Java Script

Rev. 4.8 del 04/04/2024

Java Script

Con il termine **script** si intende un **insieme di comandi interpretati da un certo applicativo** che, nel caso di Java Script, è costituito dal browser. Java Script è un linguaggio che consente di scrivere, all' interno di una pagina HTML, delle sezioni di codice che verranno interpretate dal browser al momento della visualizzazione della pag

Un tag HTML può richiamare uno script js mediante l'utilizzo dei cosiddetti **ATTRIBUTI DI EVENTO** che, in risposta ad un certo evento come ad esempio un click, consentono di associare delle righe di codice da eseguire nel momento in cui si verificherà l'evento. La potenzialità di java script consiste principalmente nel fatto che, indipendentemente dal tag che ha scatenato l'evento, **le istruzioni possono accedere in modo diretto a qualunque tag della pagina e modificarne sia l'aspetto grafico sia il contenuto**.

Le istruzioni Java Script possono essere scritte:

- Direttamente all'interno del tag (se le azioni da compiere sono poche)
```
<input type="button" value = "Esegui" onClick="alert('salve')">
```
- All'interno della sezione di HEAD della pagina all'interno del seguente TAG
```
<script type="application/javascript" >
    istruzioni js
</script>
```
- In un file esterno con estensione .js richiamato nel modo seguente:
```
<script type="application/javascript" src="index.js"> </script>
```

Sintassi base

- Le **istruzioni base** sono le stesse dell'ANSI C e sono **case sensitive** (distinzione maiuscole / minuscole)
- Il **punto e virgola** finale è facoltativo. Diventa obbligatorio quando si scrivono più istruzioni sulla stessa riga.
- Lo switch accetta come parametro anche le stringhe (come in C#, a differenza del C ANSI)
- Java Script non distingue tra **virgolette doppie** e **virgolette semplici.** Ogni coppia deve essere dello stesso tipo
  Per eseguire un terzo livello di annidamento si può usare \" oppure " : alert ("salve \"Mondo\" ");
- Le variabili dichiarate fuori dalle funzioni hanno visibilità globale ma **non vengono inizializzate.**

Dichiarazione delle variabili

```
let A, B;
```

- **Le variabili non sono tipizzate**, cioè nella dichiarazione non deve essere specificato il tipo. I tipi sono comunque **Number, String** (object), **string** (scalare), **Boolean, Object**, e il cast al tipo viene eseguito in automatico in corrispondenza di ogni assegnazione. Il tipo **Float** non esiste (usare Number) ma esiste parseFloat( ). Oltre a **`let`** è possibile utilizzare **`const`** per le costanti
- E' comunque possibile tipizzare una variabile creando una istanza esplicita:
```
let n=new Number();
let s=new String();
let b=new Boolean();    // true e false sono minuscoli.
```
  Le parentesi tonde sono quelle del costruttore e possono indifferentemente essere scritte oppure omesse.
- **La dichiarazione delle variabili non è obbligatoria**. Una variabile non dichiarata viene automaticamente creata in corrispondenza del suo primo utilizzo **come variabile globale**, e dunque sarà visibile anche fuori dalla funzione. Facile causa di errori. "**`use strict`**" rende obbligatoria la dichiarazione delle variabili
- Al momento della dichiarazione ogni variabile viene inizializzata al valore **`undefined`** che non equivale a \0 e nemmeno a false, ma equivale a "non inizializzato". Si può anche testare: **`if(A == undefined)`**
- Per i riferimenti si utilizza il valore **`null`**
- Il test **`if (!myVar)`** da esito positivo se myVar è uguale ad **undefined** o **null** o **false** o "" o **NaN**
- Un **json vuoto** { } è invece un oggetto, quindi diverso da tutti i valori precedenti
- In javascript non è consentito passare parametri scalari per riferimento (occorre trasformarli in object)

Funzioni Utente

Occorre omettere **sia** il tipo del parametro **sia** il tipo del valore restituito.

```
function eseguiCalcoli(n) {
     istruzioni;
     return risultato;  }
```

"use strict"

Inserito sulla prima riga di un file js obbliga l'utente a dichiarare le variabili.

L'accesso agli elementi della pagina

Il metodo `document.getElementById` consente di accedere al TAG avente l'ID indicato come parametro. Restituisce un riferimento all'elemento indicato. [In caso di più elementi con lo stesso ID ritorna il primo che trova].

```
let _div = document.getElementById("txtNumero");
let n = parseInt(_div.value);
```

Il metodo `getElementsByName`

Notare il plurale (**Elements**) necessario in quanto il **`name`** non è univoco e quindi gli elementi individuati sono normalmente più d uno. Restituisce un oggetto **NodeList** contenente tutti gli elementi aventi il **`name`** indicato, nell'ordine in cui sono posizionati all'interno della pagina. Un **NodeList** è in pratica un **vettore enumerativo** accessibile tramite indice numerico e con la proprietà .length.
Anche nel caso in cui sia presente un solo elemento viene comunque **sempre** restituito un vettore (lungo 1).

```
let _opts = document.getElementsByName("optHobbies");
  for (let i=0;i< _opts.length;i++)
    console.log(_opts[i].value);
```

I meotdi `getElementsByTagName` e `getElementsByClassName`

La prima restituisce sotto forma di **NodeList** (vettore enumerativo) tutti i tag di un certo tipo (es DIV).
La seconda restituisce sotto forma di **NodeList** (vettore enumerativo) tutti i tag che implementano una certa classe.
Entrambe accedono a figli e nipoti. Restituiscono un **NodeList** anche nel caso in cui sia presente un solo elemento:

```
let _body= document.getElementsByTagName("body")[0];
_body.style.backgroundColor="blue";
let _carte = document.getElementsByClassName("carta");     // senza il puntino
_carte[i].style.backgroundColor="red";
```

Ricerche annidate

```
let _wrapper = document.getElementById("wrapper")
let vet = _wrapper.getElementsByTagName("p");  // tag <p> interni a wrapper
```

Nota: **`getElementById`** e **`getElementsByName`** possono essere applicati SOLO a **document** .

Accesso agli attributi HTML e alle proprietà di stile

Dopo aver 'agganciato' il puntatore ad un elemento della pagina:

- Si può accedere a tutti i suoi **attributi HTML** mediante l'uso dell'operatore puntino : txtRis.value = 72
  Gli attributi HTML sono sempre di tipo STRING. La conversione da numero a string è fatta in automatico
- Si può accedere a tutte le sue **proprietà CSS** mediante l'uso dell'attributo **`.style`**
  `div.style.border = "2px solid black";`
  La proprietà CSS contenenti un trattino (come ad esempio **`border-color`**) in javascript devono essere scritte con tutto attaccato in camelCasing: ad esempio **`borderColor`**

Le proprietà .innerHTML e .textContent

Per accedere al contenuto interno di un tag (quello scritto tra **<apertoTag>** e **</chiusoTag>** si possono utilizzare due attributi dinamici (nel senso che esistono solo in javascript e non nella pagina html) che sono `.innerHTML e .textContent.` La differenza consiste nel fatto che:

innerHTML Accetta come valore qualsiasi tag HTML e restituisce l'interno contenuto html presente nel tag
textContent accede/imposta il solo contenuto testuale del tag (anche se annidato al''interno di più sotto-tag).

```
myDiv.innerHTML = "<i>Questo è il mio testo</i>"
```

Sono disponibili anche altri due attributi "minori" :

outerHTML mentre innerHTML accede al contenuto del tag (tag radice escluso), outerHTML restituisce anche il tag radice (cioè il tag al quale si applica la property). Può essere usata per cambiare anche il tag radice.

innerText simile a textContent, standardizzata più tardi nel DOM model. Leggermente più pesante. Differenze:

- Restituisce anche eventuali attributi hidden
- Riconosce il \n
- E' definita solo nel caso degli HTMLElement (che sono un sottoinsieme dei NodeElement definiti in XML)

L'evento onload

Nel file JS è possibile scrivere ***codice diretto*** "esterno" a qualsiasi funzione. Questo codice viene però eseguito **prima** che sia terminato il caricamento della pagina per cui **non** può fare riferimento agli oggetti della pagina che verranno istanziati soltanto al termine del caricamento. In tal caso i riferimenti saranno tutti NULL.

Per eventuali inizializzazioni occorre utilizzare, all'interno del file html, l'evento **<body onload="init()">**

Anziché definirlo all'interno della pagina HTML, l'evento **onLoad** può essere definito direttamente all'interno del file javascript come evento dell'oggetto window (finestra corrente)

```
window.onload = function(){ ..... }      // window può essere omesso
```

L'evento viene però generato ***SOLO SE*** non è già stato definito all'interno del body html (che è prioritario).

Gli oggetti base

L'oggetto Math

Proprietà Statiche

Math.PI	PI greco	3,1416
Math.E	Base logaritmi naturali	2,718

Math.LN2	Logaritmo naturale di 2
Math.LN10	Logaritmo naturale di 10
Math.LOG2E	Logaritmo in base 2 di e
Math.LOG10E	Logaritmo in base 10 di e

Metodi Statici

Math.sqrt(N)	Radice Quadrata
Math.abs(N)	Valore assoluto di N (modulo)
Math.max(N1, N2)	Math.min(N1,N2) Restituiscono il maggiore / minore in una sequenza di n numeri
Math.pow(10,N)	Elevamento a potenza : 10^N // oppure 10 ** N
Math.round(N)	Arrotondamento all'intero più vicino
Math.ceil(N)	Arrotondamento all'intero superiore
Math.floor(N)	Tronca all'intero inferiore. **L'operatore di divisione / restituisce un double anche nel caso di divisione fra interi, per cui il risultato deve eventualmente essere troncato.**
%	**restituisce il resto di una divisione fra interi**
Math.random()	Restituisce un double 0 <= x < 1 **esattamente come in ANSI C.** *Randomize è automatico* Per generare un num tra A e B (A compreso, B escluso) `Math.floor((B-A)*Math.random())+ A`

Metodi di istanza: toFixed()

```
let n = 12.34567;
alert(n.toFixed(2))     // 12.34
```

Le Stringhe

String è un oggetto che deve pertanto essere istanziato**.** Le stringhe sono **immutabili** come in C#
Per dichiarare una stringa sono disponibili le due seguenti sintassi (l'istanza statica **`String s1`** non è supportata):

```
let s1;                  // riferimento non tipizzato. Contiene undefined
let s2 = new String();   // riferimento tipizzato ma che contiene sempre undefined
```

