JavaScript

Introduzione

La dinamicità che occorre nella modifica della visualizzazione dei contenuti esistenti all'interno di una pagina web avviene tramite l'inclusione, all'interno di queste pagine, di veri e propri programmi scritti in JavaScript; programmi che verranno poi attivati, ad esempio, in funzione dell'interazione che l'utente realizza sulla pagina e che determineranno un cambiamento nelle modalità di visualizzazione della pagina stessa.

Nell'architettura basata sulle single-page applications, la nostra pagina web, una volta caricata sul client, per effetto del codice (tipicamente JavaScript) che in essa è contenuto si trasforma in una vera e propria applicazione, che genera (senza interrogare il server) il contenuto delle altre pagine del sito, caricando dei contenuti che il browser va a recuperare dal server sotto forma di dati che vengono direttamente iniettati nel layout di pagina che viene direttamente gestito dal codice JavaScript scaricato all'inizio di questo processo.

JavaScript è un vero e proprio linguaggio di programmazione (a differenza di HTML che è un linguaggio di Markup), attraverso il quale possiamo codificare algoritmi che ci permettono di manipolare lato client (cioè browser) il contenuto di una pagina web per renderla dinamica dal punto di vista della visualizzazione. Lato client si produce il contenuto attraverso un vero e proprio processo elaborativo (programma) e si rigenera il contenuto ogni volta che c'è una chiamata.

Caratteristiche

JavaScript è un linguaggio interpretato che consente di codificare programmi all'interno di pagine HTML.

Interpretato sta a significare il suo traduttore è basato su uno schema noto come **interprete**, alternativa all'altro schema possibile noto come *compilatore (vedi sopra)*.

> ***Compilatore*** → prende il sorgente, lo trasforma prima in codice oggetto poi in eseguibile. Infine l'esecutore (modulo software del SO) manda in esecuzione l'intero codice.

> ***Interprete*** → Esegue in parallelo traduzione del codice ed esecuzione del codice tradotto. Opera istruzione per istruzione a partire dal codice sorgente.

La differenza tra interprete e compilatore è che la traduzione (cioè la trasformazione di un'istruzione scritta ad un linguaggio di alto livello nella corrispondente istruzione eseguibile dall'hawdware della macchina sulla quale viene eseguito il codice) nel caso del compilatore avviene in una fase che precede la fase di esecuzione vera e propria. Nel mondo dei linguaggi compilati è gestita da un'applicazione diversa da quella che traduce. Il compilatore prende il codice sorgente scritto in un certo linguaggio di programmazione e trasforma il codice sorgente in un codice oggetto e successivamente in un codice eseguibile che l'esecutore (un modulo software incluso nel sistema operativo) manda in esecuzione.
Con gli interpreti il traduttore di tipo interprete non si limita a tradurre producendo un codice che qualcun altro esegue, ma esegue parallelamente la traduzione e l'esecuzione del codice stesso. L'interprete opera istruzione per istruzione a partire dal codice sorgente traducendo ed eseguendo subito dopo ogni singola istruzione del programma.

All'interno del nostro browser c'è bisogno di un modulo di traduzione del linguaggio stesso che verrà attivato dal browser stesso nel momento in cui scandendo il codice della pagina HTML che ha ricevuto per poter creare sulla finestra di visualizzazione (newport) l'immagine corrispondente ai contenuti si troverà a gestire un codice in JavaScript, in questo momento l'interprete del browser passa il controllo dall'interprete HTML a quello JavaScript che è un altro modulo core presente in tutti i browser.

È stato sviluppato a partire dal 1995 dalla Netscape per Navigator, ma è supportato anche da Explorer e dagli altri browser. Il codice JS può essere eseguito dal lato Server o dal lato Client (cioè dal Browser, e di questo tratteremo nel corso. Adotta il paradigma di programmazione ad oggetti utilizzando concetti quali: Classi, Oggetti, Proprietà, Metodi

Funzionalità (lato client)

Con JS possiamo:

Con JS non possiamo:

- Controllare le funzioni del browser (apertura di nuove pagine, messaggi di alert, ...)
- Modificare aspetto e contenuto delle pagine Web
- Registrare ed usare informazioni sugli utenti (cookies)
- Creare e manipolare elementi grafici nelle pagine
- Manipolare immagini
- Accedere (in lettura e/o scrittura) ai file sulla macchina client
- Utilizzare tutti i protocolli Internet, ma solo un sottoinsieme limitato (http, ws, ...)
- Operare in multithreading, cioè attivare in modo concorrente diversi processi di elaborazione

Cookies:

Piccoli contenuti d'informazioni che il browser registra all'interno di un suo spazio di memoria permanente per mantenere la conoscenza di alcune caratteristiche del browser stesso così come viene configuarato dall'utente durante la navigazione.

