: *nome\_font*** serve a specificare il font da usare. Se ne può specificare più di uno, separati da virgole, e verrà utilizzata la prima scelta possibile indicata;
* **font-size: *dimensione\_font*** serve a specificare la dimensione del font;
* **font-style: *stile\_font*** serve a specificare lo stile del font. Alcuni possibili valori di *stile\_font* sono:
  + normal: il testo verrà mostrato normale;
  + italic: il testo verrà mostrato in italic;
  + oblique: il testo verrà mostrato in obliquo;
* **font-weight: *boldness\_font*** serve a specificare la boldness del font. Alcuni possibili valori di *boldness\_font* sono:
  + normal: il testo verrà mostrato normale;
  + bold: il testo verrà mostrato in grassetto;
  + lighter: il testo verrà mostrato sottile.
* **color: *colore\_testo*** serve a specificare il colore del testo;
* **letter-spacing: *spaziatura\_caratteri*** serve a specificare la spaziatura orizzontale tra i caratteri;
* **line-height: *spaziatura\_righe*** serve a specificare la spaziatura verticale tra le righe;
* **text-align: *allineamento\_orizzontale\_testo*** serve a specificare l’allineamento orizzontale del testo. Alcuni possibili valori di *allineamento\_orizzontale\_testo sono:*
  + left: il testo verrà allineato a sinistra;
  + right: il testo verrà allineato a destra;
  + center: il testo verrà centrato;
  + justify: il testo verrà giustificato;
* **vertical-align: *allineamento\_verticale\_testo*** serve a specificare l’allineamento verticale del testo. Alcuni possibili valori di *allineamento\_verticale\_testo**sono:*
  + top: il testo verrà allineato in alto;
  + bottom: il testo verrà allineato in basso;
  + middle: il testo verrà centrato;
* **text-decoration: *decorazione\_testo*** serve a specificare la decorazione del testo. Alcuni possibili valori di *decorazione\_testo* sono:
  + underline: il testo verrà mostrato sottolineato;
  + overline: il testo verrà mostrato sopralineato;
  + line-through: il testo verrà mostrato sbarrato.

**TABELLE HTML**

Una tabella permette di rappresentare dati in formato tabellare.

Vediamo alcuni elementi necessari nella definizione di una tabella:

* **<table>…</table>:** serve a definire una tabella;
* **<tr>…</tr>:** elemento figlio di table, serve a definire una riga della tabella;
* **<th>…</th>:** elemento figlio di tr, serve a definire una cella della riga della tabella, contenente l’intestazione della colonna di cui tale cella fa parte.
  + L’attributo colspan = “*numero\_colonne*” serve a specificare il numero di colonne che la cella deve occupare;
  + L’attributo rowspan = “*numero\_righe*” serve a specificare il numero di righe che la cella deve occupare;
* **<td>…</td>:** elemento figlio di tr, serve a definire una cella della riga della tabella, contenente dei dati.
  + L’attributo colspan = “*numero\_colonne*” serve a specificare il numero di colonne che la cella deve occupare;
  + L’attributo rowspan = “*numero\_righe*” serve a specificare il numero di righe che la cella   
    deve occupare;

Esistono poi alcuni elementi semantici utili nella definizione di una tabella:

* **<caption>…</caption>:** elemento da mettere come primo figlio di table, serve ad aggiungere una didascalia alla tabella;
* **<col>…</col>:** elemento figlio di table da mettere subito dopo l’elemento caption, serve a definire un gruppo di una o più colonne della tabella.
  + L’attributo span = “*numero\_colonne*” serve a specificare il numero di colonne consecutive   
    di cui è composto il gruppo di colonne. Le colonne vengono selezionate da sinistra a destra, partendo dalla prima colonna non selezionata dal precedente elemento col (o dalla prima colonna della tabella, se non c’è un elemento col precedente).
* **<thead>…</thead>:** elemento figlio di table, serve a racchiudere le righe di intestazione della tabella;
* **<tbody>…</tbody>:** elemento figlio di table, serve a racchiudere le righe che costituiscono il corpo della tabella;
* **<tfooter>…</tfooter>:** elemento figlio di table, serve a racchiudere le righe che costituiscono la parte conclusiva della tabella.

Nel modificare lo stile di una tabella con CSS, può essere utile far sì che i border delle celle adiacenti   
collassino in unico border. Questo si può fare specificando la proprietà **border-collapse: collapse**   
per l’elemento table interessato.

**FORM HTML**

Un form permette all’utente di fornire dati in input al web server.

Vediamo alcuni elementi con cui è possibile creare un form:

* **<form>…</form>:** serve a definire un form.
  + L’attributo method = “*metodo”* serve a specificare come devono essere mandati i dati   
    al web server. Alcuni possibili valori di *metodo* sono:
    - GET: i dati vengono mandati tramite richiesta HTTP GET, appendendoli all’URL   
      in forma *name1*=*value1*&*name2*=*value2*&…&*namen*=v*aluen*;
    - POST: i dati vengono mandati tramite richiesta HTTP POST.
  + L’attributo action = “*URL\_pagina”* serve a specificare l’URL della pagina a cui devono essere mandati i dati.
* **<input>…</input>:** elemento figlio di form, serve a definire un campo di input.
  + L’attributo type = “*tipo”* serve a specificare il tipo di campo di input. Alcuni possibili valori di *tipo* sono:
    - text: serve a definire un campo in cui è possibile inserire una singola riga testuale;
    - password: serve a definire un campo in cui è possibile inserire una password;
    - search: serve a definire un campo in cui è possibile inserire una stringa di ricerca;
    - email: serve a definire un campo in cui è possibile inserire un indirizzo email;
    - tel: serve a definire un campo in cui è possibile inserire un numero di telefono;
    - url: serve a definire un campo in cui è possibile inserire un URL;
    - checkbox: serve a definire una checkbox. E’ possibile raggruppare più checkbox facendole condividere lo stesso valore per l’attributo *name*.
    - radio: serve a definire un radio button. E’ possibile raggruppare più radio button facendoli condividere lo stesso valore per l’attributo *name*, e questi saranno mutuamente esclusivi (e cioè, se ne può selezionare uno solo);
    - file: serve a definire un campo in cui è possibile caricare un file. Per poter usare questo tipo di campo di input, il form deve avere necessariamente gli attributi method= ”post” ed enctype= ”multipart/form-data”;
    - number: serve a definire un campo in cui è possibile inserire un numero.   
      Attraverso gli attributi min = “*valore\_minimo*” e max = “*valore\_massimo*” è possibile specificare il minimo e il massimo numero permesso;
    - datetime-local: serve a definire un campo in cui è possibile selezionare una data e   
      un orario;
    - date: serve a definire un campo in cui è possibile selezionare una data;
    - time: serve a definire un campo in cui è possibile selezionare un orario;
    - reset: serve a definire un bottone che, una volta premuto, riporta il form   
      al suo stato iniziale, cancellando tutti i dati inseriti dall’utente;
    - submit: serve a definire un bottone che, una volta premuto, invia al web server tutti i dati del form inseriti dall’utente.
* **<textarea>…</textarea>:** elemento figlio di form, serve a definire un campo di input in cui è possibile inserire più righe testuali;
* **<button>…</button>:** elemento figlio di form, serve a definire un bottone cliccabile;
* **<select>…</select>:** elemento figlio di form, serve a definire un menu di opzioni.
  + L’attributo multiple, se presente, fa sì che si possa selezionare più di un’opzione.
* **<option>…</option>:** elemento figlio di select, serve a definire un’opzione del menu di opzioni;
* **<label>…</label>:** elemento figlio di form, serve a definire un’etichetta per gli altri elementi figli di form. Per associare un certo label ad un certo elemento figlio di form, il valore dell’attributo *for* del label deve essere uguale al valore dell’attributo *id* dell’elemento figlio di form desiderato.

**CSS AVANZATO**

Vediamo ora un po’ di CSS avanzato.

**DA BLOCK-LEVEL A INLINE, E VICEVERSA**

Per far sì che un elemento block-level si comporti come un elemento inline, basta specificare la proprietà **display: inline** per l’elemento interessato.

Per far sì che un elemento inline si comporti come un elemento block-level, basta specificare la proprietà **display: block** per l’elemento interessato.

**POSIZIONARE GLI ELEMENTI**

Vediamo alcune proprietà che hanno a che fare con il posizionamento di un elemento nel documento.

* **position: *tipo\_posizionamento*** serve a specificare il tipo di posizionamento dell’elemento.   
  Alcuni possibili valori di *tipo\_posizionamento sono:*
  + static*:* l’elemento viene posizionato secondo il normale flusso del documento;
  + relative: l’elemento viene posizionato in relazione alla posizione che avrebbe occupato   
    nel normale flusso del documento. Lo spazio che l’elemento avrebbe occupato nel normale flusso del documento viene preservato;
  + absolute: l’elemento viene posizionato in relazione alla posizione del primo antenato posizionato (e cioè, per cui è stata specificata la proprietà position con un valore diverso da static). Lo spazio che l’elemento avrebbe occupato nel normale flusso del documento non viene preservato;
  + fixed: l’elemento viene posizionato in una specifica posizione e lì rimane, anche scrollando il documento.
* **top: *posizione\_verticale*** serve a specificare la posizione verticale di un elemento.
  + Se position: relative, *posizione\_verticale* rappresenta la distanza tra l’estremo superiore dell’elemento e l’estremo superiore della posizione che l’elemento avrebbe occupato   
    nel normale flusso del documento;
  + Se position: absolute o position: fixed, *posizione\_verticale* rappresenta la distanza   
    tra l’estremo superiore dell’elemento e l’estremo superiore del primo antenato posizionato.
* **left: *posizione\_orizzontale*** serve a specificare la posizione orizzontale di un elemento.
  + Se position: relative, *posizione\_orizzontale* rappresenta la distanza tra l’estremo sinistro dell’elemento e l’estremo sinistro della posizione che l’elemento avrebbe occupato   
    nel normale flusso del documento;
  + Se position: absolute o position: fixed, *posizione\_orizzontale* rappresenta la distanza   
    tra l’estremo sinistro dell’elemento e l’estremo sinistro del primo antenato posizionato.
* **z-index: *posizione\_z*** non l’ho capito, sai che esiste, poi se dovesse servire guardatelo;

**NASCONDERE GLI ELEMENTI**

Per nascondere un elemento senza preservare lo spazio che avrebbe occupato nel normale flusso del documento, basta specificare la proprietà **display: none** per l’elemento interessato.

Per nascondere un elemento preservando lo spazio che avrebbe occupato nel normale flusso del documento, basta specificare la proprietà **visibility: hidden** per l’elemento interessato.

**DESIGN MULTICOLONNA: FLEXBOX**

Flexbox permette di creare facilmente un design multicolonna.

Innanzitutto, gli elementi che si vogliono disporre a formare un design multicolonna – elementi   
che chiameremo *flex item* – devono essere figli di un elemento cui va specificata la proprietà display: flex   
– elemento che chiameremo *flex container* -.