Inizializzazione di una stringa in fase di dichiarazione

```
let s3 = "salve";               // oggetto stringa inizializzato a "salve"
let s4 = new String("salve");   // oggetto stringa inizializzato a "salve"
let len = s1.length    // lunghezza della stringa
```

s1 = s1.**toUpperCase**() // Restituisce la stringa convertita in maiscuolo
s2 = s1.**toLowerCase**() // Restituisce la stringa convertita in minuscolo
s2 = s1.**substr**(posIniziale, [qta]) // Estrae i caratteri da posIniziale per una lunghezza pari a qta. Deprecato
s2 = s1.**substring**(posIniziale, posFinale) // Estrae i caratteri da posIniziale a posFinale, posFinale escluso.
s1 = "Salve a tutti" s2=s1.substring(0, 5) => "Salve". Se posFinale > length si ferma a fine stringa
ris = s1.**includes**(s2) // Restituisce **true** se s1 include s2. **false** in caso contrario. **Disponibile anche sui vettori**
pos = s1.**indexOf**(s2, [pos]) // Ricerca s2 dentro s1 a partire dalla posizione pos. (Il primo carattere ha indice 0).
Restituisce la posizione della prima ricorrenza. Se non ci sono occorrenze restituisce –1
Se s2 = "" restituisce il carattere alla posizione pos. **Disponibile anche sui vettori**
pos = s1.**lastIndexOf**(s2, [pos]) // Ricerca s2 dentro s1 a partire dalla posizione pos andando all'indietro.
Se pos non è specificato parte dalla fine della stringa (length – 1).
pos = s1.**search**(s2) // Identica a indexOf ma **accetta le Regular Expr. N**on è disponibile sui vettori
s2 = s1.**replace**("x", "y") // Sostituisce **SOLO la prima occorrenza**.
replaceAll("x", "y") // Sostituisce tutte le occorrenze
s2 = s1.**repeat**(3) // Ripete 3 volte il contenuto di s1
ris = s1.**startsWith**(s2) // Restituisce TRUE se s1 inizia con la sottostringa s2
ris = s1.**endsWith**(s2) // Restituisce TRUE se s1 termina con la sottostringa s2

C = s1.**charAt**(pos) // Restituisce come **stringa** il carattere alla posizione pos (partendo a 0)
N = s1.**charCodeAt**(pos) // Restituisce il **codice Ascii** del carattere alla posizione pos (o del primo se manca pos)
s2 = **String.fromCharCode**(97,98,99) Viene istanziata una nuova stringa contenente "abc"
s2 = n.**toString**() // Restituisce la conversione in stringa. **n.toString(16)** restituisce una stringa esadecimale
n.toString(2) restituisce una stringa binaria
vect = s.**split**("separatore"); // Esegue lo split rispetto al carattere indicato.
s = vect.**join**("separatore"); // Restituisce in un'unica str tutti gli elementi del vett separati dal chr indicato

n.**toFixed**(3) // indica il numero di cifre dopo la virgola da visualizzare

- Proprietà e metodi possono essere applicati anche in forma diretta: "Mario Rossi".length
- A differenza di Ansi C in js le stringhe possono essere **confrontate** direttamente con gli operatori < >

Una semplice funzione per aggiungere uno 0 davanti ad un numero <10

```
function pad(number) {
     return (number < 10 ? '0' : '') + number
}
```

Altri metodi relativi alla formattazione grafica

.big()	restituisce una stringa in testo grande
.blink()	restituisce una stringa con testo lampeggiante
.bold()	restituisce una stringa in grassetto
.fontsize()	restituisce una stringa avente il fontsize specificato
.italics ()	restituisce una stringa in corsivo
.small()	restituisce una stringa in testo piccolo
.strike()	restituisce una stringa barrata
.sup()	restituisce una stringa in formato apice
.sub()	restituisce una stringa in formato pedice

Vettori Enumerativi (Indexed Array, cioè vettori indicizzati)

I vettori sono liste a indice 0 e possono essere dichiarati anche SENZA specificare la dimensione.
Il vettore crescerà dinamicamente man mano che si aggiungeranno elementi al suo interno.
Per creare un vettore sono disponibili due sintassi equivalenti, una breve ed una più prolissa:

```
let vect = [ ];                 // dichiaro un vettore enumerativo al momento avente lunghezza 0
let vect = new Array()          // Simile alla precedente ma tipizzata
```

L'assegnazione della dimensione in fase dichiarativa può essere realizzata soltanto mediante la seconda sintassi:

```
let vect = new Array(30)        // Viene istanziato un array di 30 elementi
```

Attenzione che invece la seguente sintassi dichiara un vettore lungo 1 contenente il valore 30:

```
let vect = [30];                // Viene istanziato un array contenente un solo elemento, con valore 30.
```

Inizializzazione di un vettore in fase di dichiarazione

```
let vect = new Array(30,31)     // Viene istanziato un array di 2 elementi, contenente i valori 30 e 31
let vect = [30, 31];            // Viene istanziato un array di 2 elementi, contenente i valori 30 e 31

let vect = new Array ('pippo',  'pluto',  'minnie');
let vect = ['pippo',  'pluto',  'minnie'];

let vect = new Array ('pippo');     // Viene istanziato un array di 1 elemento, contenente 'pippo'
let vect = [titolo, autore, categoria, prezzo];   // Viene caricato nel vettore il contenuto delle variabili indicate
```

Proprietà e Metodi di un vettore enumerativo

Gli Array possono essere **eterogenei**, cioè contenere dati differenti: numeri, stringhe, oggetti, etc
Per accedere all'i-esimo elemento si usano le **parentesi quadre** vectStudenti[3] = "Mario Rossi"

```
vect.length             // lunghezza del vettore
vect.push("a", "b", "c");   // Gli elementi indicati vengono aggiunti in coda al vettore
                            // Accetta come parametro anche un intero vettore che viene concatenato
vect.unshift("a", "b", "c");        // Gli elementi indicati vengono aggiunti in testa al vettore
vect.pop( );            // Restituisce l'ultimo elemento in coda al vettore, eliminandolo dal vettore
vect.shift( );          // Restituisce il primo elemento in testa al vettore, eliminandolo dal vettore
vect[vect.length] = "value";    // Equivalente al push()
vect[99] = "value";     // Se il vettore fosse lungo 10, verrebbe create 90 celle,
                            con le celle 10-98 undefined, ma comunque con length==100
pos = vect.indexOf(item)        // Restituisce la posizione dell'elemento indicato.
ris = vect.includes(item)       // Restituisce true se vect include l'elemento indicato.
                            // Per quanto riguarda gli oggetti si rimanda al modulo "XML and JSON"

vect.splice(pos, n);    // Consente di eliminare n oggetti a partire dalla posizione pos.
                        // Se pos == -1 elimina l'ultimo elemento !
                        // Restituisce un vettore contenente gli elementi eliminati.
```

Se come secondo parametro si passa 0 **splice()** consente di aggiungere nuovi elementi alla **pos** indicata
`vect.splice(pos,0,"A","B")` Aggiunge "A" e "B" alla posizione indicata da **pos**
E' anche possibile eseguire rimozione e aggiunta contemporaneamente :
`vect.splice(pos,3,"A","B")` Vengono rimossi 3 elementi alla posizione **pos** e vengono aggiunti "A" e "B"

In alternativa a **splice()** per eliminare una cella da un vettore enumerativo si può anche utilizzare **delete**
`delete vect[3]` che cancella il contenuto della cella **senza però ricompattare il vettore** !!

```
vect.sort();            // Ordinamento crescente. Agisce direttamente sul vettore, senza doverlo riassegnare
vect.reverse();         // Ordinamento decrescente.
let vect2= vect.slice(1, 3);  // Restituisce un nuovo vettore contenente gli elementi 1 e 2 (3 escluso) del vettore
                              originale. Numeri negativi consentono di selezionare a partire dal fondo.
let vect2= vect.slice(2, -3); // Elimina le prime due celle (0 e 1) e le ultime 3 celle
```

Disponibile anche sulle stringhe

Il metodo statico **`let array2 = Array.from(array1)`** copia un array su un altro nuovo array.

La scansione di un vettore

Il ciclo for presenta due sintassi diverse, una per vettori enumerativi l'altra per vettori associativi:

vettori enumerativi : `let vet=["a", "b", "c"];`
`for(let item of vet) alert(item);` `// item` rappresenta il contenuto della cella

Attenzione che, nel ciclo **FOR OF**, la variabile di ciclo **`item`** (detta **cursore**) è una **copia** del contenuto della cella, per cui eventuali modifiche apportate al cursore vanno perse al termine del ciclo.

vettori associativi : `let vet={a:"value1", b:"value2"};`
`for(let key in vet) alert(vet[key]) //` **key** è una stringa che rappresenta la chiave

Nota: il ciclo FOR IN in realtà può essere usato anche per i vettori enumerativi, nel qual caso però **i** è una stringa che rappresenta l'indice della cella, che dovrà eventualmente essere convertita in intero.

Il metodo funzionale **`vet.forEach(function(item, i))`** consente di scorrere un vettore enumerativo analogamente al for-of, ma essendo asincrono è più veloce. Il 2° parametro i è facoltativo

Inserimento di una variabile all'interno di una stringa

Il **backtick** (apice singolo rovesciato = **`ALT 96)`** è uno speciale delimitatore di stringa che:

- consente di inserire delle variabili direttamente all'interno della stringa tramite **`${varName}`**
- consente di **andare a capo** all'interno della stringa, che può essere scritta su più righe di testo

```
`Sono l'elemento ${i} Il mio valore è ${vet[i]}`
```

Le Matrici

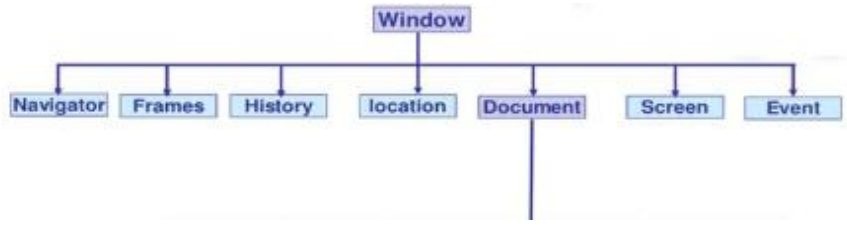
In Java Script non esiste il concetto canonico di matrice. E' però possibile, dopo aver dichiarato un vettore, trasformare ogni cella del vettore in un puntatore ad un nuovo vettore. Esempio di creazione di una **matrice 10 x 3** :

```
let mat = new Array (10);
// Le seguenti istruzioni sono obbligatorie Devo far capire a JS che si tratta di una matrice e non di un vettore
for (i=0; i<mat.length; i++)
    mat[i] = new Array (3);     // meglio: mat[i]=[]
mat[0][0] = "Marsha";   mat[0][1] = "Carol";   mat[0][2] = "Greg";
```

Il DOM di una pagina HTML

Il **DOM** (*Document Object Model*) è una **API** (*Application Programming Interface*), è **il modello con cui javascript "vede" la pagina HTML**. Cioè, attraverso il DOM. javascript mette a disposizione **un insieme di oggetti e funzioni che consentono di accedere a tutti gli elementi della pagina HTML**.

Il DOM javascript è definito nello standard W3C. L'oggetto base del DOM è l'oggetto **`window`** che rappresenta la scheda di navigazione corrente e rimane allocato finché la finestra non viene chiusa.



I principali oggetti figli dell'oggetto windows sono:

- **document** = documento html caricato nella finestra (analogo di xmlDoc nei documenti XML)
- **location** = informazioni sulla url corrente
- **navigator** = oggetto di navigazione. Utile ad esempio per eseguire dei redirect.