Scavalca la caratteristica di essere senza memoria dell'interazione fra client/server basata sul protocollo HTTP. (Es: apri google per la prima volta e ti chiede che lingua vuoi usare).

Quando un client accede a un server web il server nella risposta può inserire una stringa di testo che contiene le informazioni sulle caratteristiche del client come la lingua. Questa informazione codificata sotto forma di stringa viene memorizzata in maniera permanente dal browser e ogni volta che il browser accede allo stesso server inoltra dopo la prima volta al server stesso oltre alla richiesta anche questo biscottino, il quale costituisce l'elemento che consente al server di ricordarsi che chi gli sta chiedendo qualcosa è un client a cui ha già fatto accesso che ha caratterizzato registrando un cookie sulla lingua da utilizzare e quindi consente al server di configurare la risposta in relazione alle esigenze del client. Possiamo manipolare lato client i cookies utilizzando delle primitive del linguaggio javascript.

Codice JavaScript

Per includere in codice JS in un documento HTML, si può:

- Inserirlo direttamente all'interno del tag `<script>...</script>`
- Utilizzare lo stesso tag per includere nel documento un file esterno che contiene il codice

Il codice JS può essere collocato in qualsiasi sezione del documento HTML (`<head>` o `<body>`). Quando l'interprete HTML individua una tag script cede il controllo all'inteprete JavaScript il quale esegue il codice e se il codice produce un contenuto quel contenuto diventa un contenuto della pagina che va a sostituire il codice stesso nella stessa posizione in cui il codice è collocato.

È buona regola inserire anche contenuti da visualizzare nei browser che hanno JS disattivato, usando il tag `<noscript>...</noscript>` (avvisa l'utente che una parte dei contenuti non è visualizzata perché non supportata).

<script>

`<script>Codice script</script>`

Descrizione: inserisce nel documento codice codificato in uno script-language

Tipo: tag contenitore

Attributi: `type` , `src` , ...

- `type` : definisce il linguaggio di script utilizzato (default se non specificato: `text/javascript`)
- `src` : è l'URL del file che contiene lo script. Possiamo separare il documento HTML dal documento che contiene codice JavaScript e quindi mantenere una certa pulizia concettuale.

▼ **Esempio JS#1**

https://codepen.io/fabiocarosi/pen/BapLaOQ

Il metodo `writeln` prende come parametro una stringa e produce come contenuto HTML la stringa che gli passiamo.

Variabili e tipi di dato

Analizziamo variabili e tipi di dato che utlizzeremo per la rappresentazione delle nostre strutture dati, che poi manipoleremo per effetto delle procedure per sviluppare i nostri algoritmi. (Non ha molto senso la frase)

Variabili

- Identificatore di variabile: `<varid>::` `$` | `_` `|<letter>{<letter>|<digit>}`
  Cioè il nome con cui potremo far riferimento alla nostra variabile. Un identificatore può essere:
  - Solo il simbolo `$`
  - Solo il simbolo `_`
  - Una sequenza di una lettera, *eventualmente* seguita da altre lettere o digit.
    Per lettera si intende un qualunque simbolo del codice ASCII, ma anche *$* e _ , quindi un nome di variabile può iniziare con `$` e proseguire con una sequenza di caratteri o cifre, può iniziare con `_` e proseguire con una sequenza di caratteri o cifre oppure può iniziare con una lettera (ma non con una cifra) e proseguire con una sequenza di caratteri o cifre.
- L'identificatore è *case sensitive* (*Tavolo ≠ tavolo*)
- La dichiarazione viene fatta contestualmente all'assegnazione:
  - `var` seguito dal nome della variabile e l'operatore per assegnare un valore: `var indice=10` ;
  - Avrei anche potuto definire la variabile senza il valore: `var indice` ;
  - Si possono fare dichiarazioni multiple: `var indice=10, Testo="buongiorno"` ;
- Una variabile può essere dichiarata subito prima di essere usata, non serve dichiarare tutte le variabili all'inizio del codice.
  - Posso anche definire una variabile senza dichiararla: `indice=10` , JavaScript verifica se nel contesto locale o nel contesto globale è stata dichiarata la variabile `indice` : in caso affermativo viene considerata come l'assegnazione del valore della variabile definita altrove, in caso negativo, considera quell'istruzione come la dichiarazione globale della varibile stessa.