Vediamo alcune proprietà del flex container:

* **gap: *dimensione\_gap*** serve a specificare la dimensione del gap tra i flex item;
* **flex-direction: *direzione\_principale*** serve a specificare quale tra la direzione orizzontale o la direzione verticale è la direzione principale, e cioè la direzione lungo la quale i flex item verranno disposti nel flex container. Alcuni possibili valori di *direzione\_principale* sono:
  + row: la direzione principale è quella orizzontale, dunque i flex item verranno disposti orizzontalmente, da sinistra verso destra;
  + column: la direzione principale è quella verticale, dunque i flex item verranno disposti verticalmente, dall’alto verso il basso.
* **flex-wrap: *dispozione\_multilinea*** serve a specificare se i flex item si devono disporre su una singola linea o se si possono disporre su più linee. Alcuni valori di *disposizione\_multilinea* sono:
  + nowrap: i flex item verranno disposti su una singola linea;
  + wrap: se necessario, i flex item verranno disposti su più linee.
* **justify-content: *disposizione\_item\_principale*** serve a specificare il modo in cui i flex item   
  verranno disposti nel flex container lungo la direzione principale. Alcuni possibili valori di *disposizione\_item\_principale*sono:
  + flex-start: i flex item verranno disposti all’inizio del flex container;
  + flex-end: i flex item verranno disposti alla fine del flex container;
  + center: i flex item verranno disposti al centro del container;
  + space-between: i flex item verranno disposti con dello spazio tra di essi;
  + space-around: i flex item verranno disposti con dello spazio prima, tra e dopo di essi;
* **align-content: *disposizione\_item\_secondaria*** serve a specificare il modo in cui i flex item   
  verranno disposti nel flex container lungo la direzione secondaria (e cioè, la direzione non principale). Alcuni possibili valori di *disposizione\_item\_secondaria*sono:
  + flex-start: i flex item verranno disposti all’inizio del flex container;
  + flex-end: i flex item verranno disposti alla fine del flex container;
  + center: i flex item verranno disposti al centro del container;
  + space-between: i flex item verranno disposti con dello spazio tra di essi;
  + space-around: i flex item verranno disposti con dello spazio prima, tra e dopo di essi;
* **align-items: *disposizione\_contenuto\_secondaria*** serve a specificare il modo in cui il contenuto di ogni flex item verrà disposto all’interno dell’item lungo la direzione secondaria. Alcuni valori possibili di *disposizione\_contenuto\_secondaria* sono flex-start, flex-end e center.

Vediamo alcune proprietà di un flex item:

* **align-self: *disposizione\_contenuto\_secondaria*** serve a specificare il modo in cui il contenuto   
  del particolare flex item verrà disposto all’interno dell’item lungo la direzione secondaria. Alcuni valori possibili di *disposizione\_item\_secondaria* sono flex-start, flex-end e center;
* **flex-grow: *fattore\_di\_crescita*** serve a specificare il fattore di crescita del flex item   
  lungo la direzione principale in relazione agli altri flex item;
* **flex-shrink: *fattore\_di\_restringimento*** serve a specificare il fattore di restringimento del flex item lungo la direzione principale in relazione agli altri flex item;
* **flex-basis: *dimensione\_principale\_default*** serve a specificare la dimensione di default dell’item lungo la direzione principale prima che lo spazio rimanente venga distribuito tra i vari item.

**DESIGN A GRIGLIA: GRID**

Grid permette di creare facilmente un design a griglia.

Innanzitutto, gli elementi che si vogliono disporre a formare un design a griglia – elementi   
che chiameremo *grid item* – devono essere figli di un elemento cui va specificata la proprietà display: grid   
– elemento che chiameremo *grid container* -.

Vediamo alcune proprietà del grid container:

* **gap: *dimensione\_gap*** serve a specificare la dimensione del gap tra i grid item;
* **grid-template-columns: *larghezza\_colonna1*, *larghezza\_colonna2*, …, *larghezza\_colonnaN*** serve a specificare il numero di colonne della griglia, specificando la larghezza di ognuna.   
  *larghezza\_colonna* può essere espresso in vari modi:
  + Attraverso una delle unità di misura già viste per esprimere una dimensione;
  + Attraverso l’unità di misura *fr*, che esprime una frazione della larghezza disponibile   
    nel grid container;
  + Attraverso il valore *auto*, che indica che la colonna occuperà la larghezza rimanente   
    nel grid container;
  + Attraverso la funzione *repeat*(), che permette di specificare pattern di colonne.   
    Alcuni modi di usarla sono i seguenti:
    - repeat(*m*, *larghezza\_colonna1*, *larghezza\_colonna2*, …, *larghezza\_colonnaP*): per *m* volte, aggiunge *p* colonne che hanno, nell’ordine specificato, la larghezza specificata;
    - repeat(auto-fill, minmax(*larghezza\_min*, *larghezza\_max*): riempie ogni riga di quanti più grid item riescono ad entrare nel grid container. Questi grid item devono avere almeno *larghezza\_min*, e possono avere al massimo *larghezza\_max*;
    - repeat(auto-fit, minmax(*larghezza\_min*, *larghezza\_max*): riempie ogni riga di quanti più grid item riescono ad entrare nel grid container, espandendo la larghezza degli item fino ad occupare tutta la larghezza disponibile nel grid container.   
      Questi container item devono avere almeno *larghezza\_min*, e possono avere al massimo *larghezza\_max*.
* **grid-template-rows: *altezza\_riga1*, *altezza\_riga2*, …, *altezza\_rigaN*** serve a specificare il numero di righe della griglia, specificando l’altezza di ognuna. *altezza\_riga* può essere espresso in vari modi:
  + Attraverso una delle unità di misura già viste per esprimere una dimensione;
  + Attraverso l’unità di misura *fr*, che esprime una frazione dell’altezza disponibile   
    nel grid container;
  + Attraverso il valore *auto*, che indica che la riga occuperà l’altezza rimanente   
    nel grid container;
  + Attraverso la funzione *repeat*(), che permette di specificare pattern di righe.   
    In questo caso, l’unico modo utile di usarla è il seguente:
    - repeat(*m*, *altezza\_riga1*, *altezza\_riga*, …, *altezza\_rigaP*): per *m* volte, aggiunge *p* righe che hanno, nell’ordine specificato, l’altezza specificata.
* **justify-content: *disposizione\_item\_orizzontale*** serve a specificare il modo in cui i grid item   
  verranno disposti nel grid container lungo la direzione orizzontale. Alcuni possibili valori di *disposizione\_item\_orizzontale*sono:
  + start: i grid item verranno disposti all’inizio del grid container;
  + end: i grid item verranno disposti alla fine del grid container;
  + center: i grid item verranno disposti al centro del grid container;
  + space-between: i grid item verranno disposti con dello spazio tra di essi;
  + space-around: i grid item verranno disposti con dello spazio prima, tra e dopo di essi;
* **align-content: *disposizione\_item\_verticale*** serve a specificare il modo in cui i grid item   
  verranno disposti nel grid container lungo la direzione verticale. Alcuni possibili valori di *disposizione\_item\_verticale*sono:
  + start: i grid item verranno disposti all’inizio del grid container;
  + end: i grid item verranno disposti alla fine del grid container;
  + center: i grid item verranno disposti al centro del grid container;
  + space-between: i grid item verranno disposti con dello spazio tra di essi;
  + space-around: i grid item verranno disposti con dello spazio prima, tra e dopo di essi;
* **justify-items: *disposizione\_contenuto\_orizzontale*** serve a specificare il modo in cui il contenuto   
  di ogni grid item verrà disposto all’interno dell’item lungo la direzione orizzontale.   
  Alcuni valori possibili di *disposizione\_contenuto\_orizzontale* sono start, end e center;
* **align-items: *disposizione\_contenuto\_verticale*** serve a specificare il modo in cui il contenuto   
  di ogni grid item verrà disposto all’interno dell’item lungo la direzione verticale.   
  Alcuni valori possibili di *disposizione\_contenuto\_verticale* sono start, end e center.

Vediamo alcune proprietà di un grid item:

* **justify-self: *disposizione\_contenuto\_orizzontale*** serve a specificare il modo in cui il contenuto   
  del particolare grid item verrà disposto all’interno dell’item lungo la direzione orizzontale.   
  Alcuni valori possibili di *disposizione\_contenuto\_orizzontale* sono start, end e center;
* **align-self: *disposizione\_contenuto\_verticale*** serve a specificare il modo in cui il contenuto   
  del particolare grid item verrà disposto all’interno dell’item lungo la direzione verticale.   
  Alcuni valori possibili di *disposizione\_contenuto\_verticale* sono start, end e center.

Di default, i grid item vengono disposti nel grid container riga per riga, nell’ordine in cui sono scritti.   
E’ possibile anche indicare esplicitamente in quale posizione inserire un certo grid item, in questo modo:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Per il grid container specifico la proprietà **grid-template-areas,** definendo come deve essere costituita la griglia e le aree che devono essere occupate da ogni item, assegnando ad ognuna di queste un certo nome.   
Per indicare un area che non deve essere occupata da alcun item si scrive ‘.’.   
Creo poi una classe per ogni item, e con la proprietà **grid-area** indico in quale area deve essere messo   
tale item.

E’ possibile poi specificare la larghezza di ogni colonna e l’altezza di ogni riga.

**COMBINARE GRID E FLEXBOX**

Se si vuole costruire un design a griglia:

* Grid è ideale per costruire l’effettivo design a griglia;
* Flexbox è ideale per disporre il contenuto all’interno di ogni grid item (per incolonnare gli elementi su una singola colonna, o per disporre gli elementi su una singola riga)

**INTRODUZIONE A JAVASCRIPT** *23-10-23*

**Javascript** è un linguaggio di programmazione che viene utilizzato per creare contenuti in modo dinamico.

Un **codice Javascript** è una sequenza di statement. Ogni statement termina per ‘;’.

Ci sono tre modi per dotare un documento HTML di codice Javascript:

* **Inline**: si scrive il codice Javascript all’interno dell’elemento interessato, come valore di un qualche attributo:

<a href=”JavaScript:OpenWindow();”>for more info</a>

<input type=”button” onClick=”alert(‘Are you sure?’);”>

* **Embedded**: si scrive il codice Javascript all’interno di un elemento <script>:

<script>

alert(“Hello world!”)

</script>

* **External**: si scrive il codice CSS in un file .js separato, e lo si include al documento tramite un elemento <script>, mettendo l’URL del file come valore dell’attributo src:

<script src=”greeting.js”></script>

Una variabile si dichiara con questa sintassi:

let *nome\_variabile*

Per inizializzare una variabile con un certo valore o per assegnarle un valore successivamente si usa l’operatore =. JavaScript è un linguaggio debolmente tipizzato: una stessa variabile può contenere valori di tipi diversi in istanti diversi e, in base alla situazione, un valore di un certo tipo viene convertito automaticamente in un valore di un altro tipo.

*Es.*

Dichiaro una variabile *abc* e la inizializzo al valore 3;

Dichiaro una variabile *foo*, senza inizializzarla ad alcun valore.

Assegno alla variabile *foo* il valore “prova”;

Assegno alla variabile *foo* il valore 4.

let abc = 3;

let foo;

foo = “prova”

foo = 4;

I tipi in JavaScript si dividono in due categorie:

* Tipi primitivi:
  + Numeric: un numero;
  + Booleano: *true* o *false*;
  + String: una sequenza di caratteri. Va scritta tra doppi o singoli apici;
  + Null: un unico valore, che è *null*;
  + Undefined: un unico valore, che è *undefined*. E’ il valore che viene assegnato alle variabili che non vengono inizializzate;
* Tipi riferimento:
  + Array: un array. Va scritto tra parentesi quadre, ed ogni elemento va separato da una virgola. Può essere multidimensionale, e può contenere elementi di tipo diverso.  
    Gli elementi sono indicizzati numericamente partendo dall’indice 0, ed è possibile accedere ad un elemento usando l’operatore [];
  + Date: un Date (vedremo cos’è);
  + Object: un oggetto (vedremo cos’è).

Dando un valore di tipo primitivo ad una variabile, questa prende come valore proprio il valore primitivo. Dando un valore di tipo riferimento ad una variabile, questa prende come valore l’indirizzo dell’area di memoria in cui tale valore di tipo riferimento viene memorizzato.

*Es.*

Dichiaro una variabile *abc* e la inizializzo al valore 27. Questa variabile, dunque, prenderà come valore proprio 27 (in quanto 27 è un valore di tipo numeric, primitivo).

Dichiaro una variabile *def* e la inizializzo al valore “hello”. Questa variabile, dunque, prenderà come valore proprio “hello” (in quanto “hello” è un valore di tipo string, primitivo).