Principali Metodi dell'oggetto window

window è l'oggetto predefinito del DOM, per cui i suoi metodi sono utilizzabili anche omettendo window stesso. Cioè **`window`** può **sempre** essere omesso.

alert("Messaggio") Visualizza una finestra di messaggio con un unico pulsante OK non modificabile.
prompt("Messaggio") Rappresenta il tipico Input Box con i pulsanti OK e ANNULLA. Un secondo parametro opzionale indica un valore iniziale da assegnare al campo di immissione. Clickando su OK restituisce come stringa il valore inserito. Clickando su ANNULLA restituisce **null**
confirm("Messaggio") Finestra di conferma contenente i due pulsanti OK e ANNULLA. Restituisce **true** se l'utente clicka su OK oppure **false** se l'utente clicka su ANNULLA. Il test sul boolean è in genere diretto: if (confirm("Vuoi veramente chiudere?")) { …….. }

parseInt(s) Converte una stringa o un float in numero intero.
In caso di stringa **si ferma automaticamente al primo carattere non numerico**.
In caso di float **il numero viene troncato all'intero inferiore** (come `Math.floor()`)
nota: .value di un input type=number, a differenza di .innerHTML, nel caso di contenuti numerici restituisce non una stringa ma un numero intero.
parseFloat(s) Converte una stringa in float
toString( ); Converte qualsiasi variabile in stringa
NaN Not a Number. Quando si esegue una operazione matematica su una variabile non inizializzata (o non contenente numeri) Java Script restituisce come risultato dell'espressione il valore NaN.
isNaN(s) Consente di testare se la variabile s contiene il valore NaN, nel qual caso restituisce true.

open`("file.html", [target], ["Opzioni separate da virgola"])`
target: **`"_self"`** apre la nuova pagina nella stessa scheda
`"_blank"` apre la nuova pagina in una nuova scheda
A differenza di **`<a href>`** in cui il default di target era **`"_self"`**, questa volta il default è **`"_blank"`**
Come target si può passare anche il nome di un **`iframe`**.
close( ) Chiude la finestra corrente se è stata aperta in locale oppure tramite lo script medesimo.
Se la pagina proviene da un server la finestra **non** viene chiusa: Funziona soltanto per finestre aperte in locale oppure tramite il metodo .open()
scroll(x,y) Fa scorrere la finestra di x colonne orizzontalmente e y righe verticalmente.
focus() / blur() Porta la finestra in primo piano / sotto le altre finestre
self window.self **è** un puntatore alla window medesima (come se fosse un alias)

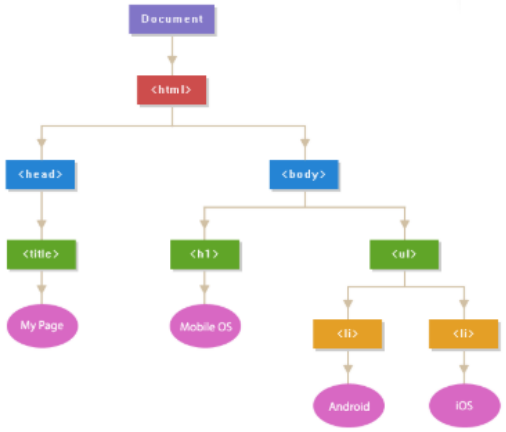
escape(s) Codifica gli spazi e tutti i caratteri divers da (A–Z, a–z, 0–9, _ @ \ * _ + - . /) nei rispettivi codici **utf-8** in formato esadecimale (su 1 o 2 byte a seconda del carattere). Ad esempio lo spazio viene codificat come %20 : **escape ("salve mondo")** diventa "salve%20mondo". Deprecata per **encodeURI(s)**
unescape(s) Opposta rispetto alla precedente. Sostituisce i codici esadecimali con il carattere corrispondente.
Es: "RIS: " + **unescape("%B1")** %B1 è la codifica esadecimale di ± (177 dec \xB1 esa)
str.escapeHtml() Analoga a escape ma esegue una codifica dei caratteri speciali nella relativacodifica **html** (es `<div>`) per poterli andare a scrivere in una pagina html).
Non è una funzione di libreria ma un metodo di istanza delle stringhe; Alternativa a replace

Metodi di evento

onLoad() Richiamato in corrispondenza del caricamento del documento
onUnload() Richiamato quando si abbandona un documento (sostituito da un nuovo documento)
onResize() Richiamato quando la finestra attiva viene ridimensionata

L'oggetto `document` e la gestione degli elementi della pagina

Rappresenta il documento HTML che si sta visualizzando all'interno dell'oggetto window.
Sono figli di **`document`** tutti i tag della pagina html, rappresentabili mediante la seguente struttura **ad albero**:



I cerchi fucsia rappresentano le foglie dell'albero

Proprietà comuni

textContent	rappresenta il solo contenuto testuale di un tag: `"testo"`
innerHTML	rappresenta l'intero contenuto html di un tag `"<p> testo </p>"`
tagName	contiene il nome stesso del tag : `h1, div, input, button`, etc

Eventi comuni

onclick( )	// return **false** abbatte l'evento.
ondblclick()	// poco utilizzato
focus()	metodo che assegna il fuoco all'oggetto
onfocus()	evento generato dopo che l'oggetto ha ricevuto il fuoco
onblur()	evento generato dopo che l'oggetto ha perso il fuoco

I controlli: INPUT, TEXTAREA, SELECT

Proprietà comuni che NON valgono per tutti gli altri tag che non sono dei controlli (DIV, P, etc.) :

disabled true / false. In caso di **disabled=true** il controllo assume un aspetto grigio e non risponde più agli eventi. Vale **SOLO** nel caso dei controlli e **per il tag BUTTON.**

value esiste **SOLO** nel caso dei controlli e rappresenta il contenuto del tag medesimo.
Il tag ***button*** non dispone della proprietà ***value*** ma occorre utilizzare ***innerHTML*** o ***textContent***

Input[type=button]

Scopo di questo pulsante (a differenza dell'input **type=submit**) è quello di richiamare una procedura javascript.

value	testo del pulsante
onclick()	Click sul pulsante. Il metodo **`.click( )`** consente di forzare un click sul pulsante

Text Box

value	rappresenta il contenuto del campo, anche nel caso delle TEXT AREA.
oninput()	generato dopo ogni variazione all'interno del textbox. Richiamato soltanto *DOPO* che il contenuto del TextBox è stato aggiornato. (*vale solo per textbox e textarea, non vede il keyCode del tasto*)
onchange()	generato dopo che è cambiato il contenuto del campo, ma soltanto quando si abbandona il campo.
select( )	metodo che seleziona il contenuto del textbox
onSelect()	generato dopo che il contenuto del campo è stato (anche solo parzialmente) selezionato.

TextArea

Come visto negli appunti di HTML staticamente all'interno di una **textarea** è possibile inserire testo soltanto all'interno del tag e non all'interno dell'attributo **value** (che invece rappresenta il corrispondente **currentState** dinamico utilizzabile **SOLO** da javascript).

Ciò che verrà inserito dinamicamente dall'utente all'interno dell'interfaccia grafica viene dunque salvato nel **currentState** (***cioè nel value***). Se il currentState (value) è vuoto, value restituisce il valore il defaultState (scritto all'interno del tag nella pagina HTML). In javascript:

- la property **`.innerHTML`** restituisce sempre solo il **defaultState** statico
- la property **`.value`** restituisce il **currentState** dinamico che maschera completamente il precedente.

Quindi in js, nel caso della `textarea`, occorre utilizzare SEMPRE SOLO value
Anche in corrispondenza di un submit, ciò che viene trasmesso al server è sempre il value.

Radio Button

Sono mutualmente esclusivi soltanto gli option button con lo **stesso name**, che possono pertanto essere gestiti come un vettore di controlli.

.length	numero di pulsanti appartenenti al gruppo
[i].checked	true / false indica se il radio button è selezionato.
[i].value	Rappresenta il contenuto dell'attributo value (valore restituito in corrispondenza del submit se il radio button è selezionato)
onchange()	Deve essere associato ad ogni singolo radio button e viene generato ***soltanto*** in corrispondenza della selezione del radio button (cioè NON viene attivato quando il radio button viene deselezionato) e ***soltanto dopo che il cambio di stato è avvenuto***
onclick()	Uguale identico al precedente, però viene generato ad ogni click sul radio button, anche se il radio button è già selezionato.
[i].click()	Forza un click sul pulsante che cambia pertanto di stato. Notare che la selezione di un elemento da codice **NON provoca la generazione degli eventi onchange e onclick**, che devono essere esplicitamente richiamati da codice.

In java script NON è possibile associare gli eventi onClick e onChange ad una intera collezione di radio button (come in jQuery) ma occorre assegnare la procedura di evento per ogni singolo pulsante

Per vedere qual è la voce selezionata all'interno di un gruppo di radio buttons occorre pertanto fare un **ciclo** su tutti gli elementi del gruppo e valutare se i singoli attributi **checked** sono true/false
Oppure si può usare **`querySelector`**

```
let chk = document.querySelector("input[type=radio]:checked")
alert(chk.value)
```

CheckBox

Presentano un funzionamento simile rispetto ai Radio Button, con l'unica differenza che non sono esclusivi.
L'evento **onchange()** questa volta viene generato ad ogni cambio di stato, cioè su ogni singolo elemento, sia in selezione che in de-selezione

List Box <select>

Non riconosce placeholder

onchange()	Evento standard richiamato in corrispondenza della selezione di una nuova opzione. Non viene richiamato nel caso in cui da codice si imposti **`selectedIndex=-1`**
onClick	Evento richiamato ad ogni singolo click sulla Lista. Intercetta anche il click di apertura della lista.

Options

_myList.options restituisce una Collection contenente tutte le **options**.

.options.length	Numero di opzioni
.options[i].textContent	Testo visualizzato all'interno dell'opzione ("Fossano")
.options[i].value	Valore Nascosto interno all'opzione ('1')
.options[i].selected	true / false a seconda che l'opzione i-esima sia selezionata o meno
.options[i].onclick()	Generato in corrispondenza del click sulla singola option. Però Attenzione : **Firefox** lo riconosce sempre **Chrome** (febbraio 2024) lo riconosce *SOLTANTO* se l'attributo SIZE è > 0

Principali proprietà dell'oggetto <*SELECT*>

.length numero di `options` presenti nella lista. Equivale a **options.length**

.selectedIndex indice della voce selezionata. Ricordando che size rappresenta il n. di voci visualizzate nella lista

- Se size==0 (default, funzionamento **combo**) selectedIndex viene impostato a 0, cioè sulla prima voce selezionata. Per cui l'evento change sulla prima voce **NON** si verificherà mai. Impostando da codice selectedIndex=-1 viene visualizzata una riga iniziale vuota che scomparirà in corrispondenza della 1 selez.
- Se invece size>0 (lista tradizione aperta) selectedIndex viene automaticamente impostato a -1.