Binding

Una variabile è un oggetto associato a cinque proprietà:

- Nome
- Valore
- Tipo (di dato)
- Tempo di vita
- Campo di azione

Il legame che associa queste cinque proprietà alla varibile è detto **binding**, che può essere, *per ognuna delle proprietà*, dinamico o statico.
Per binding
statico (a *compile-time*) si intende un legame definito nel momento della compilazione del codice e che rimane costante (immutabile) durante il tempo di esecuzione del programma.
Per binding
dinamico (a *run-time*) si intende invece un legame che può cambiare durante l'esecuzione.

Ogni linguaggio ha diversi binding per i vari legami, alcuni sono tipici come il binding statico tra nome e variabile. Il binding varibile-valore è tipicamente dinamico, ma in alcuni casi statco (in JS è dinamico).

Il Binding variabile-tipo in JS è dinamico, una varibile può cambiare il suo tipo durante l'esecuzione:

```
var miaVariabile = 10; //int
        miaVariabile = "Testo"; //int -> string
        miaVariabile = 3.14; //string -> float
        miaVariabile = true; //float -> boolean
```

quindi, in sede di dichiarazione, alla variabile non è associato un tipo di dato, esso è associato dinamicamente a seconda del valore.

Tipi di dato

Tipo	Operatori
1. Numero (Int, Real)	1. `+`, `-`, `*`, `/`, `%`, `++`, `--`, `-` (opposto), `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `==`, `!=`, `>`, `<`, `>=`, `<=`
2. Booleano	2. `&&`, `||`, `!`
3. Stringhe	3. `+`, `+=`
4. Array	4. Dipendenti dal tipo degli elementi

5. Oggetto 5. `new` (Es: `var cinquecento = new auto`)

Internamente a JS, tutti i numeri sono reali, quindi con mantissa e caratteristica. Le stringhe sono un tipo di dato predefinito.

Conversione di tipo

Derivante dal binding dinamico variabile-tipo, posso convertire facilmente.

Conversione stringa → numero

```
Y = "3";
Y *= "7";
// Y = 21
```

Questo è possibile quando non viene applicato un operatore sovraccarico (cioè se lo stesso simbolo è un operatore per tipi di dato diversi) a due operandi. Dato che `*=` è un operatore solo per il tipo `int` , JS convertirà le stringhe in interi.

Conversione numero → stringa

```
Y = 3;
Y += "tavoli";
// Y = "3tavoli"
```

Questo è possibile quando viene applicato un operatore sovraccarico (cioè se lo stesso simbolo è un operatore per tipi di dato diversi) a due operandi. Dato che `+=` è un operatore sia per `int` che per `string` , JS convertirà l'intero in stringa.
Si converte in quello che tra i due è il **super-tipo**: `string` contiene i caratteri degli interi, ma non il contrario, quindi `string` è il super- tipo.

Array

Sono oggetti, e vengono quindi creati con la sintassi: `var <nome_variabile> = new Array([<num_elem])`

Possiamo definire elementi del vettore di tipo diverso:

```
var vettore = new Array(4);
        vettore[0] = "bianco";
        vettore[1] = "nero";
        vettore[2] = 3;
        vettore[3] = 3.14;
```

Per effetto della dinamicità tra variabile e tipo un `Array` può modificare facilmente le dimensioni, semplicemente aggiungendo altri elementi al suo interno. Anche la dinamicità dei tipi dei singoli elementi rimane. Una sintassi alternativa (preferibile) per gli `Array`:

```
var vettore = ["bianco", "nero", 3, 3.14]
```

Scope di una variabile

Il campo di azione è anche detto **scope** della variabile. In JS le variabili possono essere:

- **Locali**, definite all'interno di una funzione ed utilizzabili solo in essa (e nelle eventuali funzioni definite al suo interno)
- **Globali**, definite all'esterno delle funzioni e visibili in ogni parte del programma

Lo scope definisce quindi in quale parte di codice quella variabile è visibile o meno. Lo scope, in JS, è STATICO (come in C).