Dichiaro una variabile *foo* e la inizializzo al valore [45, 35, 25]. Questa variabile, dunque, prenderà come valore l’indirizzo dell’area di memoria in cui viene memorizzato il valore [45, 35, 25] (in quanto [45, 35, 25] è un valore di tipo array, riferimento).

Dichiaro una variabile *xyz* e la inizializzo al valore di *def*, che è la stringa “hello”.

Dichiaro una variabile *bar* e la inizializzo al valore di *foo*, che è l’indirizzo dell’area di memoria contenente l’array [45, 35, 25]. Le due variabili, dunque, punteranno allo stesso array.

Assegno al primo elemento dell’array puntato da *bar* (che è lo stesso puntato da *foo*) il valore 200.

let abc = 27;

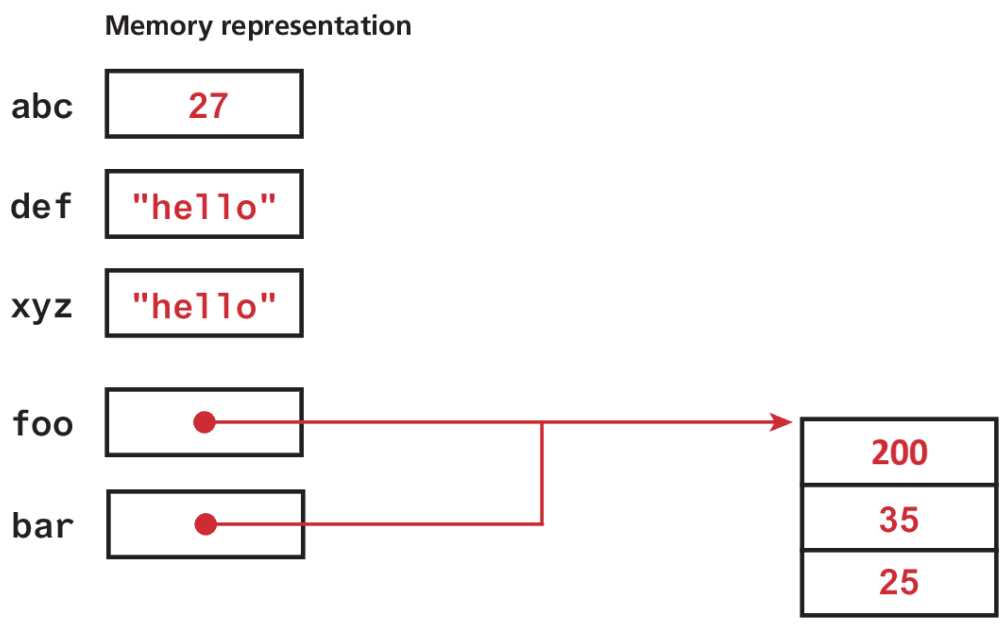
let def = “hello”;

let foo = [45, 35, 25];

let xyz = def;

let bar = foo;

bar[0] = 200;



Una variabile costante si dichiara con questa sintassi:

const *nome\_variabile*

Una variabile di tipo costante va necessariamente inizializzata, e non si può cambiare il suo valore successivamente.

*Es.*

Dichiaro una variabile costante *abc* e la inizializzo al numero 27. Essendo *abc* costante, non posso cambiare il suo valore successivamente.

Dichiaro una variabile costante *foo* e la inizializzo all’array [45, 35, 25], e quindi all’indirizzo dell’area di memoria in cui viene memorizzato tale array.   
Essendo *foo* costante, non posso cambiare il suo valore, e cioè non posso farla puntare a qualcos’altro. Questo, dunque, non mi impedisce di modificare gli elementi dell’array a cui *foo* punta.

const abc = 27;

//abc = 35;

const foo = [45, 35, 25];

foo[0] = 200;

Vediamo delle prime funzioni JavaScript utili:

* **alert(*message*)**: mostra una finestra di avviso con su scritto *message*
* **prompt(*message*):** mostra una finestra con su scritto *message* che richiede l’immissione di un dato da parte dell’utente. Restituisce ciò che l’utente ha immesso se l’utente clicca su “OK”, null se clicca su “Cancel”;
* **confirm(*message*):** mostra una finestra di conferma con su scritto *message*. Restituisce true se l’utente clicca su “OK”, false se clicca su “Cancel”;
* **document.write(*content*)**: stampa *content* (così com’è scritto) nel documento HTML, nel punto in cui la funzione è stata incontrata;
* **console.log(*content*):** stampa *content* nella console del browser.
* **typeof(*operand*):** restituisce una stringa indicante il tipo di *operand*;
* **eval(string):** valuta la stringa *string* e la esegue come se fosse codice Javascript;
* **isFinite(*value*):** restituisce true se *value* è un numero finito, false altrimenti;
* **isNaN(value):** restituisce true se *value* è NaN, false altrimenti.
* **parseInt(string):** restituisce il primo numero intero estratto dalla stringa *string.* Se il primo carattere di *string* non si può convertire in un intero, la funzione restituisce NaN;
* **parseFloat(string):** restituisce il primo numero decimale estratto dalla stringa *string.* Se il primo carattere di *string* non si può convertire in un intero, la funzione restituisce NaN.

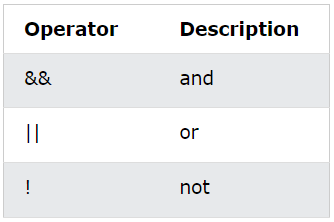
Javascript presenta if…else, switch, operatore ternario, while, do…while e for. La sintassi è la stessa del C.

Gli operatori di confronto possibili sono i seguenti:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Gli operatori logici possibili sono i seguenti:



In una condizione, tutti i valori vengono convertiti in true, ad eccezione di null, la stringa vuota, 0, NaN   
e undefined.

Le stringhe si possono concatenare tra loro usando l’operatore +:

*Es.*

Dichiaro una variabile *country* e la inizializzo alla stringa “France”.

Dichiaro una variabile *city* e la inizializzo alla stringa “Paris”.

Dichiaro una variabile *msg* e la inizializzo alla stringa ottenuta concatenando *city*, “ is the capital of “ e *country*. La variabile, dunque, avrà come valore “Paris is the capital of France”.

let country = “France”;

let city = “Paris”;

let msg = city + “ is the capital of “ + country;

Ogni stringa è dotata della proprietà *length*, che ha come valore la lunghezza della stringa.

E’ possibile confrontare due stringhe tra loro usando gli operatori di confronto, e una stringa sarà minore   
di un’altra se viene prima nel dizionario.

Ogni array è dotato della proprietà *length*, che ha come valore la lunghezza dell’array. Questo permette di scorrere senza problemi gli elementi di un array attraverso un ciclo for:

*es.*

for (let i = 0; i < years.length; i++) {

let yr = years[i];

console.log(yr);

}

Un modo alternativo di scorrere gli elementi di un array è attraverso un ciclo for…of:

*es.*

for (let yr of years) {

Ad ogni iterazione, in *yr* finirà il valore dell’elemento successivo   
dell’array *years*

console.log(yr);

}

In Javascript è possibile destrutturare un array in questo modo:

*es.*

Dichiaro tre variabili *x*, *y, z*, inizializzando la prima al primo elemento dell’array, la seconda al secondo elemento dell’array e la terza al terzo elemento dell’array.

E’ possibile saltare un elemento dell’array scrivendo semplicemente ‘,’. In questo modo, dichiaro tre variabili *a*, *b*, *c*, inizializzando la prima al primo elemento dell’array, la seconda al terzo elemento dell’array e la terza al quarto elemento dell’array.

Con la stessa sintassi, è possibile scambiare con facilità il valore   
di due variabili.

const league = [‘A’, ‘B’, ‘C’, ‘D’, ‘E’,];

let [x, y, z] = league;

let [a, ,b, c] = league;

[a, c] = [c, a]

E’ possibile estendere dinamicamente la dimensione di un array, ed è possibile evitare di specificare un valore per tutti gli elementi dell’array. Gli elementi di cui non si specifica un valore saranno undefined:

*es.*

let myCars = ["Saab", ,"BMW"];

Dichiaro un array *myCars*, specificando un valore per il primo e per il terzo elemento. Il secondo elemento, non avendo specificato il suo valore, sarà undefined.

Aggiungo all’array un elemento all’indice 6. Gli elementi di indice 3, 4, 5, non avendo specificato il loro valore, saranno undefined.

myCars[6] = “Fiat”;

E’ possibile espandere un array nei suoi singoli elementi usando l’operatore … :

*es.*

let pippo = [‘B’, ‘C’, ‘D’];

Dichiaro un array *paperino* che ha come elementi, nell’ordine, ‘A’, una copia degli elementi dell’array *pippo*, ed ‘E’;

let paperino = [‘A’, …pippo, ‘E’];

};

Una funzione si dichiara in questo modo:

function *nome\_funzione*(*lista\_parametri*){

…

}

Non va specificato dunque né il tipo di ritorno della funzione, né il tipo dei parametri d’ingresso.

La sintassi per ritornare il controllo al chiamante e, eventualmente, ritornare un valore, così come la sintassi per invocare una funzione, è la stessa del C.

*es.*

function subtotal(price, quantity) {

return price \* quantity;

}

let result = subtotal(10,2);

E’ possibile assegnare dei valori di default ai parametri, che verranno usati quando per tali parametri non viene passato alcun valore:

function foo(a=10, b=0) {

return a+b;

}

E’ possibile dichiarare una funzione che accetta un numero variabile di argomenti usando l’operatore … :

*es.*

function concatenate(...args) {

let s = "";

In questo modo, la funzione può prendere in ingresso un numero indeterminato di parametri.

for (let a of args) {

s += a + " ";

}

return s;

}

Un **oggetto** è una collezione di coppie chiave-valore, dette proprietà. Un modo di creare un oggetto è attraverso un object literal:

let *objName* = {

Si indicano dunque tra parentesi graffe, in forma *name*: *value*, le proprietà dell’oggetto con il relativo valore, separate da virgola.

Una proprietà può assumere un valore di qualsiasi tipo, anche di tipo oggetto.

*name1*: *value1*,

*name2*: *value2*,

// ...

*nameN*: *valueN*

};

*Es.*

const stud1 = {

Creo un oggetto *stud1* che ha come proprietà:

* *nome* di valore “Mario”;
* *cognome* di valore “Rossi”;
* *matricola* di valore 1234;

Creo un oggetto o1 che ha come proprietà:

* *propr1* di valore v1;
* *propr2* di valore v2.

Se non si specifica il nome della proprietà, viene usato il nome della variabile.   
In questo modo, dunque, dichiaro un oggetto *o1bis* che ha come proprietà:

* *v1* di valore v1;
* *v2* di valore v2.

E’ possibile accedere alle proprietà di un oggetto usando l’operatore *.propertyName.*

Un altro modo è usando l’operatore [“*propertyName*”].

E’ possibile aggiungere una nuova proprietà ad un oggetto semplicemente assegnando un valore alla nuova proprietà.

nome: “Mario”,

cognome: “Rossi”,

matricola: 1234

};

let v1 = 22;

let v2 = “ABC”;

const o1 = {

propr1 : v1,

propr2: v2,

};

const o1bis = {

v1,

v2

};

console.log(o1.propr1);

console.log(o1bis[“v1”]);

o1.propr3 = 44;

E’ possibile destrutturare un oggetto in questo modo:

*es.*

const photo = {

id: 1,

title: “Benito”,

location: {

country: “Italy”,

city: “Predappio”

}

Dichiaro due variabili *id*, *title,* inizializzando la prima alla prima proprietà dell’oggetto e la seconda alla seconda proprietà dell’oggetto.