.value L'oggetto <Select> dispone di un comodissimo attributo riassuntivo **value** che, **in corrispondenza della selezione di una voce, contiene il value della <options> attualmente selezionata.**
Se il value è vuoto restituisce automaticamente il valore del contenuto della OPTION (*textContent*)
Notare invece che **option button e check box**, essendo costituiti da un vettore di oggetti **privo di un contenitore esterno** come <select>, non dispongono di una proprietà riassuntiva **value** riferita all'intero vettore, per cui per vedere quale option è selezionato occorre fare un **ciclo** oppure utilizzare :checked.

.selectedOptions Collection delle sole ***options*** selezionate (ricordare che in presenza del'attributo HTML ***multiple*** potrebbero esserci più options selezionate). Strutturalmente è una collection analoga a ***options*** e presenta gli stessi campi

Deselezione della prima voce

Al termine del caricamento la prima voce risulta automaticamente selezionata. Per deselezionarla si può assegnare :

```
_myList.selectedIndex=-1      // oppure
_myList.value=""
```

ma solo DOPO che la lista è stata riempita

sintassi di accesso alla voce selezionata

```
_myList.value                                          // value
_myList.options[_myList.selectedIndex].value           // value
_myList.options[_myList.selectedIndex].textContent     // contenuto testuale
_myList.selectedOptions[0].value                       // value
_myList.selectedOptions[0].textContent                 // contenuto testuale
```

Aggiunta di nuove Opzioni:

```
_myList.innerHTML  +=  "<option value='1'> Fossano </option>"     // oppure
_myList.add(new Option('Fossano', '1'))   // il secondo parametro è facoltativo
_myList.options[i] = new Option('Fossano', '1')
```

Il push NON è ammesso

```
_myList.remove(index)     // rimuove l'elemento indicato
```

Eventi relativi a mouse e tastiera

Eventi relativi al mouse:

onmousedown( ) // Inizio click
onmouseup( ) // Fine click
onmouseover( ) // Ingresso del mouse sull'elemento
onmouseout( ) // Uscita del mouse dall'elemento
onmousemove( ) // Spostamento del mouse all'interno dell'elemento

Eventi relativi alla tastiera:

onkeydown( ) // richiamato al momento della pressione del tasto
onkeypress( ) // Riconosce **Enter** ma non gli altri tasti speciali (Freccine, Backspace, Delete, etc). *deprecato*
onkeyup( ) // richiamato al momento del ritorno su del tasto
onkeydown e onkeyups riconoscono tutti i tasti speciali, compreso **Enter**
onkeydown e onkeypress vengono richiamati ***PRIMA*** che il value del textbox venga aggiornato
e consentono di abbattere il carattere (cioè fare in modo che non 'arrivi' al textbox)
onkeyup viene richiamati ***DOPO*** che il value del textbox è stato aggiornato e **NON** consente di abbattere il carattere

L'oggetto event

A tutte le procedure di evento javascript viene **iniettato** un oggetto **`event`** che contiene diverse informazioni sull'evento stesso e sull'oggetto che lo ha scatenato**.**

- Nel caso dell'onclick eseguito nell'html, il parametro **`event`** **NON** viene iniettato automaticamente e, se lo si vuole utilizzare, occorre scriverlo esplicitamente sia in fase di chiamata sia come parametro della function
  `<button onclick="esegui(event)"> esegui </button>`
- Nel caso di associazione dinamica degli eventi (tramite onclick e addEventListener), il parametro **`event`**, n fase di chiamata, viene iniettato **automaticamente** alla procedura di evento.

La procedura di evento, se vuole utilizzare il parametro **`event`**, lo deve sempre **dichiarare esplicitamente**:

```
function esegui(event) { console.log(event.target.value) }
```

Principali Proprietà del parametro event

`event.target`	puntatore all'elemento che ha scatenato l'evento (derivazione mozilla)
`event.srcElement`	puntatore all'elemento che ha scatenato l'evento (derivazione chrome)
`event.currentTarget`	puntatore all'elemento a cui è associato l'evento
`event.keyCode`	codice ASCII del tasto premuto (con distinzione tra minuscole e maiuscole)
`event.key`	carattere vero e proprio: ad es `"a", "A", "Enter"` (ascii 13)
`event.type`	contiene il nome dell'evento (es "click")
`event.clientX`	coordinate X del mouse rispetto alla window corrente
`event.clientY`	coordinate Y del mouse rispetto alla window corrente
`event.which`	nel caso della tastiera e mouse contiene informazioni aggiuntive sul tasto premuto
`event.timeStamp`	Tempo trascorso (in millisecondi) dal caricamento della pagina

Fra le proprietà del parametro **`event`**, la più importante è **`event.target`** che rappresenta il puntatore all'elemento che ha scatenato l'evento (come il **`this`**) e risulta comodo laddove il **`this`** non è utilizzabile (ad es Angular)

Caratteri speciali

keydown e **keyup**, a differenza di **keypress** (che fra i tasti speciali intercetta SOLO "**Enter**") consentono di intercettare anche tutti gli altri caratteri speciali:

```
"ArrowUp"     "ArrowDown"     "ArrowLeft"     "ArrowRight"
"PageUp"      "PageDown"
"Home"        "End"
"Delete"      "Backspace"
"CapsLock"    "NumLock"
"Shift"
"Alt"
"AltGraph"
"Control"
```

Gli eventi **keydown keypress keyup** possono essere definiti su un textbox oppure direttamente su body, nel qual caso vengono intercettati da tutti gli elementi della pagina

```
<body                onkeyup="elabora(event)" >
<input type="text"   onkeyup="elabora(event)" >
```

```
function elabora(event){
   switch(event.key){
      case "ArrowUp": spostaSu(); break;
      case "ArrowDown": spostaGiu(); break;
```

Come 'abbattere' un carattere digitato da tastiera

1. keypress richiamato dall'html

```
onkeypress="return controlla(event)"

function controlla(event){
   let ascii = event.keyCode
   if(ascii>=48 && ascii<=57)
      return true
   else
      return false
```

2. procedura generale (richiamabile sia dall'html sia da codice)

```
onkeypress="controlla(event)"

function controlla(event){
   let ascii = event.keyCode
   if(!(ascii>=48 && ascii<=57))
        event.preventDefault();
}
```

Meglio della precedente perché non fa uso di **nessun return**

Creazione di nuovi attributi "personali" nascosti

Su tutti i tag HTML è possibile creare, sia staticamente nel file .html sia dinamicamente tramite javascript, dei nuovi attributi '***personali***' nascosti in cui salvare delle informazioni che potrebbero poi essere utili in corso d'opera. Notare che questi attributi **NON** vengono visualizzati dall' inspector.

```
_div.nuovoAttributo = "valore";
```

Aggiunta / Rimozione di una classe

```
_div.classList.add("className1", "className2")
_div.classList.remove("className")
_div.classList.toggle("className", true/false);
// se la classe non c'è la aggiunge, altrimenti la toglie
```

Per vedere se un certo elemento contiene una classe oppure no si usa il seguente metodo:

```
if (_div.classList.contains("className ")) ……...
```

Accesso diretto agli elementi del DOM attraverso i selettori CSS

Il metodo **`selector.children`** restituisce un vettore enumerativo contenente tutti i figli diretti dell'elemento corrente, nel ordine in cui sono scritti all'interno del file html

I metodi **`document.querySelector()`** e **`document.querySelectorAll()`** accettano come parametro qualunque selettore / pseudoselettore CSS **così come potrebbe essere scritto all'interno di un file CSS**.

- Il metodo **querySelector()** ritorna il primo elemento **discendente** (cioè compresi figli e nipoti) che corrisponde al selettore specificato
- Il metodo **querySelectorAll()** ritorna un **vettore enumerativo** di tutti i nodi corrispondenti al selettore.

document.querySelector()

Se si specifica un **ID** diventa sostanzialmente equivalente a `document.getElementById`.
Però attenzione che, a livello sintattico, a differenza di `document.getElementById`, nel caso di `querySelector` occorre anteporre un #, in quanto querySelector si aspetta un selettore CSS

```
let _wrapper = document.querySelector("#wrapper");
```

document.querySelectorAll()

L'esempio restituisce tutti i tag input di tipo **`text`** (non fattibile con getElementsByTagsName)

```
let vet=document.querySelectorAll("input[type=text]")
for (let item of vet)  item.style.backgroundColor="red"
```

Invece di partire da document si può partire da un **`parentNode`**:

```
let _wrapper = document.querySelector("#wrapper");
let vet=_wrapper.querySelectorAll("input[type=text]")
```

Il metodo.contains()

```
if (_div1.contains(_div2))
```

verifica se _div2 è un **`discendente`** di _div1, cioè se _div2 coincide con _div1 oppure è un suo figlio o nipote.

Accesso diretto ai controlli di una form

Per accedere direttamente ai controlli di una form si può utilizzare il **name** della form seguito dal **name** del controllo

```
let nomeUtente = document.form1.txtUser.value
```

Nota: Utilizzando questa sintassi, nel caso di un vettore di radio buttons aventi tutti `name="opt"`, l'istruzione

```
form1.opt.value
```

restituisce il `value` della voce attualmente selezionata. Però vale solo in questo caso.
Utilizzando invece `document.getElementsByName(opt)` non funziona più.

Accesso e gestione degli attributi HTML

E' possibile accedere agli attributi HTML di un elemento in due modi, che però **NON** sono del tutto equivalenti:

1. **accesso diretto** tramite il **puntino** che, partendo dal puntatore all'elemento, consente di accedere a qualsiasi attributo html di quell'elemento
```
let _div = document.getElementById("myImg");
_div.id="id1";
_div.className = "class1 class2"
```
2. metodi **getAttribute( )** **setAttribute( )** e **removeAttribute( ).**
```
_div.setAttribute ("id", "id1");
_div.setAttribute ("class", " class1 class2");
```

Default state (*statico*) e Current State (*dinamico*)

- Gli attributi definiti staticamente all'interno del file HTML rappresentano il cosiddetto ***defaultState*** del controllo, che rimane immutato anche se l'utente a run time modifica il contenuto del controllo attraverso l'interfaccia grafica, scrivendo all'interno di un TextBox o selezionando un CheckBox.
- I valori che l'utente scrive a run time attraverso l'interfaccia grafica oppure attraverso javascript vengono salvati all'interno del cosiddetto ***currentState***, **il cui valore inizialmente coincide con il defaultState, ma poi viene modificato nel momento in cui l'utente scrive qualcosa dall'interfaccia grafica o da codice**.

Lo scopo di questa doppia informazione è quella di mantenere memorizzato il valore scritto staticamente all'interno del file html in modo da poterlo ripristinare in qualsiasi momento.

getAttribute e setAttribute vanno a leggere/scrivere il ***defaultState*** del controllo. Se il valore del campo viene modificato attraverso l'interfaccia grafica oppure da codice, getAttribute() non se ne accorge.
Allo stesso modo se si usa setAttribute() per modificare il defaultState del campo **dopo** che l'utente ne ha modificato il contenuto tramite l'interfaccia grafica, **la modifica NON viene visualizzata sulla pagina**.