Funzioni e Oggetti

Funzioni

Dichiarazione: `function <nome_funzione> ( [<parametri>] ) { <istruzioni> }`

```
function somma (a,b) {  // a e b sono parametri formali
        var risultato = a + b;
        return risultato;
 }
```

Attivazione:

```
var c = somma(3,7); // 3 e 7 sono parametri attuali
```

Per chiarezza del codice, *preferibilmente* la dichiarazione va inserita nella `<head>` del documento, mentre l'attivazione nel `<body>`

▼ **Esempio JS#2**

https://codepen.io/fabiocarosi/pen/poREzBE

Classi

Una Classe è un prototipo che definisce le proprietà ed i metodi degli oggetti che da essa discendono.
In JS un oggetto può essere creato attraverso l'uso della funzione costruttrice (che, di fatto, definisce la classe da cui l'oggetto discende)

```
function libro (isbn, titolo, autore) { // libro è la class
e
 this.isbn_libro = isbn;
 this.titolo_libro = titolo;
 this.autore_libro = autore; }
mio_libro = new libro ("111-2222", "Div_comm", "D_Alighier
i") // mio_libro è l'oggetto
```

Una volta definito l'oggetto possiamo utilizzare la notazione `mio_libro.autore_libro`

Estensione di Oggetti e Classi

- Estensione delle proprietà degli oggetti, lavorando a livello di singola istanza.
  Aggiungo la proprietà "prezzo" al singolo oggetto `mio_libro`, anche ne non presente nel costruttore iniziale. Tutti i libri hanno tre proprietà, solo `mio_libro` ne avrà quattro.

```
mio_libro = new libro ("111-2222", "Div_comm", "D_Alighier
i");
mio_libro.prezzo = "12 euro";
```

- Estensione delle proprietà delle classi. Aggiungo la proprietà "prezzo" all'intera classe `libro`, tutti gli oggetti generati (sia prima che dopo) questa dichiarazione avranno quattro proprietà.

```
libro.prototype.prezzo = "12 euro";
```

Se voglio aggiungere una nuova classe, invece che sostituirla basterà fare:

```
function libro_con_prezzo () {
 this.prezzo = "12 euro";  // definisco la proprietà mancan
te
}
libro_con_prezzo.prototype = new libro(); // eredito le pro
prietà della classe libro
```

Abbiamo quindi due classi e la possibilità di generare due tipi di oggetti a seconda se vogliamo un libro con o senza prezzo.

Metodi di una classe

I metodi sono funzioni definite nell'ambito di una classe che si applicano a tutti gli oggetti che da essa discendono.
Una funzione JS diventa metodo di una classe inserendo all'interno della funzione costruttrice di quest'ultima l'istruzione:
`this.<nome_metodo> = <nome_funzione>`.

▼ **Esempio JS#3**

https://codepen.io/fabiocarosi/pen/yLgaNgy

▼ **Esempio JS#3.1**

Variante sintattica per dichiarare le classi e l'ereditarietà ("zucchero sintattico", variante formale), quando l'interprete trova queste varianti le riconduce alla forma base.

https://codepen.io/fabiocarosi/pen/gOgwpvZ

Oggetti 'Singoli'

In JS è possibile definire oggetti che non discendono da alcuna classe. Ad esempio se necessito di una sola istanza, non avrebbe senso definire una classe per una singola istanza. Non necessito del `this` perchè so che è riferito

a questa istanza. Dichiaro il metodo senza necessità del nome, perchè la utilzzo contestualmente.

```
var mio_libro = {
        isbn_libro:"111-2222",
        titolo_libro:"Div_comm",
        autore_libro:"D_Alighieri",
        stampa_prop: function () {
                document.write(this.isbn_libro + ", " +
                this.titolo_libro + ", " + this.autore_libr
o);
        }
}
```

Strutture di controllo condizionale ed iterativo

If

```
if (<espressione_condizional
   <istruzioni>;
}
```

If ... else

```
if (<espressione_condizional
   <istruzioni>;
}
else {
  <istruzioni>;
}
```

Switch

```
switch (<espressione>) {
   case <etichetta>: <istruzioni>; break;
   case <etichetta>: <istruzioni>; break;
   …
   default: <istruzioni>;
}
```

Di solito `<espressione>` non è altro che il valore di una variabile di tipo carattere o intero, e le `<etichette>` sono i possibili valori assunti da questa variabile.

While

```
while (<espressione_condizio
   <istruzioni>;
}
```

do ... while

```
do {
   <istruzioni>;
}while (<espressione_condizi
```

Le `<istruzioni>` vengono ripetute fintanto che `<espressione_condizionale>` è vera.