};

let {id, title} = photo;

E’ possibile espandere un oggetto nelle sue singole proprietà usando l’operatore … :

*es.*

Creo un oggetto *foo* che ha come proprietà*:*

* *name* di valore “Bob”;
* Una copia delle proprietà e dei valori dell’oggetto *location* dell’oggetto *photo*, dunque:
  + *country* di valore “Italy”;
  + *city* di valore “Predappio”;
* *iso* di valore “Ca”.

const foo = {

name: “Bob”,

…photo.location,

iso: ”Ca”

};

E’ possibile dotare un oggetto di proprietà che sono funzioni:

*es.*

const order = {

La proprietà *product* dell’oggetto *order* è dotata di una proprietà *output*, che è una funzione anonima.

Invoco poi la funzione normalmente, come se stessi accedendo ad una proprietà dell’oggetto *order.product*.

La parola chiave *this* è un riferimento all’oggetto per cui si chiama la funzione (dunque *this.brand* e *this.price* sono riferimenti alle proprietà *brand* e *price* dell’oggetto *product* per cui si chiama la funzione)

salesDate : "May 5, 2016",

product : {

price: 500.00,

brand: "Acer",

output: function () {

return this.brand + ' $' + this.price;

}

};

console.log(order.product.output());

E’ possibile scorrere le proprietà di un oggetto attraverso un ciclo for…in:

*es.*

const studente = {

nome: "Mario",

cognome: "Rossi",

Ad ogni iterazione, in *p* finirà una stringa rappresentante il nome della prossima proprietà dell’oggetto *studente*.

Uso poi il nome della proprietà per accedere al suo valore.

matricola: 1234,

voti: [23, 28, 29]

};

for(let p in studente) {

console.log(p + ' di tipo ' + typeof(p));

console.log('con valore di tipo ' + typeof(studente.p) + ': ' + studente.p);

}

Un altro modo di creare un oggetto è attraverso una funzione costruttrice, attraverso la quale è possibile definire un “tipo di oggetto”: si definisce una funzione in cui si specificano le proprietà e i metodi che gli oggetti di tale tipo dovranno avere. A questo punto, invocando la funzione usando l’operatore new, verrà creato un oggetto con le proprietà e i metodi specificati, e verrà restituito un riferimento all’oggetto creato:

Es.

function Customer(name, address, city) {

Creo una funzione costruttrice per oggetti di tipo *Customer*, che dovranno avere le proprietà e i metodi specificati.

this.name = name;

this.address = address;

this.city = city;

this.output = function () {

return this.name + " " + this.address + " " + this.city;

}

}

const cust1 = new Customer("Sue", "123 Somewhere", "Calgary");

Creo due oggetti di tipo *Customer*, con i valori delle proprietà passati come parametri.

alert(cust1.output());

const cust2 = new Customer("Fred", "32 Nowhere St", "Seattle");

alert(cust2.output());

**ARRAY IN DETTAGLIO**

Su ogni *array* è possibile chiamare i seguenti metodi:

* ***array.*concat(*array2, array3, …*, *arrayN*):** restituisce l’array ottenuto concatenando *array* con *array2*, *array3*, …, *arrayN*;
* ***array*.join(*separator*):** restituisce una stringa contenente gli elementi di *array*, separati tra loro da *separator*. Omettendo *separator*, gli elementi saranno separati da ‘,’;
* ***array.*shift():** rimuove il primo elemento da *array* e restituisce l’elemento rimosso;
* ***array.*unshift(*element1, element2, …, elementN*):** inserisce *element1, element2, …, elementN*all’inizio di *array*, e restituisce la nuova lunghezza di *array*;
* ***array.*pop():** rimuove l’ultimo elemento da *array* e restituisce l’elemento rimosso;
* ***array.*push(*element1, element2, …, elementN*):** inserisce *element1, element2, …, elementN*alla fine di *array*, e restituisce la nuova lunghezza di *array*;
* ***array.*reverse():** inverte l’ordine degli elementi di *array*;
* ***array.*slice(*start*, *end*):** restituisce l’array ottenuto da *array* prendendo gli elementi dall’indice *start* incluso all’indice *end* escluso. Omettendo *end*, verranno presi gli elementi fino alla fine di *array;*
* ***array.*splic*e*(*index, howmany, element1, element2, …, elementN*):** aggiunge e/o rimuove elementi   
  da *array*:
  + *index* è l’indice a partire dal quale aggiungere e/o rimuovere elementi;
  + *howmany* è il numero di elementi da rimuovere. Se non ci sono elementi da rimuovere, va passato 0;
  + *element1, element2, …, elementN* sono gli elementi da aggiungere. Se non ci sono elementi da aggiungere, non va passato nulla.

Il metodo prima rimuove *howmany* elementi di *array* a partire dall’indice *index*, e poi aggiunge   
gli elementi *element1*, *element2*, *elementN* a partire dall’indice *index*, spostando agli indici successivi gli elementi che si trovavano a partire dall’indice *index* in poi;

* ***array.*sort(*sortfunc*):** ordina *array* attraverso la funzione *sortfunc*. *sortfunc* deve prendere in ingresso due parametri *a* e *b,* verrà chiamata per ogni coppia di elementi di *array*, e a seconda se restituisce un valore < 0 o > 0, si avrà che l’elemento passato come *a* verrà messo in *array* rispettivamente prima o dopo l’elemento passato come *b*. Omettendo *sortfunc,* gli elementi di *array* vengono ordinati alfabeticamente;
* ***array.*forEach(*func*):** invoca la funzione *func* per ogni elemento di *array*. *func* deve prendere in ingresso un parametro *a*, che conterrà l’elemento corrente di *array*;
* ***array.*find(*func*):** restituisce il primo elemento di *array* che soddisfa la funzione *func*. *func* deve prendere in ingresso un parametro *a*, che conterrà l’elemento corrente di *array*, e questo elemento verrà restituito dal metodo se e solo se *func* restituisce true. Se *func* restituisce false per ogni elemento di *array*, il metodo restituisce undefined;
* ***array.*findIndex(*func*):** restituisce l’indice del primo elemento di *array* che soddisfa la funzione *func*. *func* deve prendere in ingresso un parametro *a*, che conterrà l’elemento corrente di *array*, e verrà restituito l’indice di questo elemento se e solo se *func* restituisce true. Se *func* restituisce false per ogni elemento di *array*, il metodo restituisce -1;
* ***array.*filter(*func*):** restituisce l’array ottenuto da *array* prendendo gli elementi che soddisfano *func*.   
  *func* deve prendere in ingresso un parametro *a*, che conterrà l’elemento corrente di *array*, e questo elemento verrà inserito in fondo al nuovo array se e solo se *func* restituisce true. Se *func* restituisce false per ogni elemento di *array*, il metodo restituisce un array vuoto;
* ***array.*map(*func*):** restituisce l’array ottenuto invocando *array* su ogni elemento di *func* e inserendo il risultato restituito da *func* in fondo al nuovo array. *func* deve prendere in ingresso un parametro *a*, che conterrà l’elemento corrente di *array*.
* ***array*.fill(*value*):** riempie *array* per tutta la sua *length* con *value*.

**BUILT-IN OBJECT BOOLEAN**

Javascript mette a disposizione il built-in object **Boolean**, che offre un metodo utile:

* **Boolean(v)**: restituisce la conversione in booleano di *v*;

**DATE IN DETTAGLIO**

Un **Date** è un particolare tipo di oggetto che memorizza una data e un orario;

Ci sono 4 modi per creare un Date:

* **new Date():** crea un Date contenente la data e l’orario corrente;
* **new Date(*milliseconds*):** crea un Date contenente la data e l’orario che distano *milliseconds* millisecondi dal 01/01/1970;
* **new Date(*dateString*):** crea un Date contenente la data e l’orario indicati in *dateString,* una stringa in formato “*month* *day*, *year* *hours*:*minutes*:*seconds*”;
* **new Date(year, month, day, hours, minutes, seconds, milliseconds):** crea un Date contenente la data e l’orario indicati attraverso i parametri, passati come numeri.

Su ogni *Date* è possibile chiamare i seguenti metodi:

* **Date.getTime():** restituisce il numero di millisecondi del *Date* dal 01/01/1970;
* ***Date.*getFullYear():** restituisce l’anno del *Date*, come numero su 4 cifre;
* **Date.getMonth():** restituisce il mese del *Date*, come numero tra 0 e 11;
* ***Date.*getDate():** restituisce il giorno del mese del *Date,* comenumero tra 1 e 31;
* ***Date.*getDay():** restituisce il giorno della settimana del *Date*, come numero tra 0 e 6;
* ***Date.*getHours():** restituisce l’ora del *Date*, come numero tra 0 e 23;
* ***Date.*getMinutes():** restituisce i minuti del *Date*, come numero tra 0 e 59;
* **Date.getSeconds():** restituisce i secondi del *Date*, come numero tra 0 e 59;
* ***Date.*getMilliseconds():** restituisce i millisecondi del *Date*, come numero tra 0 e 999;
* ***Date.*setTime(*milliseconds*):** imposta la data e l’orario contenuti nel *Date* a quelli che distano *milliseconds* millisecondi dal 01/01/1970;
* ***Date.*setFullYear(*year*):** imposta l’anno del *Date* a *year,* indicato come numero su 4 cifre;
* ***Date.*setMonth(*month*):** imposta il mese del *Date* a *month*, indicato come numero tra 0 e11;
* ***Date*.setDate(*day*):** imposta il giorno del mese del *Date* a *day*, indicato come numero tra 1 e 31;
* ***Date.*setHours(*hour*):** imposta l’ora del *Date* a *hour*, indicato come numero tra 0 e 23;
* ***Date.*setMinutes(*minutes*):** imposta i minuti del *Date* a *minutes*, indicati come numero tra 0 e 59;
* ***Date*.setSeconds(*seconds*):** imposta i secondi del *Date* a *seconds*, indicati come numero tra 0 e 59;
* ***Date.*setMilliseconds(*milliseconds*):** imposta i millisecondi del *Date* a *milliseconds*, indicati come numero tra 0 e 999;
* ***Date.*toDateString(), *Date*.toLocaleDateString(), *Date*.toLocaleTimeString(), *Date*.toLocaleString(), *Date*.toString(), *Date*.toTimeString(), *Date*.toUTCString():** restituiscono una stringa rappresentante la data e l’orario del *Date*, formattata in modo diverso a seconda del metodo chiamato.

**BUILT-IN OBJECT MATH**

Javascript mette a disposizione il built-in object **Math**, che offre una serie di proprietà e metodi utili:

* **Math.E**: restituisce (il numero di Eulero);
* **Math**.**LN2:** restituisce ;
* **Math**.**LN10:** restituisce ;
* **Math.LOG2E:** restituisce ;
* **Math.LOG10E:** restituisce ;
* **Math.PI:** restituisce ;
* **Math.SQRT2**: restituisce ;
* **Math.max(*num1*, *num2*, …, *numN*):** restituisce il *num* di massimo valore;
* **Math.min(*num1*, *num2*, …, *numN*):** restituisce il *num* di minimo valore;
* **Math.random():** restituisce un numero casuale tra 0 incluso e 1 escluso;
* **Math**.**abs(*x*):** restituisce ;
* **Math**.**ceil(*x*):** restituisce ;
* **Math**.**floor(*x*):** restituisce ;
* **Math**.**round(*x*):** arrotonda all’intero più vicino;
* **Math.exp(*x*):** restituisce ;
* **Math**.**pow(*x*, *y*):** restituisce ;
* **Math.log(*x*):** restituisce
* **Math**.**sin(*x*):** restituisce , con espresso in radianti;
* **Math**.**cos(*x*):** restituisce , con espresso in radianti;
* **Math**.**tan(*x*):** restituisce , con espresso in radianti;

**BUILT-IN OBJECT NUMBER E NUMBER IN DETTAGLIO**

Javascript mette a disposizione il built-in object **Number**, che offre una serie di proprietà e metodi utili:

* **Number.MAX\_VALUE:** restituisce il più grande numero rappresentabile in Javascript;
* **Number.MIN\_VALUE:** restituisce il più piccolo numero rappresentabile in Javascript;
* **Number.NEGATIVE\_INFINITY**: restituisce -Infinity;
* **Number.POSITIVE\_INFINITY**: restituisce +Infinity;
* **Number(*v*):** restituisce la conversione in numero di *v*. Restituisce NaN se non è possibile convertire *v* in numero.