L'accesso diretto tramite puntino viceversa accede in lettura / scrittura al ***currentState*****. Molto più comodo**.
Se il ***currentState*** non è ancora stato impostato, in lettura accede automaticamente al ***defaultState***
Indispensabile per tutti i **controlli** che possono essere modificati dinamicamente dall'interfaccia grafica, cioè:

- l'attributo **value** dei textBox sia singoli che multiline
- l'attributo **checked** di checkbox e radiobutton
- l'attributo **selectedIndex** di un tag select che non dispone di un *defaultState* html e dunque è accessibile soltanto tramite accesso diretto con il puntino (così come anche il **value** riassuntivo).
- La stessa cosa vale per il **value** di una textArea, che non dispone di un *defaultState* html.

*Notare che se l'utente non apporta variazioni all'interfaccia grafica (ad esempio se si utilizzano dei checkbox soltanto in visualizzazione per indicare degli stati), setAttribute() potrebbe essere utilizzato anche per i controlli. Però, ad esempio nel caso dei **radio button**, in corrispondenze del setAttribute("checked"), occorre eseguire manualmente un removeAttribute("checked") sugli altri radio button, altrimenti rimangono tutti selezionati.*

E' anche sempre possibile accedere agli attributi dinamici con le quadre : `let id = div["id"]`

Attributi booleani

In HTML gli attributi booleani non accettano valori. Qualunque valore venga assegnato al defaultState, questo valore viene sempre convertito in **`true`** indipendentemente dal valore medesimo.
Scrivendo ad esempio `<button disabled="false">` è come settare a **`true`** l'attributo disabled
L'indicazione di HTML5 è quello di ripetere **eventualmente** il nome dell'attributo medesimo:
`<button disabled="disabled">`

L'unico modo per abilitare il pulsante rimane solo quello di "rimuovere" l'attributo disabled.

```
btn.setAttribute("disabled")
btn.removeAttribute("disabled")
btn.setAttribute("disabled", "disabled")
```

Viceversa il **currentState** è un booleano vero e proprio, per cui lato javascript si può scrivere:

```
btn.disabled = true / false
```

Attributi personalizzati

Gli attributi personalizzati possono essere creati sia in modo statico con setAttribute sia in modo dinamico con il punto

Gli attributi statici creati mediante **setAttribute()** :

- **vengono visualizzati dagli inspector**
- **possono essere utilizzati con querySelector** `.querySelectorAll("input[citta=fossano]")`
- **accettano SOLO stringhe.**
- **Le best practices,** davanti al nome degli attributi personalizzati, **indicano di anteporre il prefisso data-**

Gli attributi dinamici creati mediante il **puntino**

- **non vengono visualizzati dagli inspector**
- **non possono essere utilizzati all'interno di querySelector**
- **accettano come valore interi json.**
- **Anche per gli attributi dinamici si può usare il prefisso data- Il nome dell'attributo verrà tradotto in camelCasing *`div.dataCitta`* E' possibile anche l'accesso con le quadre *`div["data-Citta"]`***

Anche gli attributi del DOM, come ad es **value**, vengono visti dall'inspector SOLO se definiti all'interno del file HTML oppure da js tramite setAttribute, ma non vengono visti se definiti mediante il puntino.

Il prefisso data-

Quando si definisce un nuovo attributo statico (in javascript oppure in jQ) è consigliato di utilizzare il prefisso **`data-`** che fa sì che l'attributo venga completamente ignorato dal browser in fase di rendering della pagina.

```
<div data-student-name='pippo'> student Info </div>
```

Gli attributi creati in questo modo possono essere acceduti tramite l'attributo speciale **dataset** seguito dal nome dell'attributo privato del prefisso **`data-`** I trattini vengono tradotti in camelCasing

```
alert(_div.dataset.studentName)
```

Accesso e gestione delle proprietà CSS

Anche nel caso delle proprietà CSS c'è una distinzione fra le **proprietà di stile impostati staticamente** tramite HTML/CSS e le **proprietà di stile impostate dinamicamente** tramite javaScript.
Quando tramite javaScript si assegna un nuovo valore ad una proprietà di stile, questo valore non sovrascrive completamente il valore statico impostato tramite HTML/CSS, il quale, pur non essendo più visibile, rimane memorizzato come valore statico della proprietà.

- Nel momento in cui viene assegnato un nuovo valore tramite javascript, questo valore "maschera" il valore statico, e qualunque funzione di lettura (compresa getComputedStyle) restituisce sempre il valore dinamico
- Nel momento in cui il valore assegnato tramite javascript viene rimosso (tramite assegnazione di **stringa vuota** oppure **none**), automaticamente viene riassegnato all'elemento il valore statico memorizzato all'interno del file HTML/CSS che non può in alcun modo essere rimosso / modificato

Sintassi di accesso:

1. In modo diretto tramite la proprietà **.style**: `_div.style.backgroundColor = "blue";`
   Questa sintassi rappresenta il modo migliore per modificare singolarmente i vari attributi di stile. L'eventuale trattino nel nome dell'attributo (es `background-color`) deve essere sostituito dalla notazione camelCasing.
   `style` è un object che può accedere ai vari campi sia tramite **puntino** sia tramite **parentesi quadre**
   In lettura restituisce *soltanto* le proprietà impostate dinamicamente tramite java script,
   a differenza degli attributi html in cui, in assenza dell'attributo dinamico, viene restituito l'attributo statico.
   E' chiaramente possibile utilizzare gli attributi composti impostando più valori in una sola riga:
   `_div.style.border = "2px solid black";`

2. I metodi **`.getAttribute("style") / .setAttribute("style")`** sono simili ai precedenti, però leggono / scrivono l'intero attributo **style** in un sol colpo, per cui hanno senso SOLTANTO quando effettivamente interessa leggere / scrivere TUTTE le proprietà di stile insieme.

   `.getAttribute`("style") restituisce tutte le proprietà di stile dell'oggetto corrente, sotto forma di ***stringa serializzata***, però **SOLTANTO** quelle impostate dinamicamente tramite javaScript e **NON** quelle impostate staticamente in HTML/CSS, e nemmeno quelle impostate tramite setAttribute ("class", "className")

   `.setAttribute("style",` `"font-size:100px; font-style:italic; color:#ff0000");`
   consente di definire contemporaneamente più CSS property (con un'unica stringa CSS), però **sovrascrive** tutte le proprietà di stile eventualmente già impostate dinamicamente tramite javascript.

   `.removeAttribute`("style") rimuove tutte le proprietà di stile impostate tramite java script

3. **`_div.style.cssText +=`** `"font-size:100px; font-style:italic; color:#ff0000";`
   Utilizzabile sia in lettura che in scrittura, va bene nel caso in cui si desideri impostare più proprietà insieme.
   In lettura esegue in pratica una serializzazione della proprietà style esattamente come getAttribute("style")
   In scrittura **consente il concatenamento** per cui diventa possibile aggiungere nuove proprietà CSS all'elemento corrente senza rimuovere quelle precedentemente impostate.
   Anche questa proprietà 'vede' soltanto le proprietà di stile impostate dinamicamente in javaScript.

Accesso in lettura agli attributi di stile impostati tramite CSS statico

La funzione **`getComputedStyle(refPointer)`** consente di accedere in **lettura** a TUTTE le proprietà CSS dell'elemento corrente, sia quelle create dinamicamente in JavaScript, sia quelle create staticamente in HTML.

```
if(getComputedStyle(_div).visibility=="hidden")
            _div.style.visibility="visible";
```

- **Se esiste un valor dinamico restituisce il valore dinamico.**
- **Se non esiste un valore dinamico restituisce i valori statici memorizzati nel file CSS.**

Note Operative

1) I **`colori`** vengono restituiti sempre in formato RGB e dunque i confronti vanno fatti sull'RGB

```
if(_div.style.backgroundColor != "rgb(234, 234, 234)")
```

2) **`backgroundImage`** restituisce una url sempre scritta tra apici doppi interni. Es **`url("img/img1.gif")`**
Per cui i confronti vatti sempre utilizzando gli apici doppi interni

```
this.style.backgroundImage = `url("img/img1.gif")`
```

Il collegamento degli Eventi tramite codice

Le procedure di evento possono essere associate staticamente all'interno del file html oppure dinamicamente tramite codice javascript, nel qual caso sono disponibili 2 diverse sintassi quasi equivalenti.

1. Assegnazione diretta della Property **onclick** analogo all'assegnazione inline nel file HTML

```
_div.onclick = esegui
_div.onclick = function(){ let n = random();  esegui(n) }
```

esegui è il **nome** di una funzione che deve tassativamente essere **priva di parametri** e **SENZA parentesi tonde**. Le parentesi tonde provocherebbero il richiamo immediato della funzione senza eseguire nessuna associazione di evento. All'evento onclick verrebbe in pratica assegnato il risultato ritornato da esegui()

- **onclick** deve essere scritto tutto in minuscolo
- La **forma anonima** ha il vantaggio che consente di passare dei parametri ad esegui()
- `_div.onclick=`**null**
  rilascia il gestore di evento che, in corrispondenza dei click successivi, non verrà più eseguito.
  Nel caso dei **controlli** per disabilitare la risposta agli eventi è sufficiente impostare **disabled=true**

2. Utilizzo del metodo **addEventListener** (preferibile) che si aspetta due parametri:

- Il nome dell'evento scritto **senza il prefisso on**
- un puntatore a funzione **priva di parametri.** La funzione anche può essere scritta in forma anonima o come named function, **ma sempre senza parametri**

```
_div.addEventListener("click", esegui )
_div.addEventListener("click",  function(){ esegui(n1, n2) });
_div.addEventListener("click",  function myFunction(){  });
```

`myFunction` sarà però visibile soltanto all'interno di addEventListener

removeEventListener rilascia il gestore di evento che, nei click successivi, non verrà più eseguito.

```
_div.removeEventListener("click", esegui)
```

Il secondo parametro (puntatore alla funzione da rimuovere) è **obbligatorio** e NON può essere omesso

Differenze

Fra le due sintassi ci sono sostanzialmente due differenze:

1) La prima sintassi consente **una unica associazione** di evento. Una eventuale seconda assegnazione sovrascrive l'associazione precedente. **Viceversa tramite addEventListener si possono eseguire più associazioni di evento a funzioni diverse.** Notare che se si eseguono più associazioni ad una stessa named function**, l'associazione viene memorizzata una volta sola, per cui la procedura di evento verrà eseguita una sola volta**. Viceversa ciascuna funziona anonima rappresenta un oggetto diverso, per cui se si definiscono più handler di evento relativi a funzioni anonime differenti, questi handler, verranno sempre eseguiti tutti. Stessa cosa (ovviamente) se, tramite un ciclo, si definisce più volte l'associazione ad un medesimo handler anonimo.