Su ogni *number* è possibile chiamare i seguenti metodi:

* ***number.*toExponential(*x*):** restituisce una stringacontenente la rappresentazione di *number* in notazione esponenziale con una mantissa di *x* cifre. Omettendo *x*, viene usato il numero di cifre necessario a rappresentare *number*;
* ***number.*toFixed(*x*):** restituisce una stringa contenente la rappresentazione di *number* in notazione decimale con *x* cifre decimali. Omettendo *x*, non viene usata alcuna cifra decimale;
* ***number.*toPrecision(*x*):** restituisce una stringa contenente la rappresentazione di *number* in notazione decimale con *x* cifre. Omettendo *x*, viene usato il numero di cifre necessario a rappresentare *number*.

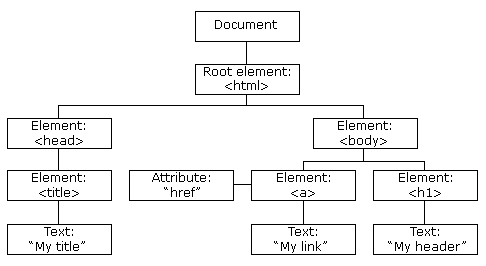
**STRING IN DETTAGLIO**

Su ogni *string* è possibile chiamare i seguenti metodi:

* ***string.*charAt(*index*):** restituisce il carattere che si trova all’indice *index*;
* ***string.*concat(*string2, string3, …*, *stringN*):** restituisce la stringa ottenuta concatenando *string*con *string2*, *string3*, …, *stringN*;
* ***string.*indexOf(*searchstring*, *start*):** restituisce l’indice della prima occorrenza della sottostringa *searchstring* in *string*, analizzando *string* a partire dall’indice *start*. Omettendo *start*, *string* viene analizzata partendo dall’indice 0. Restituisce -1 se non viene trovata alcuna occorrenza di *searchstring* in *string*;
* ***string.*lastIndexOf(*searchstring*, *start*):** restituisce l’indice dell’ultima occorrenza della sottostringa *searchstring* in *string*, analizzando *string* a partire dall’indice *start*. Omettendo *start*, *string* viene analizzata partendo dall’indice 0. Restituisce -1 se non viene trovata alcuna occorrenza di *searchstring* in *string*;
* ***string.*replace(*oldstring, newstring*):** restituisce la stringa ottenuta da *string* sostituendo la prima occorrenza della sottostringa *oldstring*con la sottostringa *newstring***;**
* ***string.*replaceAll(*oldstring, newstring*):** restituisce la stringa ottenuta da *string* sostituendo ogni occorrenza della sottostringa *oldstring*con la sottostringa *newstring***;**
* ***string.*slice(*start*, *end*):** restituisce la stringa ottenuta da *string* prendendo i caratteri dall’indice *start* incluso all’indice *end* escluso. Omettendo *end*, verranno presi i caratteri fino alla fine di *string;*
* ***string.*split(*separator*, *limit*):** restituisce un array ottenuto splittando *string* ogni volta che viene trovato il carattere *separator*, effettuando al massimo *limit* split. Una volta effettuati *limit* split, se non si è ancora arrivati alla fine di *string*, i caratteri rimanenti vengono esclusi dall’array.   
  Omettendo *limit*, non ci sarà alcun numero massimo di split;
* ***string.*toLowerCase():** restituisce la stringa ottenuta da *string* convertendo tutti i caratteri   
  in minuscolo;
* ***string.*toUpperCase():** restituisce la stringa ottenuta da *string* convertendo tutti i caratteri   
  in maiuscolo.

**DOM (DOCUMENT OBJECT MODEL)**

Il **DOM** è una struttura ad albero che rappresenta come è organizzato il documento HTML.



Ogni elemento HTML, ogni testo e ogni attributo è rappresentato come un **nodo**, che ha come figli   
gli elementi figli e i testi di cui è composto. Chiameremo:

* Element node: un nodo che rappresenta un elemento HTML;
* Attribute node: un nodo che rappresenta un attributo;
* Text node: un nodo che rappresenta un testo.

Alla radice di ogni DOM si ha il nodo *document*, che rappresenta l’intero documento HTML. Questo ha   
come nodo figlio il nodo *html*, che rappresenta l’elemento html, e da questo si diramano tutti i vari nodi.

Su Javascript, ogni nodo è rappresentato come un oggetto di tipo *node*. Esiste anche il tipo di oggetto *NodeList*, che può essere visto come un array contenente riferimenti ad un insieme di nodi.

Vediamo alcune proprietà comuni ad ogni *node*:

* ***node*.childNodes**: restituisce una NodeList dei nodi figli di *node*;
* ***node*.parentNode:** restituisce un riferimento al nodo padre di *node*;
* ***node*.firstChild:** restituisce un riferimento al primo nodo figlio di *node*;
* ***node*.firstElementChild():** restituisce un riferimento al primo element node figlio di *node*;
* ***node*.lastChild**: restituisce un riferimento all’ultimo nodo figlio di *node*;
* ***node*.lastElementChild():** restituisce un riferimento all’ultimo element node figlio di *node*;
* ***node*.previousSibling:** restituisce un riferimento al nodo fratello immediatamente precedente a *node*
* ***node*.nextSibling:** restituisce un riferimento al nodo fratello immediatamente successivo a *node*;
* ***node*.nodeName:** restituisce una stringa contenente il nome di *node*. In particolare, restituisce:
  + il nome in maiuscolo dell’elemento che *node* rappresenta, se *node* è un element node;
  + il nome dell’attributo che *node* rappresenta, se *node* è un attribute node;
  + #text, se *node* è un text node;
* ***node*.nodeType:** restituisce un numero indicante il tipo di *node*. In particolare, restituisce:
  + 1 se *node* è un element node;
  + 2 se *node* è un attribute node;
  + 3 se *node* è un text node.
* ***node*.nodeValue**: restituisce il valore di *node*. In particolare, restituisce:
  + null se *node* è un element node;
  + il valore dell’attributo se *node* è un attribute node;
  + il testo se *node* è un text node.

Vediamo alcune funzioni che permettono di selezionare uno o più element node:

* **document.getElementById("*id*"):** restituisce un riferimento al nodo che ha l’*id* indicato;
* **document.getElementsByClassName("*class*"):** restituisce una NodeList dei nodi che appartengono alla *class* indicata;
* **document.getElementsByTagName(“*TagName*”):** restituisce una NodeList dei nodi che hanno il   
  *TagName* indicato;
* **document.querySelector(“*selector\_CSS*”):** restituisce un riferimento al primo nodo selezionato da *selector\_CSS*;
* **document.querySelectorAll(“*selector\_CSS*”)**: restituisce una NodeList dei nodi selezionati da *selector\_CSS*.

Vediamo alcune proprietà degli *element\_node* che, se modificate, permettono di modificare l’*element* che *element\_node* rappresenta:

* ***element\_node*.classList**:un particolare tipo di oggetto indicante le classi cui *element* appartiene.   
  Ci sono alcune funzioni per manipolare *element\_node.*classList*,* cambiando così le classi   
  cui *element* appartiene:
  + ***element\_node.*classList.add(“*class*”):** aggiunge *class* alle classi cui *element* appartiene;
  + ***element\_node.*classList.remove(“*class*”):** rimuove *class* dalle classi cui *element* appartiene;
  + ***element\_node*.classList.toggle(“*class*”):** aggiunge *class* alle classi cui *element* appartiene se *class* non è tra le classi cui element appartiene, altrimenti la rimuove;
* **element\_*node*.style:** un particolare tipo di oggetto che ha, come proprietà, le proprietà CSS di *element* con il loro valore;
* ***element\_node*.id:** contiene una stringa contenente il valore dell’attributo id di *element*;
* ***element\_node*.href:** contiene una stringa contenente il valore dell’attributo href di *element*;
* ***element\_node*.name:** contiene una stringa contenente il valore dell’attributo name di *element*;
* ***element\_node*.src:** contiene una stringa contenente il valore dell’attributo src di *element*;
* ***element\_node*.value:** contiene una stringa contenente il valore dell’attributo value di *element*;
* ***element\_node.*checked:** significativo se *element* è una checkbox o un radio button, contiene true   
  se *element* attualmente è selezionato, false altrimenti;
* ***element\_node.*selected:** significativo se *element* è un’option contenuta in un select, contiene true   
  se *element* attualmente è selezionato, false altrimenti.
* ***element\_node.*disabled:** contiene true se *element* presenta l’attributo disabled, false altrimenti;
* ***element\_node*.innerHTML:** contiene tutto il contenuto HTML di *element*.
* ***element\_node.*textContent:** contiene tutto il testo contenuto in *element* (privato di eventuali tag).

Vediamo alcuni metodi che permettono di modificare il DOM:

* **document.createElement(*element\_name*):** crea un nuovo element node che rappresenta un elemento *element\_name*;
* **document.createTextNode(*text*):** crea un nuovo text node che rappresenta un testo *text*;
* ***node.*insertAdjacentElement(“*position”*, *new\_element\_node*):** aggiunge l’element node *new\_element\_node* nella *position* indicata rispetto al nodo *node*:
  + Se “*position”* = “beforebegin”, *new\_element\_node* viene aggiunto come nodo fratello immediatamente precedente a *node*;
  + Se “*position”* = “afterbegin”, *new\_element\_node* viene aggiunto come primo nodo figlio   
    di *node*;
  + Se “*position”* = “beforeend”, *new\_element\_node* viene aggiunto come ultimo nodo figlio   
    di *node*;
  + Se “*position”* = “afterend”, *new\_element\_node* aggiunto come nodo fratello immediatamente successivo a *node*.
* ***node.*insertAdjacentText(“*position”*, *new\_text\_node*):** aggiunge il text node *new\_text\_node*   
  nella *position* indicata rispetto al nodo *node*:
  + Se “*position”* = “beforebegin”, *new\_text\_node* viene aggiunto come nodo fratello immediatamente precedente a *node*;
  + Se “*position”* = “afterbegin” *new\_text\_node* viene aggiunto come primo nodo figlio di *node*;
  + Se “*position”* = “beforeend”, *new\_text\_node* viene aggiunto come ultimo nodo figlio   
    di node;
  + Se “*position”* = “afterend”, *new\_text\_node* viene aggiunto come nodo fratello immediatamente successivo a *node*.
* ***node*.setAttribute(*attribute*, *value*):** imposta il valore dell’attributo *attribute* del nodo *node* a *value*;
* ***node.*remove():** rimuove il nodo *node* dal DOM;
* ***node*.replaceChild(new*Child, oldChild*):** rimpiazza il nodo *oldChild* figlio del nodo *node* con il nodo *newChild*, e restituisce un riferimento al nodo rimosso.

**EVENTI**

Un ***evento*** è “qualcosa” che il browser o l’utente fa.

Attraverso la funzione ***node*.addEventListener(“*event*”, *function*)** è possibile associare l’event handler *function* al nodo *node* per l’evento *event*, e cioè far sì che *function* vada in esecuzione ogni volta che *node* genera l’evento *event*. *function* può prendere in ingresso un parametro *e*, e quando *function* andrà in esecuzione a seguito del generarsi dell’evento, *e* conterrà un event object, e cioè un certo tipo di oggetto contenente informazioni circa l’evento che è stato generato. La proprietà ***e*.target** restituisce un riferimento al nodo che ha generato l’evento.

Prima di modificare il DOM, tuttavia, bisogna assicurarsi che il DOM sia stato effettivamente costruito.   
Per farlo, è possibile associare al nodo document un event handler per l’evento **DOMContentLoaded**, evento che viene generato quando il DOM viene costruito. L’idea, dunque, è di inserire tutto il codice che modifica il DOM e associa gli event handler ai nodi dentro un event handler per l’evento DOMContentLoaded.

Quando un nodo genera un evento, l’evento si propaga a tutti i suoi nodi antenati, e se questi hanno un event handler associato per lo stesso evento, questo andrà in esecuzione. La proprietà ***e*.currentTarget** restituisce un riferimento al nodo a cui attualmente è stato propagato l’evento.

In alcuni casi la propagazione degli eventi può essere utile: supponiamo di avere vari elementi img in un elemento section e di volere che, quando un utente clicca su un elemento img, succeda *qualcosa*:

<section id="list">

<h2>Section Title</h2>

<img ... >

<img ... >

...

<img ... >

</section>

Invece di associare lo stesso event handler ad ogni nodo img per l’evento onclick, quello che si può fare è associare un event handler solo al nodo section per l’event onclick che, quando l’utente clicca su un’immagine (e quindi, quando il nodo che ha generato l’evento ha nome “IMG”), fa *qualcosa*:

const parent = document.querySelector("#list");

function Handler(e){

if (e.target.nodeName == "IMG") {

*qualcosa*(e.target);

}

parent.addEventListener(“click”, Handler);

In questo modo, quando l’utente clicca su un elemento img, l’evento si propagherà al nodo section padre, verrà eseguito l’event handler associato al nodo section per tale evento, e così verrà effettivamente   
fatto *qualcosa.*

In altri casi, la propagazione degli eventi può causare problemi. E’ possibile allora interromperla,   
invocando il metodo ***e*.stopPropagation()**.

Vediamo ora alcuni tipi di eventi che vengono generati dall’interazione dell’utente con il documento HTML:

* **Mouse events**: vengono generati quando il mouse interagisce con un elemento HTML.   
  Alcuni mouse events sono:
  + click: viene generato quando viene fatto click su un elemento;
  + dblclick: viene generato quando viene fatto doppio click su un elemento;
  + mouseover: viene generato quando il cursore del mouse viene spostato sopra un elemento;
  + mouseout: viene generato quando il cursore del mouse viene spostato fuori da un elemento;
  + mousemove: viene generato quando il cursore del mouse viene spostato mentre si trova sopra un elemento;
  + mousedown: viene generato quando un tasto del mouse viene premuto su un elemento.

Per questo tipo di eventi sono importanti le seguenti proprietà dell’event object *e*:

* + *e*.button: restituisce un numero indicante quale tasto del mouse ha generato l’evento;
  + *e*.clientX: restituisce la coordinata orizzontale (espressa in pixel a partire dall’angolo in alto a sinistra del viewport) in cui si trovava il cursore quando è stato generato l’evento;
  + *e*.clientY: restituisce la coordinata verticale (espressa in pixel a partire dall’angolo in alto a sinistra del viewport) in cui si trovava il cursore quando è stato generato l’evento;
* **Keyboard events:** vengono generati quando viene premuto un tasto della tastiera.   
  Il relativo event handler può essere associato al nodo document (e quindi all’intero documento HTML), oppure ad un nodo che accetta input da tastiera. Alcuni keyboard events sono:
  + keydown: viene generato quando viene premuto un tasto della tastiera;
  + keyup: viene generato quando viene rilasciato un tasto della tastiera.

Per questo tipo di eventi è importante la seguente proprietà dell’event object *e*:

* + *e*.key: restituisce una stringa indicante quale tasto della tastiera ha generato l’evento.
* **Form events:** vengono generati quando si interagisce con un form. Alcuni form events sono:
  + focus: viene generato quando un elemento del form ottiene il focus;
  + blur: viene generato quando un elemento del form perde il focus;
  + change: viene generato quando viene cambiato il valore di un elemento del form;
  + reset: viene generato quando viene fatto il reset del form;
  + submit: viene generato quando viene fatto il submit del form. Un event handler per questo evento va in esecuzione prima che i dati vengano effettivamente mandati al form (dunque è possibile, ad esempio, fare una prima validazione dei dati del form e, se questa non va a buon fine, si può bloccare il submit, in un modo che vedremo successivamente).
* **Media events:** vengono generati quando si interagisce con un contenuto multimediale.   
  Alcuni media events sono:
  + ended: viene generato quando la riproduzione del contenuto multimediale viene completata;
  + pause: viene generato quando la riproduzione del contenuto multimediale viene messa in pausa;
  + play: viene generato quando la riproduzione del contenuto multimediale viene iniziata o viene tolta dalla pausa;
  + volumechange: viene generato quando il volume dell’audio viene modificato.

Quando un nodo genera un evento, oltre a venire eseguito l’eventuale event handler associato a tale nodo per tale evento, viene anche eseguita l’azione predefinita per tale evento. Se si vuole che questa non venga eseguita, nel corpo dell’event handler bisogna invocare il metodo ***e*.preventDefault()**:

*es.*

Quando viene fatto il submit di un form, oltre a venire eseguito l’eventuale event handler associato, viene eseguita anche l’azione predefinita, e cioè mandare i dati del form al server. Quello che si può fare allora è effettuare un primo controllo dei dati forniti dall’utente, e se risulta che questi   
non sono validi (*es.* se l’utente non ha inserito   
la password) non li si manda al server.

function passChecker(e){

let pass = document.querySelector("#pw").value;

if (pass == ""){

alert ("enter a password");

e.preventDefault();

}

}

document.querySelector("#loginForm").addEventListener("submit", passChecker);

**TIMER**

E’ possibile mandare in esecuzione una *function* una volta dopo tot *milliseconds* attraverso la funzione **setTimeout()**, in questo modo:

setTimeout(*function***,** *milliseconds*)

E’ possibile mandare in esecuzione una *function* ogni tot *milliseconds* attraverso la funzione **setInterval()**,   
in questo modo:

let *timerId* = setInterval(*function***,** *milliseconds*)

setInterval() restituisce un id che identifica il timer creato. E’ possibile interrompere il timer attraverso la funzione **clearInterval(),** passando l’id del timer:

clearInterval(*timerId*)

*es.*

let counter = 0;

In questo modo, *mycallback* verrà eseguita ogni secondo, per 5 volte.

**NB:** in questo esempio, la variabile *timerhandle* compare nel blocco della funzione *mycallback* quando ancora non è stata dichiarata. Questo è possibile in JS grazie al cosiddetto meccanismo dell’**hoisting**: non appena si entra in un blocco, come prima cosa vengono dichiarate tutte le funzioni e tutte le variabili del blocco, senza definirle.

Nel nostro caso, dunque, vengono subito dichiarate *counter*, *mycallback*() e *timerhandle*, ma senza definirle, dunque senza inizializzare *counter*, senza definire il corpo della funzione *mycallbac*k() e senza inizializzare *timerhandle*. Queste verranno poi definite man mano che si scende nel blocco, ma almeno, nell’atto della definizione di *mycallback*(), l’istruzione clearTimeout(*timerhandle*) non darà problemi, perché *timerhandle* è stata già dichiarata.

function mycallback(){

console.log("Counter: " + counter++);

if(counter === 5) {

clearTimeout(timerhandle);

}

}

let timerhandle = setInterval(mycallback, 1000);

**VALIDAZIONE INPUT**

Per verificare se un dato inviato dall’utente è un numero, il modo migliore è usare la seguente funzione,   
che restituisce true se *n* è un numero, false altrimenti:

function isNumeric(n) {

return !isNaN(parseFloat(n)) && isFinite(n);

}

**TRY…CATCH**

Quando si verifica un errore a runtime, viene lanciata un’eccezione che può bloccare l’esecuzione del codice Javascript. E’ possibile catturare questi errori, evitando così che l’esecuzione del codice venga interrotta, attraverso try…catch:

*Es.*

try {

nonexistantfunction("hello");

}

catch(err) {

alert("An exception was caught:" + err);

}

**INTRODUZIONE A PHP** *23-10-23*

**PHP** è un linguaggio di programmazione che viene utilizzato per creare contenuti lato server.

Un **frammento di** **codice PHP** è una sequenza di statement, racchiusa tra un *tag di apertura* <?php e   
un *tag di chiusura* ?>. Ogni statement termina per ‘;’.

Per dotare un documento HTML di codice PHP, bisogna innanzitutto cambiare la sua estensione in .php.   
A questo punto, sono due gli approcci possibili:

* **Mixed**: si scrive tutto il codice PHP nel documento HTML;
* **Separated**: si creano vari file .php in cui si definiscono tutte le classi, funzioni e altre cose utili   
  (es. frammenti di codice HTML), e li si includono nel documento HTML attraverso l’espressione include *nome\_file,* che equivale a copiare il contenuto di *nome\_file* nel documento HTML, nel punto in cui include viene incontrato (e quindi, se *nome\_file* presenta frammenti di codice HTML, questo porterà a modifiche del documento HTML nel punto in cui include viene incontrato).   
  A questo punto, il documento HTML presenterà solo il minimo codice PHP indispensabile, come la dichiarazione di nuovi oggetti classe o la chiamata a funzioni.

Una variabile si dichiara con questa sintassi:

$*nome\_variabile*

Una volta dichiarata, una variabile va sempre indicata preceduta dal simbolo ‘$’. E’ possibile inizializzare una variabile con un certo valore o assegnarle un valore successivamente attraverso l’operatore =.

PHP è un linguaggio debolmente tipizzato: una stessa variabile può contenere valori di tipi diversi in istanti diversi e, in base alla situazione, un valore di un certo tipo viene convertito automaticamente in un valore di un altro tipo.

Una costante si definisce in questo modo:

define(*constant\_name*, *constant\_value*)

Una volta definita, una costante può essere indicata senza precederla dal simbolo ‘$’.

I tipi in PHP sono i seguenti:

* Integer: un numero intero;
* Float: un numero decimale;
* Booleano: *true* o *false*;
* String: una sequenza di caratteri. Va scritta tra doppi o singoli apici;
* Null: un unico valore, che è *null*;
* Array: un array. Va scritto tra parentesi quadre, ed ogni elemento va separato da una virgola. Può essere multidimensionale, e può contenere elementi di tipo diverso. Di default, gli elementi sono indicizzati numericamente partendo dall’indice 0, ed è possibile accedere ad un elemento usando l’operatore []. Se si cerca di accedere ad un elemento che non esiste nell’array verrà restituito null.
* Object: un oggetto classe.

PHP presenta if…else, switch, operatore ternario, while, do…while e for. La sintassi è la stessa del C.

Gli operatori di confronto possibili sono i seguenti:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Gli operatori logici possibili sono i seguenti:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

In una condizione, tutti i valori vengono convertiti in true, ad eccezione di null, 0, la stringa “0”,   
la stringa vuota e un array con zero elementi.

Una funzione si dichiara in questo modo:

function *nome\_funzione*(*lista\_parametri*){

Ogni parametro va indicato preceduto dal simbolo ‘$’.

…

}

Non va specificato dunque né il tipo di ritorno della funzione, né il tipo dei parametri d’ingresso.

La sintassi per ritornare il controllo al chiamante e, eventualmente, ritornare un valore, così come la sintassi per invocare una funzione, è la stessa del C.

*es.*

function subtotal($price, $quantity) {

return $price \* $quantity;

}

$result = subtotal(10,2);

E’ possibile assegnare dei valori di default ai parametri, che verranno usati quando per tali parametri non viene passato alcun valore:

*es.*

function foo($a=10, $b=0) {

return $a + $b;

}

Di default, i parametri vengono passati per valore. Per fare in modo che un parametro venga passato   
per riferimento, bisogna far precedere tale parametro dal simbolo ‘&’ nella *lista\_parametri*:

*es.*

function changeParameter(&$arg) {

$arg += 300;

}

In questo modo, alla funzione *changeParameter* verrà passato come parametro *arg* proprio la variabile *initial*, e non una sua copia. La funzione, dunque, cambierà proprio il valore di *initial*.