   Per contro nel caso di addEventListener non è possibile rimuovere i listener definiti tramite **funzione anonima**, che continueranno a 'vivere' finchè 'vive' l'oggetto (a meno di usare appositi escamotage)

2) **addEventListener** dispone di un terzo parametro `useCapture` che per default è ***false*** e significa che l'evento viene eseguito durante la bubbling phase (dal più interno al più esterno). Con ***true*** gli eventi vengono invece eseguiti in capturing phase, cioè in ordine inverso rispetto al caso precedente (dal più esterno al più interno).

Note

- Il **(click)** di angular NON è un evento inline come può sembrare, ma viene tradotto con un addEventListener
- **div.style.pointerEvents="none"** rimuove tutti gli eventi relativi al mouse (hover, click, etc)

event

A tutte le procedure di evento dichiarate dinamicamente viene **automaticamente iniettato il parametro event**, che però per poter essere utilizzato dovrà essere scritto esplicitamente come parametro della funzione:

```
_div.addEventListener("click", esegui)
function esegui(event) { event.target.value=n }
// oppure in forma anonima :
_div.addEventListener("click", function(event){event.target.value=n})
```

this

Tutte le procedure di evento dichiarate dinamicamente diventano metodi di evento dell'oggetto che ha eseguito l'associazione, per cui al loro interno, **senza scrivere NULLA** , **diventa possibile utilizzare l'oggetto `this`** che rappresenta un puntatore all'oggetto che ha scatenato l'evento (esattamente come **event.target**).

```
_div.onclick = esegui
_div.onclick = function(){let n=random(); this.value=n}
_div.addEventListener("click", esegui)
_div.addEventListener("click", function(){this.value=n});
function esegui() { this.value=n }
```

event e this nell'html

Nel caso invece dell'utilizzo di onclick nell'html, sia **event** che **this** devono essere passati esplicitamente come parametro. Tra l'altro passando **event**, **this** diventa inutile.

```
<input type="button" onclick="visualizza(event)">
_div.setAttribute("onclick", "visualizza(event)")
```

event e this nelle sottofunzioni

Prestare MOLTA attenzione al fatto che, se la funzione di evento richiama a sua volta un'altra sottofunzione, all'interno della sottofunzione l'associazione di **event** e **`this` NON è più valida.** Infatti la sottofunzione non può sapere chi è che l'ha richiamata e quindi a che cosa si riferisce il **`this`**, Nota: All'interno delle sottofunzioni, **`this`** NON rappresenta l'oggetto che ha scatenato l'evento, ma un generico "spazio" delle funzioni.

In caso di necessità occorre eventualmente passare **event** e/o **this** in modo manuale ed esplicito:

```
_div.addEventListener("click",  function(){ visualizza(this) })
function visualizza (_this)  {
     alert (_this.value); }
```

Il passaggio dei parametri in javascript

I numeri e le variabili scalari in genere vengono **copiati**, **passati** e **confrontati** per valore e, come in java, NON è possibile passarli per riferimento.

Vettori, Matrici e Oggetti sono sempre **copiati**, **passati** e **confrontati** per riferimento e, come in java, NON è possibile passarli per valore.
Il confronto fra due oggetti identici restituisce false a meno che i puntatori non stiano puntando allo stesso oggetto.

Le **stringhe** vengono sempre **copiate** e **passate** per riferimento come gli obiect.
Le stringhe dichiarate con il **new** vengono **confrontate** per riferimento, come tutti gli Obect.
Le stringhe dichiarate senza il **new** vengono **confrontate** per valore.
Se uno o entrambi i valori è una semplice stringa (senza il new), allora il confronto viene eseguito per valore.

Parametri facoltativi nelle funzioni di evento

Le funzioni di evento non possono ricevere parametri (a parte `event`). In fase di definizione è però comunque possibile definire ed inizializzare dei parametri facoltativi che potranno essere utilizzati in caso di chiamate esplicite al fuori dell'associazione di evento

```
function myEventFunction(flag=true){}
```

typeof e istanceof

La funzione **typeof**(`myVar`) restituisce come stringa il tipo di una **variabile scalare**. Esempio:

```
if (typeof(myVar)== "number"  ||  typeof(myVar)== "string"  )
```

Applicata **però** ad una variabile di tipo Oggetto, la funzione **typeof(myVar)** restituisce **sempre** "**object**".

Per verificare di quale object si tratta occorre utilizzare la seguente istruzione che restituisce l'object type come semplice stringa;

```
if (myVar.constructor.name == "FormData")
```

Oppure, in alternativa, si può utilizzare l'operatore **instanceof** che consente di eseguire il confronto direttamente con l'oggetto. A differenza di typeof(), non è però una funzione ma un semplice operatore come es l'operatore `==`

```
if (myVar instanceof FormData)
```

Per vedere se è un array : `if(Array.isArray(myVar))`

Differenza fra var e let

var

Si supponga di avere un elenco di `<button>` all'interno della pagina html e si consideri il seguente codice:

```
var btns = document.getElementsByTagName("button");
for (var i=0; i<btns.length; i++) {
      btns[i].addEventListener("click", function() { esegui(i) });
}
```

La variabile **i**, pur essendo dichiarata all'interno del ciclo for, viene in realtà allocata a livello globale. L'istruzione **var** infatti alloca sempre la variabile **globalmente**. Ad ogni iterazione del ciclo viene creata una associazione tra l'evento "**click**" ed una funzione a cui verrà passato come parametro un riferimento alla variabile globale i, con il valore che avrà al momento del click, cioè `btns.length` (tra l'altro indipendentemente da quale pulsante sia stato premuto). Dunque **non va bene.** Il problema è che la variabile i continua a vivere ed essere utilizzabile anche dopo che il ciclo FOR di creazione degli eventi è terminato, mantenendo sempre l'ultimo valore acquisito

let

L'istruzione **let** dichiara invece una variabile **allocata localmente** nella sezione di codice in cui viene utilizzata. Se ad esempio si utilizza l'istruzione **let** all'interno di un ciclo for, la variabile dichiarata con let non sarà accessibile al di fuori del ciclo medesimo.

Se la variabile i del ciclo precedente venisse dichiarata tramite **let** nel modo seguente:

```
for (let i=0; i<btns.length; i++)
      btns[i].addEventListener("click", function() { esegui(i) });
```

alla funzione esegui(i) verrebbe passata una COPIA del valore corrente della variabile i, cioè al primo btn verrebbe passato 0, al secondo btn verrebbe passato 1 e così via, **per cui il parametro i viene passato correttamente**.

La creazione dinamica degli elementi

In Java Script è possibile creare tag dinamicamente ed aggiungerli all'interno di altri tag utilizzando un tipico modello ad albero. I metodi da utilizzare sono:

- `let ref = document.createElement("tagName") // creo un nuovo tag`
- `parent.appendChild(ref)  // lo appendo ad un tag esistente`

Esempio di creazione di una nuove righe e celle all'interno di una tabella :

```
let tabella = document.getElementById("mainTable");
for (let i=0; i<DIM; i++){
    let tr = document.createElement("tr")
    tabella.appendChild(tr)
    for (let j=0; j<DIM; j++){
        td = document.createElement("td")
        tr.appendChild(td)
        let img = document.createElement("img")
        td.appendChild(img);
        img.id=`img-${i}-${j}`
        img.addEventListener("click", cambiaImmagine)
        let span = document.createElement("span")
        span.innerHTML+="immagine"+j;
        td.appendChild(span);
    }
}
let btn = document.createElement("input");
btn.type = "button"
btn.value = "Elabora"
btn.addEventListener("click", elabora);
```

Per accedere ad un elemento figlio del nodo corrente si può usare .**childNodes[i]** a base 0

```
let div = wrapper.childNodes[2]  // terzo figlio
```

Per vedere se l'elemento corrente ha figli si può usare .**hasChildNodes**

```
if(wrapper.hasChildNodes) .....
```

Note

1) Esiste anche un metodo **.append()** simile ad **appendChild()** ma che consente di appendere più elementi contemporaneamente, separati da virgola. Inoltre, rispetto ad appendChild(), append() accetta come parametro anche stringhe html ma non ritorna il puntatore all'elemento parent comodo per le chiamate in cascata.
2) Nel metodo **parent.appendChild(elem)**, se l'elemento ricevuto come parametro è già appeso al DOM, viene automaticamente 'tagliato' ed appeso nella nuova posizione.
3) Nel caso di **createElement("input")** si può specificare un secondo parametro che indica il tipo di input che si intende creare : **createElement("input", "text")**
4) Per alcuni elementi, in alternativa a `document.createElement()`, è disponibile anche l'operatore **new**
   let opt = new Option(text, value) // Opzione da aggiungere ad un select
5) `document.createTextNode("testo")` crea un nodo testuale che può essere appeso a qualsiasi tag. Analogo all'utilizzo della property **.textContent**

Nota sul concatenamento di stringhe all'interno di `innerHTML`

Supponiamo di avere all'interno della pagina html un tag DIV con **`id=wrapper`** al quale andiamo ad appendere tramite java script altri tag creati dinamicamente. Se ad un certo punto si utilizza una istruzione dei tipo

```
_wrapper.innerHTML += "<br>"     // oppure += qualunque altra cosa
```

il **concatenamento** all'interno della proprietà **`innerHTML`** forza nel browser una **rigenerazione dell'intero contenuto del tag wrapper**, cioè serializza l'oggetto, aggiunge il nuovo contenuto e poi parsifica di nuovo.

Per cui eventuali puntatori javascript agli oggetti creati dinamicamente all'interno di wrapper vengono tutti persi. Insieme ai puntatori vengono persi proprietà e metodi assegnati dinamicamente al puntatore, cioè

- le proprietà HTML assegnate dinamicamente (sia quelle appartenenti al DOM sia quella personalizzate)
- i metodi di evento assegnati dinamicamente tramite addEventListener

questo perché il puntatore usato in fase di creazione per definire l'evento NON punta più al nuovo oggetto ridisegnato all'interno di wrapper.

Per cui, quando si lavora con gli oggetti, occorre abbandonare definitivamente il concatenamento di stringhe. Può essere utilizzata in assegnazione ma NON in concatenamento !! Usare SEMPRE createElement()

Tecniche per la creazione di una matrice di oggetti interni ad un contenitore

Supponiamo di avere un tag statico `<div id="wrapper">` e di volerlo riempire con 10x10 = 100 tag DIV da creare dinamicamente ed appendere a wrapper.