$initial = 15;

changeParameter($initial);

Di default, a differenza degli altri linguaggi di programmazione, una variabile globale (e cioè, una variabile dichiarata fuori da ogni funzione) non è visibile all’interno di una funzione. Per rendere una certa   
variabile globale visibile all’interno di una certa funzione, all’inizio del corpo della funzione bisogna scrivere   
**global *nome\_funzione***:

*es.*

$count = 56;

function testScope() {

Rendo la variabile globale *count* visibile all’interno della funzione *testScope*, così che si possa accedere a tale variabile all’interno di tale funzione.

global $count;

echo $count;

}

Le stringhe si possono concatenare tra loro usando l’operatore . :

*Es.*

Dichiaro una variabile *country* e la inizializzo alla stringa “France”.

Dichiaro una variabile *city* e la inizializzo alla stringa “Paris”.

Dichiaro una variabile *msg* e la inizializzo alla stringa ottenuta concatenando *city*, “ is the capital of “ e *country*. La variabile, dunque, avrà come valore “Paris is the capital of France”.

$country = “France”;

$city = “Paris”;

$msg = $city . “ is the capital of “ . $country;

E’ possibile confrontare due stringhe tra loro usando gli operatori di confronto, e una stringa sarà minore   
di un’altra se viene prima nel dizionario.

Attraverso il costrutto **echo *stringa***è possibile stampare *stringa* (così com’è scritta) nel documento HTML, nel punto in cui tale costrutto viene incontrato:

*es.*

In questo modo verrà stampato “hello” nel documento HTML, nel punto in cui il costrutto echo viene incontrato.

In questo modo verrà stampato “Hello World” nel documento HTML, nel punto in cui il costrutto echo viene incontrato.

In questo modo verrà stampato ‘<img src=”23.jpg”>’ nel documento HTML, nel punto in cui il costrutto echo viene incontrato.

Notare che in questo caso ho usato i singoli apici per racchiudere   
i caratteri della stringa, in quanto tra i caratteri compare anche “,   
che usando i doppi apici sarebbe stato visto erroneamente   
come la fine della stringa. In alternativa avrei potuto far precedere “   
dal carattere di escape \, indicando così che “ non va inteso come   
la fine della stringa, ma come un carattere normale.

echo “hello”;

$what = “World”;

echo “Hello” . $what;

echo ‘<img src=”23.jpg”>’

Vediamo ora alcune funzioni utili per lavorare su una *string*:

* **explode(*separator, string, limit*):** restituisce un array ottenuto splittando *string* ogni volta che viene trovato il carattere *separator*, effettuando al massimo *limit* split. Una volta effettuati *limit* split, se non si è ancora arrivati alla fine di *string*, i caratteri rimanenti vengono esclusi dall’array.   
  Omettendo *limit*, non ci sarà alcun numero massimo di split;

Un altro modo di creare un array, che permette di specificare anche gli indici degli elementi, è il seguente:

$array = array(*index1* => element1*, index2* => element2, …, *indexN* => elementN)

Gli indici degli elementi possono essere anche non progressivi, e persino non numerici:

*es.*

Dichiaro un array *people* che ha come elementi, nell’ordine, “Bob” di indice 0, “Rick” di indice 3   
e “Gay” di indice 1.

Dichiaro un array *forecast* che ha come elementi, nell’ordine, 40 di indice “Mon”, 47 di indice “Tue” e 30 di indice “Wed”.

$people = array(0 => “Bob”, 3 => “Rick”, 1 => “Gay”);

$forecast = array(“Mon” => 40, “Tue” => 47, “Wed” => 30)

E’ possibile aggiungere un elemento ad un array semplicemente assegnando un valore al nuovo elemento dell’array, specificando un indice diverso da quelli già utilizzati. Questo elemento verrà aggiunto all’array, senza però garantire in quale posizione verrà aggiunto:

*es.*

Dichiaro un array *days,* inizialmente vuoto.

Aggiungo all’array *days* l’elemento “Tue”, di indice 1.

Aggiungo all’array *days* l’elemento “Mon”, di indice 0.

Se non specifica un indice, l’elemento verrà aggiunto in fondo all’array, con indice numerico pari al più alto indice numerico utilizzato + 1 (o 0 se non è stato utilizzato alcun   
indice numerico). Aggiungo dunque in fondo all’array *days* l’elemento “Wed”, di indice 2.

$days = [];

$days[1] = "Tue";

$days[0] = “Mon”;

$days[] = “Wed”;

La funzione **count(*array*)** restituisce il numero di elementi di *array*. Nel caso in cui gli elementi dell’array avessero indici numerici progressivi, questa funzione permette di scorrere gli elementi dell’array   
attraverso un ciclo for:

*es.*

$arr = [“A”, “B”, “C”];

for($i = 0; i < count($arr); $i++){

echo $*arr* [$i] . “<br>”;

}

Un modo alternativo di scorrere gli elementi di un array è attraverso un ciclo foreach, che permette di scorrere gli elementi dell’array anche nel caso in cui questi non avessero indici numerici progressivi:

*es.*

$grades = array(“Luca” => 18, “Mick” => 0, “Simo” => 30);

foreach($grades as $value){

Ad ogni iterazione, in *value* finirà il valore dell’elemento successivo dell’array *grades*.

echo $value . “<br>”;

}

foreach($grades as $key => $value){

Ad ogni iterazione, in *key* e in *value* finiranno rispettivamente l’indice e il valore dell’elemento successivo dell’array *grades*.

echo “name ”. $key “ = “ . $value;

}

Vediamo ora alcune funzioni utili per lavorare su un *array*:

* **unset(*var*):** distrugge *var*. Questa funzione può essere usata per eliminare un elemento da *array*,   
  e gli altri elementi manterranno invariato il proprio indice;
* **isset(*var*):** restituisce true se *var* è stata definita ed ha un valore diverso da null, false altrimenti. Questa funzione può essere usata per vedere se *array* presenta o meno un elemento che ha un certo indice.
* **array\_keys(*array*):** restituisce un array, indicizzato numericamente partendo dall’indice 0, contenente gli indici degli elementi di *array*;
* **array\_values(*array*):** restituisce un array, indicizzato numericamente partendo dall’indice 0, contenente i valori degli elementi di *array*;
* **begin(*array*):** restituisce il valore del primo elemento di *array*;
* **end(*array*):** restituisce il valore dell’ultimo elemento di *array*;
* **array\_reverse(*array*):** restituisce un array, indicizzato numericamente partendo dall’indice 0, contenente i valori degli elementi di array in ordine inverso;
* **array\_search(*array*, *val*):** restituisce l’indice del primo elemento di *array* che ha come valore *val*,   
  false se tale elemento non esiste;
* **shuffle(*array*):** mescola *array*, indicizzando numericamente partendo dall’indice 0 l’array risultante;
* **sort(*array*):** ordina *array* in senso crescente di valore, indicizzando numericamente partendo dall’indice 0 l’array risultante;
* **asort(*array*):** ordina *array* in senso crescente di valore, mantenendo le associazioni indice-valore originarie;
* **ksort(*array*):** ordina *array* in senso crescente di indice.

**ARRAY SUPERGLOBALI**

Gli **array superglobali** permettono di recuperare facilmente i dati inviati dal client al file PHP corrente.

L’array superglobale **$\_GET** contiene i dati inviati dal client appendendoli all’URL in forma *name1*=*value1*&*name2*=*value2*&…&*namen*=v*aluen*. Ognuno di questi dati ha dunque un *name,*ed è possibile accedere a ciascuno di essi scrivendo $\_GET[*name*]. Per un dato proveniente da un elemento di un form, il suo *name* sarà uguale al valore dell’attributo name dell’elemento del form da cui tale dato proviene.

L’array superglobale **$\_POST** contiene i dati (file esclusi) inviati dal client tramite richiesta HTTP di tipo POST.

Ognuno di questi dati ha un *name*, ed è possibile accedere a ciascuno di essi scrivendo $\_POST[*name*].   
Per un dato proveniente da un elemento di un form, il suo *name* sarà uguale al valore dell’attributo name dell’elemento del form da cui tale dato proviene.

L’array superglobale **$\_SERVER** contiene informazioni sul client e sulla richiesta HTTP che ha effettuato.   
Alcuni campi dell’array superglobale $\_SERVER sono:

* **$SERVER[“REQUEST\_METHOD”]:** restituisce una stringa contenente iltipo di richiesta HTTP effettuata dal client (*es.* “GET”, “POST”, …)
* **$SERVER[“REMOTE\_ADDR”]:** restituisce una stringa contenente l’indirizzo IP del client.

Per vedere se un certo dato è stato inviato o meno si può sfruttare la funzione isset():

*es:*

Supponendo che questo script debba andare in esecuzione a seguito di una richiesta HTTP di tipo POST contenente i dati di un form, vedo se l’array $\_POST presenta gli elementi di indice *uname* e *pass*, e quindi se l’utente ha inserito un dato negli elementi del form che hanno quel valore per l’attributo *name*.

**NB:** Alcuni browser, se per un certo elemento del form non si è inserito alcun dato, inviano comunque una stringa vuota. Questo non è un grosso problema, perché tanto poi bisognerà validare l’input attraverso le espressioni regolari, gestendo così anche il caso in cui l’input sia una stringa vuota.

if ((isset($\_POST[“uname”]) && $\_POST[“pass”] != “”){

…

}

L’array superglobale **$\_FILES** contiene un elemento per ogni file inviato dal client.

Ognuno di questi elementi ha un *name*, ed è possibile accedere a ciascuno di essi scrivendo $\_POST[*name*].   
Per un file proveniente da un elemento di un form, il suo *name* sarà uguale al valore dell’attributo name dell’elemento del form da cui tale file proviene.

Il valore di ogni elemento è a sua volta un array, che presenta i seguenti campi:

* **$\_POST[*name*][“error”]:** restituisce la costante UPLOAD\_ERR\_OK se il file è stato ricevuto con successo, una costante diversa se si è verificato un errore;
* **$\_POST[*name*][“name”]:** restituisce il nome che il file aveva quando è stato inviato dal client;
* **$\_POST[*name*][“size”]:** restituisce un intero indicante la dimensione in byte del file;
* **$\_POST[*name*][“tmp\_name”]:** restituisce il path temporaneo del file;
* **$\_POST[*name*][“type”]:** restituisce il mime-type del file;

Una volta ricevuto un file, questo verrà salvato con un altro nome ad un path temporaneo, da cui verrà rimosso prima o poi se non viene spostato da un’altra parte. Prima di spostarlo in quello che dovrà essere il suo path definitivo, tuttavia, conviene controllare che il file ricevuto sia effettivamente ciò che ci si aspettava.   
Tipicamente sono tre i controlli che vanno fatti su un file ricevuto dall’utente:

* Bisogna controllare che sia stato ricevuto con successo;
* Bisogna controllare che non superi la dimensione massima accettata;
* Bisogna controllare che abbia un’estensione e un mime-type accettati.

Supponiamo di avere un form in cui l’utente deve caricare un’immagine che sia di estensione .jpg o .png   
e che non superi la dimensione di 1MB. Un modo di procedere è mandare l’immagine al seguente script PHP:

Definisco una variabile contenente la dimensione massima accettata espressa in byte, e due array contenenti rispettivamente le estensioni e   
i mime-type accettati.

Per comodità, salvo in *uploadedFile* il valore $\_FILES[“file”], così da non dover scrivere ogni volta $\_FILES[“file”][…].

Se il file non è stato ricevuto con successo, mostro a video un errore e termino lo script.

Se il file ha una dimensione superiore a quella massima accettata, mostro a video un errore e termino lo script.

Recupero l’estensione e il mime-type del file.

Per recuperare la prima, considero l’array ottenuto splittando ad ogni “.” la stringa contenente il nome che il file aveva quando è stato inviato dal client, e prendo l’ultimo elemento.

Se il file ha un’estensione o un mime-type che non è tra quelli accettati, mostro nel documento HTML   
un errore e termino lo script.

Arrivati qui, il file è ciò che ci si aspettava, dunque può essere spostato in quello che dovrà essere il suo path definitivo. Recupero allora il path temporaneo del file, scelgo quello che dovrà essere il path definitivo del file (in questo caso, la directory *upload*, decidendo anche di rinominare il file ridandogli il nome che aveva quando è stato inviato dal client), e invoco la funzione **move\_uploaded\_file(*path\_temporaneo\_file*, *path\_definitivo\_file*),** che sposta un file ricevuto dal client da *path\_temporaneo\_file* a *path\_definitivo\_file,* restituendo true se tutto è andato bene, false altrimenti. Salvo allora in una variabile il valore restituito dalla funzione, e se si è verificato un errore nello spostare il file, mostro nel documento HTML un errore e termino lo script.

$max\_file\_size = 1000000;

$validExt = [“jpg”, “png”];

$validMime = [“image/jpeg”, “image/png”];

$uploadedFile = $\_FILES[“file”];

if ($uploadedFile[“error”] != UPLOAD\_ERR\_OK){

echo “Errore: file non ricevuto con successo”;

exit();

}

if ($uploadedFile[“size”] > $max\_file\_size){

echo “Errore: dimensione troppo grande”;

exit();

}

$file\_extension = end(explode(“.”, $uploadedFile[“name”]);

$file\_mime = $file[“type”];

if (!array\_search($file\_extension, $validExt

|| !array\_search($file\_mime, $validMime){  
 echo “Errore: estensione e/o mime-type non valido”;

exit();

}

$current = $file[“tmp\_name”];

$destination = “../upload/” . $uploadedFile[“name”];

$ret = move\_uploaded\_file($current, $destination);

if (!$ret){

echo “Errore nello spostare il file”;

exit();

}

**MYSQLI**

**Mysqli** è un’estensione PHP che permette di accedere ad un database MySQL.

Il modo generale per farlo è il seguente:

1. Come prima cosa, bisogna aprire una connessione con il database. Questo si fa invocando la funzione **mysqli\_connect(*hostname*, *dbname*, *dbuser*, *dbpass*)**, che apre una connessione con il database *dbname*, presente sull’host *hostname*, acquisendo i privilegi dell’utente che ha come username e password rispettivamente *dbuser* e *dbpass*. Restituisce un oggetto che rappresenta la connessione appena aperta.

Una volta invocata la funzione, bisogna controllare che la connessione con il database sia stata aperta senza errori. Per farlo, si può invocare la funzione **mysqli\_connect\_errno()**, che restituisce 0 se la connessione è stata aperta con successo; un numero diverso se si è verificato un errore. La funzione **mysqli\_connect\_error()** restituisce una descrizione dell’errore che si è verificato.

Dato che ci si troverà spesso ad accedere al database, invece   
di scrivere a mano ogni volta i parametri di mysqli\_connect, conviene definire delle costanti che hanno come valore tali parametri,   
e invocare la funzione passando tali costanti.

define(“DBHOST”, *hostname*);

define(“DBNAME”, *dbname*);

define(“DBUSER”, *username*);

define(“DBPASS”, *password*);

Apro la connessione con il database. Se si è verificato un errore, mostro nel documento HTML la descrizione dell’errore e termino lo script.

$connection = mysqli\_connect(DBHOST, DBUSER, DBPASS, DBNAME)

if(mysqli\_connect\_errno()){

exit(mysqli\_connect\_error());

}

1. A questo punto, si può preparare ed eseguire la query. Vanno distinti due casi:
   1. Se la query non contiene dati inseriti dall’utente, una volta preparata una stringa *query* contenente la query da eseguire senza il ‘;’ alla fine, si può invocare la funzione   
      **mysqli\_query(*connection*, *query*)**, che esegue *query* sulla connessione rappresentata dall’oggetto *connection*. Restituisce:
      * Un oggetto che rappresenta il result set restituito da *query*, se *query* restituisce un result set e non si sono verificati errori;
      * true, se *query* non restituisce un result set e non si sono verificati errori;
      * false, se si è verificato un errore.

Una volta eseguita la query con successo, se questa restituisce un result set, per prelevarlo si può creare un ciclo in cui si invoca ad ogni iterazione la funzione **mysqli\_fetch\_assoc(*result*),**

che restituisce:

* + - Un array contenente i valori degli attributi del record successivo del result set rappresentato dall’oggetto *result*, se non si è ancora arrivati alla fine del result set.   
      L’array è indicizzato dal nome degli attributi del result set;
    - null, se si è arrivati alla fine del result set.

Preparo la query e la eseguo. In caso di errore, termino lo script.

$query = “SELECT ID, CName FROM Categories ORDER BY OrderName”;

$result = mysqli\_query($connection, $query);

if(!$result){

exit();

}

Scorro tutto il result set restituito dalla query, mostrando nel documento HTML i valori degli attributi “ID” e “CName” di ogni record.

while($row = mysqli\_fetch\_assoc($result){

echo $row[“ID”] . “ - “ . $row[“CName”] . “<br>”;

}

* 1. Se la query contiene dati inseriti dall’utente, bisogna prima creare un *prepared statement*, e cioè un template della query memorizzato nel DBMS. Per fare ciò, bisogna prima creare una stringa *parametrized\_query* contenente la query da eseguire senza il ‘;’ alla fine, mettendo il simbolo ‘?’ in ogni in punto in cui dovrà essere messo un dato inserito dall’utente. Fatto ciò, bisogna invocare la funzione **mysqli\_prepare(*connection*, *parametrized\_query*)**, che crea un prepared statement sulla base di *parametrized\_query,* da poter usare per la connessione rappresentata dall’oggetto *connection*. Restituisce:
     + Un oggetto che rappresenta il prepared statement creato, se non si sono verificati errori;
     + false se si è verificato un errore.

Una volta invocata la funzione con successo, bisogna associare ad ogni simbolo ‘?’ del prepared statement il corrispondente dato inserito dall’utente. Questo si fa invocando la funzione **mysqli\_stmt\_bind\_param(*prepared\_stmt, types*, *lista\_dati*)**, che associa all’i-esimo simbolo ‘?’ del prepared statement rappresentato dall’oggetto *prepared\_stmt* l’i-esimo dato presente in *lista\_dati*. *types* è una stringa, dove l’i-esimo carattere indica come deve essere interpretato l’i-esimo dato:

* + - Se l’*i*-esimo carattere è i, allora l’*i*-esimo dato verrà interpretato come un intero;
    - Se l’*i*-esimo carattere è s, allora l’*i*-esimo dato verrà interpretato come una stringa;
    - …

A questo punto, per eseguire il prepared statement, bisogna invocare la funzione **mysqli\_stmt\_execute(*prepared\_stmt*)** che esegue il prepared statement rappresentato dall’oggetto *prepared\_stmt,* sostituendo ogni simbolo ‘?’ con il corrispondente dato associato attraverso l’ultima chiamata alla funzione mysqli\_stmt\_bind\_param(). Restituisce:

* + - true se non si sono verificati errori;
    - false se si è verificato un errore.

Una volta eseguito il prepared statement con successo, se questo restituisce un result set, per prelevarlo bisogna prima invocare la funzione **mysqli\_stmt\_get\_result(*prepared\_stmt*),** che restituisce un oggetto che rappresenta il result set restituito dall’ultima esecuzione di *prepared\_stmt*. Fatto ciò, si può creare un ciclo in cui si invoca ad ogni iterazione la funzione mysqli\_fetch\_assoc(*result*), nel modo descritto prima.

Recuperati i dati inseriti dall’utente, creo la *parametrized\_query*

$id = $\_GET[“id”];

$query = “SELECT Title, Year FROM Books WHERE ID=?”;

$statement = mysqli\_prepare($connection, $query);

Preparo il prepared statement.   
In caso di errore, termino lo script.

if (!$statement){

exit();

}

Associo ad ogni simbolo ‘?’ del prepared statement il corrispondente dato inserito dall’utente, ed eseguo il prepared statement. In caso di errore, termino lo script.

mysqli\_stmt\_bind\_param($statement, “i”, $id);

$success = mysqli\_stmt\_execute($statement);

if(!$success){

exit();

}

$result = mysqli\_stmt\_get\_result($statement)

Scorro tutto il result set restituito dal prepared statement, mostrando nel documento HTML i valori degli attributi “Title” e “Year”   
di ogni record.

while($row = mysqli\_fetch\_assoc($result){

echo $row[“Title”] . “ - “ . $row[“Year”] . “<br>”;

}

Se si vuole eseguire un’altra query, bisogna prima liberare la memoria occupata dal result set restituito dalla query precedente. Questo si fa invocando la funzione **mysqli\_free\_result(*result*)**, che libera la memoria occupata dal result set rappresentato dall’oggetto *result*.

1. Infine, bisogna chiudere la connessione. Questo si può fare invocando la funzione **mysqli\_close(*connection*)**, che chiude la connessione rappresentata dall’oggetto *connection*.

mysqli\_close($connection);

Per quanto riguarda i file inviati dagli utenti, dopo averli salvati da qualche parte, memorizzeremo nel database il loro path espresso a partire dalla directory root del web server (e non a partire dalla directory base, visto che questa sarà diversa a seconda del documento HTML corrente). In questo modo, quando in un documento HTML ci sarà bisogno ad un certo punto di un file inviato da un utente, basta accedere al database per recuperare il suo path e includerlo opportunamente (*es*. attraverso un elemento <img> se è un’immagine, attraverso un elemento <audio> se è un audio…).

Per quanto riguarda le informazioni dell’utente, la password va memorizzata hashata nel database.   
Per creare un hash a partire da una stringa si utilizza la funzione password\_hash(), usando come algoritmo PASSWORD\_BCRYPT. Per verificare la correttezza della password inserita dall’utente si utilizza la funzione password\_verify().

Vedi poi da slide alcuni esempi di accesso al database per fare cose utili.

**MECCANISMO DELLE SESSIONI**

Il **meccanismo delle sessioni** permette di memorizzare dati di ogni tipo circa l’utente che sta utilizzando l’applicazione web.

Per attivare il meccanismo delle sessioni in un certo file PHP, prima di ogni altro statement, bisogna invocare la seguente funzione:

session\_start()

Da questo momento in poi si ha a disposizione l’array $\_SESSION, in cui si possono inserire dati   
di ogni tipo circa l’utente che sta utilizzando l’applicazione web. Questi dati rimarranno memorizzati fino a che l’utente non chiude il browser, e saranno accessibili in tutti i file PHP in cui il meccanismo delle sessioni   
è attivo.

Questo meccanismo può servire a varie cose:

* A memorizzare se l’utente ha effettuato o meno l’accesso all’applicazione web:

session\_start();

Quando l’utente effettuerà l’accesso all’applicazione web, posso memorizzare in $\_SESSION[“user”] il suo username. Prima di allora, dunque, questo elemento non sarà presente nell’array, e da ciò posso capire se l’utente ha effettuato o meno l’accesso all’applicazione web.

if (isset($\_SESSION[“user”]){

…

}

else{

…

}

* A memorizzare il carrello della spesa dell’utente:

include(“ShoppingCart.class.php”);

session\_start();

Posso memorizzare in $\_SESSION[“Cart”] il carrello della spesa dell’utente.   
Se questo elemento non è presente nell’array, lo aggiungo.

if(!isset($\_SESSION[“Cart”])){

$\_SESSION[“Cart”] = new ShoppingCart();

}