A tale scopo si possono implementare diverse soluzioni:

1) Utilizzare una tabella come nel caso precedente. A questo punto il wrapper potrebbe essere di tipo <table> al quale si aggiungono poi i <TR> e infine i <TD>. Soluzione perfetta per visualizzare tabelle di dati ma poco adatta ai giochi (spaziature indesiderate fra celle e anche fra righe)

2) Utilizzare un contenitore di tipo <DIV> e impostare sugli elementi interni **`float:left`** oppure ancora meglio **`display:inline-block`**. Gli elementi interni vengono disposti uno a fianco dell'altro fino al raggiungimento del margine destro del contenitore, in corrispondenza del quale vanno automaticamente a capo. A tale scopo diventa fondamentale la larghezza del contenitore che deve essere esattamente 10 volte la larghezza degli elementi interni. A volte però (gioco della roulette) il contenitore necessita di un maggior spazio vuoto a destra. In tal caso si potrebbe agire sul padding-right del contenitore

3) Utilizzare un contenitore di tipo <DIV> e con **`position:relative`** ed assegnare **`position:absolute`** agli elementi interni. In questo caso ci svincoliamo dalla larghezza del contenitore

4) Anziché impostare **`float:left`** o **`display:inline-block`** sugli elementi interni, si possono lasciare gli elementi interni così come sono (display:block) e costruire il corpo per righe creando prima un tag <DIV> con **`display:flex`**, e poi aggiungendo al suo interno i 10 tag <DIV> che andranno a costituire la riga, simulando in pratica una tabella. Anche in questo caso ci svincoliamo completamente dalle dimensioni del contenitore.

Accesso alle righe di una Tabella

Il puntatore a tabella presenta una interessante proprietà **`rows`** che rappresenta un vettore enumerativo contenente i puntatori alle varie righe che costituiscono la tabella. A sua volta la riga contiene una collezione di **`cells`**

```
let _table = document.getElementById("table")

_table.rows.count = numero di righe compresi header e footer
_table.tBodies[0].rows.count = numero di righe del tBody
_table.rows[i].cells  = vettore delle celle della riga corrente
_table.rows[i].cells[j]  = cella j-esima
```

```
if(_table.rows.length > 0) {
    let tr = _table.rows[i];
    if(tr.cells.length > 0)
         let cella = tr.cells[j]
```

La gestione delle immagini ed il pre-Caricamento in memoria

Per scaricare una immagine da un server web occorrono mediamente alcuni secondi. Improponibile se questa immagine deve essere utilizzata per eseguire un rollover. A tal fine è possibile, durante il caricamento della pagina, scaricare le immagini necessarie salvandole in memoria. Queste immagini verranno poi visualizzate in corrispondenza di un evento successivo (un click o un mouseOver).

Per creare una singola immagine in memoria occorre utilizzare il costruttore dell'oggetto Image:

```
let img = new Image();
img.src = imageSrc;     // assegnazione dell'immagine
img.onload =  function(){}   // funzione da richiamare a caricamento eseguito
img.onerror = function(){}   // funzione da richiamare in caso di errore
```

Cioè dentro ***.src*** viene salvato il puntatore all'immagine che potrà poi essere utilizzato per copiare l'immagine all'interno di un qualunque altro tag <img> presente all'interno della pagina HTML:

```
_img2.src = img.src
```

Gestione dell'errore

Quando si setta l'attributo **src** di una immagine, automaticamente viene creato un oggetto Image in cui viene caricata l'immagine ed src punterà a questo oggetto Image

Se l'immagine non esiste, lato client viene generato sull'oggetto IMG un evento **error** che può essere gestito con le solite due sintassi javascript:

```
_img.onerror = function () {
    this.src = "default.jpg"
}
_img.addEventListener("error", function() {
    this.src = "default.jpg"
})
```

Attenzione che i nomi dei file contenenti le immagini **NON devono contenere spazi** (che è un carattere non ammesso all'intern di una URL). Se il nome del file contiene uno spazio, questo spazio da codice potrà essere sostituito con %20 (codifica esadecimale dello spazio), che però in talune applicazioni crea problemi.

Effetto RollOver

Il metodo più comune per realizzare un pulsante grafico è quello di includere un tag IMG all'interno di un tag <a>. Sul pulsante grafico si può poi applicare un effetto di RollOver al passaggio del mouse.

Per ottenere questo effetto occorre, al momento dell'onLoad, caricare le immagini in due variabili globali:

```
imgOn.src  = "img/immagine1.jpg";
imgOff.src = "img/immagine2,jpg";
```

caricando staticamente l'immagineOff anche dentro il pulsante grafico imgBox. Dopo di che :

```
onMouseOver="this.src = imgOn.src"
onMouseOut= "this.src = imgOff.src"
```

L'oggetto window: proprietà, metodi ed eventi

name E' il nome assegnato ad una finestra aperta da codice.

closed Dalla finestra attuale è possibile creare una nuova finestra mediante il metodo open ricevendo un puntatore alla finestra. Con questo puntatore è possibile verificare lo stato *closed* della nuova finestra

status Contenuto della barra di stato inferiore. Passando il mouse su un collegamento ipertestuale, il browser visualizza automaticamente nella barra di stato inferiore l'URL completo del link.
Java Script può modificare questo msg mediante l'evento **onMouseOver.** Però se si vuole sostituire l'azione di default con una azione utente, occorre restituire al gestore onMouseOver il valore **true**. Altrimenti l'azione di default maschera l'azione utente.
<a href="pagina3.htm" onMouseOver="window.status='Caratteristiche Tecniche'; **return true**">

defaultStatus Messaggio iniziale visualizzato nella barra di stato dopo il caricamento di una nuova pagina

height altezza della finestra

setInterval

Sia `setTimeout` che `setInterval` sono asincrone, cioè avviano la procedura all'interno di un thread separato, per cui eventuali istruzioni successive a setInterval / setTimeout vengono eseguite subito dopo.

setInterval( ) richiede due parametri:

- Un puntatore a funzione
- Un tempo espresso in millisecondi

Esempio:

```
let timerID=setInterval(visualizza, 1000);
```

La procedura **`visualizza`** verrà richiamata ciclicamente a intervalli regolari di 1000 msec, cioè ogni secondo, in modo analogo all'oggetto timer di C#.

setInterval() restituisce un ID che può essere utilizzato per arrestare il temporizzatore :

```
if(timerID) clearInterval(timerID)
```

Per riavviarlo occorre riscrivere l'intera istruzione setInterval()

Esempio di visualizzazione dell'ora corrente

```
function visualizzaOraCorrente() {
     let d = new Date();
     _div.innerHTML = d.toLocaleTimeString();
}
```

In alternativa si può incrementare una variabile globale `seconds` ad intervalli di 1000 msec.

```
seconds++;
if(seconds%60==0) {minutes++;  seconds=0; }
```

setTimeout

setTimeout( ) è analogo a setInterval() ma la funzione indicata **viene eseguita una sola volta**

```
let timerID =setTimeout(visualizza, 1000)
```

`visualizza` viene avviata dopo 1 sec dal richiamo di setTimeout() e **viene eseguita una sola volta** a meno che, al termine della procedura medesima, venga di nuovo richiamato setTimeout() che la fa ripartire un'altra volta.

Esattamente come setInterval() restituisce un **ID diverso da 0** che consente di disabilitare il timer prima dello scadere del tempo indicato:

```
if(timerID) clearTimeout(timerID)
```

window.open

open("file.htm", ["target"], ["Opzioni separate da virgola"])

Il **primo parametro** indica il file da caricare. Se si specifica "", verrà aperta una nuova scheda vuota.

Il **secondo parametro target** rappresenta la scheda di apertura della pagina e può assumere i valori "_blank", "_self", etc. oppure un nome alfanumerico (**TARGET**) nel qual caso il file verrà aperto in una nuova scheda a cui verrà assegnato il target indicato.

```
<a href="#" onClick='window.open("pag2.htm", "Finestra2");'> apri </a>
<a href='TerzaPagina.html'  target="Finestra2"> vai</a>
```

TerzaPagina.html verrà aperta all'interno della scheda Finestra2 creata da window.open()

Il terzo parametro consente di aprire la pagina **in una nuova finestra** e consente di esprimere le caratteristiche della nuova finestra. In tal caso come secondo parametro si può impostare stringa vuota oppure un target identificativo. I vari parametri devono essere scritti **senza spaziature.**
window.open('pagina2.htm', '', 'resizable=no, width=300, height=300, left=320, top=230, fullsceen=no, menubar, toolbar=no, scrollbars=yes, status=no');
Il valore yes può anche essere omesso scrivendo soltanto il nome dell'opzione.

Opzioni terzo parametro:

Nome	Valore	Spiegazione
width height	Numerico pixel	Larghezza – Altezza della finestra
left	Numerico pixel	Distanza dalla sinistra del monitor
top	Numerico pixel	Distanza dal lato superiore del monitor
fullscreen	yes / no	Apertura a tutto schermo
menubar	yes / no	Presenza del menù
toolbar	yes / no	Presenza della toolbar
scrollbars	yes / no	Presenza delle scroll bar
status	yes / no	Presenza della status bar in basso
location	***yes / no***	***Presenza della barra degli indirizzi***
resizable	***yes / no***	***Ridimensionabile***

location e resizable sembrano deprecati. Al loro posto si può usare il widget dialog di jQueryUi che è basato non su windows.open ma sulle inline dialogs, cioè la visualizzazione di un tag DIV in primo piano con oscuramento della parte sottostante.

Il metodo open viene spesso sfruttato per aprire banner pubblicitari

```
<a href="pagina2.htm" onClick='window.open("banner.htm", " NuovaFinestra ");'> Vai a pagina2 </a>
<body onLoad='window.open("banner.htm", " NuovaFinestra ");'>
```

oppure `body onUnload`

Gestione del riferimento alla finestra aperta da open

Il metodo open restituisce un puntatore alla nuova finestra appena aperta. Esempio:

```
let _div = window.open("pagina2.htm", "NuovaFinestra");
_div .close()    // Chiude la nuova finestra
_div  = null     // Dopo la chiusura di una finestra è bene rilasciare il puntatore
```

La proprietà opener

Ogni finestra (window) ha una interessante proprietà **opener** che è un puntatore alla finestra o frame che ha generato la sottofinestra mediante window.open(). Per la finestra principale opener = null. Esempio:

```
<input type="text" onChange = "opener.document.getElementById().value="x">
```

window. beforeunload

```
window.addEventListener('beforeunload', function () {  }
```

Questo evento viene richiamato quando la finestra corrente sta per essere chiusa (a seguito del click sulla X oppure in seguito alla chiusura del browser), *prima* che il DOM venga deistaziato

- **Consente** l'accesso agli elementi del DOM (non ancora deistaziato)
- **Consente** di rilasciare delle risorse allocate dalla pagina (come ad esempio l'esecuzione di una chiamata Ajax per informare il server della chiusura della pagina)
- **Non Consente** di eseguire delle alert() o aprire finestre di dialogo in genere

Altre Proprietà e metodi dell'oggetto document

Proprietà

title	E' il titolo della pagina impostato nella head dal tag title
lastModified	Data e ora dell'ultima modifica della pagina

Il metodo document.write (s) Consente di scrivere dinamicamente il contenuto di una **nuova** pagina.
All'interno della stringa s può essere inserito qualunque tag html.
Se il metodo viene eseguito verso una pagina già caricata, write troverà il documento chiuso e provvederà a rimuoverlo sostituendolo con un documento vuoto in cui andrà a scrivere il contenuto di s.

Creazione dinamica di un documento tramite document.write()

Dopo aver creato una nuovo finestra vuota tramite window.open()

```
let w=window.open("","_blank");
```

è possibile andare a scrivere dentro utilizzando il metodo window.document.write():

```
w.document.write("<h1 align='center'>Titolo della nuova pagina</h1>");
```

Metodi dell'oggetto document per la scrittura dinamica:

open()	Apre un documento in scrittura. Opzionale. Se il documento è chiuso write lo apre automaticamente
write (s)	Se usato all'interno di una pagina vuota consente di creare dinamicamente il contenuto della pagina. Il contenuto di s viene scritto alla posizione attuale del cursore. **All'interno della stringa s può essere inserito qualunque tag html** compreso \n Il flusso di output viene però automaticamente chiuso al termine del caricamento della pagina. Dunque se il metodo viene eseguito verso una pagina già caricata, write troverà il documento chiuso e provvederà a rimuoverlo sostituendolo con un documento vuoto in cui andrà a scrivere il contenuto di s.
writeln (s)	Come write() con in più il ritorno a capo aggiunto automaticamente al fondo di s
close ()	Serve per chiudere il flusso al termine delle write. Sebbene il flusso venga chiuso automaticamente al termine del caricamento della pagina, i manuali consigliano di eseguire sempre il close() subito dopo l'ultimo write. Altrimenti potrebbero esserci problemi nel caricamento di immagini e moduli

OGGETTO window.location

Contiene tutte le informazioni sulla URL corrente

Proprietà e Metodi

href URL attuale completa http://indirizzo. **Proprietà predefinita di location, per cui può anche essere omessa**. Modificare la proprietà href dell'oggetto location è il modo più semplice per caricare una nuova pagina mediante uno script: `location.href="pagina3.htm"` oppure anche da HTML:

```
onclick="window.location.href='home.html'"
```

Accetta come parametro anche un'ancora interna alla pagina corrente
`window.location.href='#ancora'`

Notare che l'impostazione della proprietà **`href`** **NON termina l'elaborazione dello script** che prosegue eseguendo eventuali istruzioni successive. Per terminare lo script si può utilizzare:

- `return false`; termina la funzione in corso
- **`window.stop()`**; termina l'intero script

Nota: All'interno di href, come in html, si può specificare un indirizzo di posta elettronica, preceduto da mailto: in tal caso verrà aperto il client di posta predefinito. All'indirizzo di posta possono essere concatenati anche dei parametri riguardanti ad esempio il body da preimpostare.

reload( ) Ricarica l'intero documento (come il tasto Reload del browser)
Ha un parametro facoltativo che ha un valore di default pari a **false** nel qual caso il reload viene fatto dalla cache se possibile). Impostando **true** viene forzato il reload dal server.

replace("URL") Carica una nuova pagina nella finestra corrente. Rispetto alla precedente elimina la pagina attuale dalla cronologia. Facendo INDIETRO l'utente non vedrà più la pagina corrente ma ritornerà alla pagina antecedente. Utile per eliminare dalla cronologia pagine intermedie utilizzate in una certa fase.

protocol protocollo di accesso alla risorsa. Es **http**, **file**, **ftp**.
hostname nome del dominio richiesto
port porta di comunicazione. 80 nel caso di http, stringa vuota nel caso del procollo *file*
host hostname : port

pathname risorsa richiesta
search restituisce la ***`queryString`*** della url comprensiva del ?
hash = "Capitolo2" Consente di navigare verso un nuovo ancoraggio presente nella pagina

OGGETTO window.history

Contiene tutte le informazioni relative alle URL visitate prima e dopo rispetto alla URL attuale.
Consente la navigazione avanti e indietro attraverso la storia della finestra corrente.

Proprietà

length Numero di pagine visitate precedentemente rispetto alla pagina attale
current URL della pagina attualmente caricata
previous URL della pagina precedente nella cronologia
next URL della pagina successiva nella cronologia (ha senso solo se si è usato il pulsante back)

Metodi

back()	Ritorna alla pagina precedente. La pagina **viene ricaricata**, però vengono automaticamente ripassati al server eventuali parametri get e post (esattamente come avviene con il pulsante BACK del browser).
forward()	Va avanti di una pagina (se esiste)
go(-1)	Va avanti / indietro ad una posizione ben definita. go(-2) torna indietro di 2 pagine. La pagina **viene ricaricata** con il passaggio automatico dei parametri get e post, esattamente come avviene per il metodo back() e per il pulsante BACK del browser.

OGGETTO navigator

Al momento dell'apertura del browser, viene allocato un oggetto NAVIGATOR, fratello dell'oggetto WINDOW, contenente tutte le informazioni sul browser che si sta utilizzando. Questo oggetto rimane allocato in *unica* istanza fino alla chiusura del browser.

appName	Nome del browser. Es Microsoft Internet Explorer
appVersion	Versione del browser Es versione 4.0 (compatible; MSIE 5.5; Windows 98)
appCodeName	Nome in codice del browser. Es "Mozilla"
userAgent	E' la stringa di intestazione inviata all'host quando gli si richiede una pagina web. Contiene informazioni sul browser, sul sistema operativo e sulle rispettive versioni

Approfondimenti

L'operatore ===

Confronta non solo il valore ma anche il tipo

```
let a = 1;
let b = "1";
if (a==b)   // true
if (a===b)  // false
```

try and catch

```
try {
      alert("Welcome guest!");
}
catch(err) {
      document.getElementById("demo").innerHTML = err.message;
}
```

Parametri opzionali

```
function ricerca(param1 = false) {}
```

Se il chiamante non passa nessun parametro, **param1** viene automaticamente settato a false

Assegnazione di una booelana tramite condizione diretta

```
let ok = (a > 0)
```

Se `(a > 0)` allora ad ok viene assegnato il valore `true`, altrimenti viene assegnato il valore `false`

The ternary conditional operator

E' una tecnica disponibile in tutti i linguaggi per compattare al massimo il costrutto if.

Si supponga di dover eseguire le seguenti assegnazioni:

```
if(ok)  msg = 'yes';
else    msg = 'no';
```

Questo costrutto può essere riscritto in modo molto più conciso nel modo seguente:

```
msg = ok ? 'yes' : 'no';
```

Cioè se la variabile **`ok`** è vera, alla variabile **`msg`** viene assegnato yes, altrimenti viene assegnato no.

Note:

1) Attenzione al fatto che, dopo il ?, occorre necessariamente utilizzare o dei valori diretti (come nell'esempio) oppure delle funzioni che restituiscono un valore. Non è consentito utilizzare delle procedure perché non potrebbero assegnane nessun valore a msg
2) **msg potrebbe anche essere omesso**, nel qual caso il costrutto si limita ad eseguire una delle due funzioni di destra a seconda del valore di ok. Anche in questo caso però a destra non sono ammesse procedure ma sempre soltanto funzioni.

Utilizzo dell'operatore || sulle stringhe

In java script è possibile eseguire una OR fra due o più stringhe.
Il risultato è pari al contenuto della prima stringa che presenta una valore **diverso da undefined**.

```
let ris  = stringa1 || stringa2 || stringa3
```

```
let a;
let b = "hi"
console.log(a)      // undefined
console.log(a||b)   // "hi"
```

Accesso diretto agli elementi del DOM

E' possibile accedere direttamente a tutti gli elementi del DOM attraverso il loro ID

```
window["btnIndietro"].disabled=true;
```

Allo stesso modo è possibile accedere anche alle variabili globali:

```
let a = 15;
window["a"]++;
alert(a);       // 16
```

DEFER

L'attributo **defer** indica al browser di caricare lo script "in background", parallelamente al caricamento della pagina, e poi eseguire lo script quando è caricato.

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

- Gli script con defer non bloccano il caricamento la pagina
- Gli script con defer vengono sempre eseguiti SOLTANTO al termine del caricamento del DOM, subito prima prima dell'evento window.onload()
- L'attributo defer viene ignorato se il tag <script> non ha l'attributo src.

In sostanza l'attributo **defer** rappresenta una interessante alternativa all'evento **window.onload()**

ASYNC

L'attributo **async** indica al browser di caricare lo script "in background" (esattamente come defer) ma in questo caso lo script verrà eseguito appena pronto. Il DOM e gli altri script non restano in attesa che questi vengano caricati. Avremo quindi un script completamente indipendente che verrà subito eseguito al termine del caricamento.

- L'evento DOMContentLoaded potrebbe scattare sia prima che dopo rispetto ad uno script async, dipende dalla pesantezza dello script.
- Allo stesso modo gli script async non si aspettano a vicenda. Uno script più piccolo come small.js inserito nella pagina per secondo probabilmente verrà caricato prima di long.js e quindi anche eseguito per primo. Questo ordine viene chiamato "load-first".

Gli script async sono ottimali quando dobbiamo integrare uno script di terze parti indipendente: contatori, ads, e così via, visto che essi non dipendono dai nostri script e i nostri script non devono aspettare il loro caricamento:

Script dinamici

E' anche possibile aggiungere uno script dinamicamente tramite JavaScript:

```
let script = document.createElement('script');
script.src = "myscript.js";
document.body.append(script);
```

Lo script inizia a caricarsi nel momento in cui viene appeso a document. Il caricamento è esattamente come quello degli script "async", cioè:

- Non aspettano nessun altro script e nessuno aspetta loro.
- Lo script che viene caricato per prima viene anche eseguito per primo (ordine "load-first")

E' possibile cambiare l'ordine "load-first" nell'ordine con cui vengono richiamati (come per i normali script) settando l'attributo async a false:

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  script.async = false;
  document.body.append(script);
}

loadScript("long.js");
loadScript("small.js");
```

Senza script.async=false gli script verrebbero eseguito secondo l'ordine load-first (small.js probabilmente per primo). Ma con il flag `async = false` l'ordine diventa "come nel documento", per cui long.js verrà eseguito prima di small.js

Lettura di parametri non dichiarati nella firma

E' anche possibile definire una funzione con **firma priva di parametri** e poi andare a leggere eventuali parametri all'interno dell'oggetto **arguments** Esempio:

```
visualizza("pippo", "pluto", "minnie");
function visualizza(){
    let result = '';
    for (let i = 0; i < arguments.length; i++)
       result += arguments[i] + "\n";
    alert(result);
```

Lettura dei parametri GET

```
function leggiParametriGet(){
    let json = {};
    let parametri = [];
    let s = window.location.search;
    // estraggo dal punto interrogativo in avanti
    s = s.substr(s.indexOf("?") + 1);
    // sostiuisco %20 con " "
    let exp = new RegExp("%20", "g");
    s = s.replace(exp, " ");
    parametri = s.split("&");

    let parametro = [];
    for (let i = 0; i < parametri.length; i++)
    {
      parametro = parametri[i].split("=");
      let key = parametro[0];
      let value = parametro[1];
      // Se il nome del parametro termina con [], compatto i valori in una stringa
      if(key.substr(key.length-6, 6)=="%5B%5D"){
            key=key.substr(0,key.length-6);
            if (!(key in json))
                  json[key] = value;
            else
                  json[key]+=", " + value;
      }
      else
            json[key] = value;
    }
    return json;
```

Un sito di rapido test del codice

www.webtoolkitonline.com