For

```
for (<inizializzazione>; <co
   <istruzioni>;
}
```

for ... in

```
for (<variabile> in <oggetto
   <istruzioni>;
}
```

`for … in` scandisce le componenti di un oggetto, (proprietà e metodi) associandone i nomi ai valori di nel corso dell' iterazione.

▼ **Esempio JS#4**

https://codepen.io/Romandini/pen/vYgXRzB?editors=1000

With

Ci consente di semplificare il codice che dobbiamo scrivere per accedere alle componenti (desiderate) di un oggetto.

```
with (<oggetto>) {
   <istruzioni>;
}
```

Esempio :

```
mio_libro = new libro();
with (mio_libro) {
   isbn_libro = "11-222-33";
   titolo_libro = "La Divina Commedia";
   autore_libro = "Dante Alighieri";
}
```

Break, continue

- `break` : consente di interrompere un ciclo iterativo prima che la condizione di fine ciclo si sia verificata.
- `continue` : permette di passare direttamente alla successiva iterazione di un ciclo.

Gestione degli eventi

Eventi

Gli **eventi** sono condizioni rilevate dal browser e relative ad azioni che :

- l'utente realizza in fase di visualizzazione di un documento
- il browser stesso realizza sul documento

Siamo nella fase in cui il browser web, dopo aver richiesto al server una pagina ed aver ricevuto il codice HTML della pagina stessa, ha provveduto alla renderizzazione della pagina stessa attraverso questi meccanismi che si basano sul concetto di blocco e quindi sta presentando all'utente il contenuto della pagina. Nel momento in cui si completa la fase di visualizzazione, si attiva una fase nella quale il browser si mette in ascolto degli eventi che possono essere generati al termine del processo di visualizzazione stessa, e questi eventi possono essere generati o dal browser stesso al verificarsi di certe condizioni associate alla gestione della visualizzazione della pagina, oppure possono essere generati dall'utente.

Nel primo caso, gli eventi generati dal browser sono per esempio l'evento di termine del caricamento della pagina (quando il browser completa il suo processo di renderizzazione, segnala a se stesso il termine di questa procedura

sotto forma di un evento che può essere intercettato da un codice JavaScript che può determinare ciò che va fatto al termine di caricamento della pagina). Per esempio, nelle pagine che generano (al termine del caricamento) delle altre finestre che appaiono in modalità pop-up con la pubblicità, questo tipo di azione (l'apertura di pagine al termine del caricamento della pagina che abbiamo esplicitamento richiesto di caricare) viene gestita intercettando l'evento di fine caricamento pagina che il browser definisce e associando a questo evento un codice JavaScript che riapre ad altre *n* pagine, caricando su questa pagina contenuti che sono interessanti per chi ha prodotto questa pagina.

Più vicino alla nostra esperienza è il concetto di eventi generati dall'utente: quando noi leggiamo una pagina, col mouse ci spostiamo sui contenuti della pagina e li puntiamo oppure ci clicchiamo sopra o digitiamo dei caratteri con la tastiera, ognuna di queste azioni che l'utente compie di fatto è associata ad un evento che il browser riconosce e che può associare (al verificarsi dell'evento) ad un processo elaborativo ben definito; processo elaborativo che è un programma che realizza un algoritmo e che è tipicamento realizzato utilizzato il linguaggio JavaScript.

Meccanismo generale: al verificarsi di un evento che il browser riconosce, il browser stesso cerca di individuare se nel codice della pagina è presente un riferimento a quell'evento sotto forma di codice JavaScript che determina il programma che va eseguito al verificarsi dell'evento stesso. Questo programma è noto come **EventHandler**, il gestore dell'evento.

Come fa il browser ad associare un evento ad un handler, cioè ad un pezzo di codice ?

Io non posso dichiarare su un pezzo di codice (tipicamente una funzione JavaScript) che questa funzione va eseguita quando l'utente clicca su questa icona; il browser costruisce un meccanismo che consente di associare gli eventi agli elementi della pagina, e questo processo si basa sulla capacità del browser di riconoscere una serie di azioni (click del mouse, spostamento del puntatore, pressione di un tasto, ...) contestualmente agli elementi HTML che sono contenuti nella pagina.
Cioè è un meccanismo che consente al mouse di capire che noi abbiamo cliccato sull'immagine che sta nella pagina che sta sotto al primo paragrafo della pagina stessa e sopra alla tabella che segue l'immagine stessa.
Analogamente consente al browser di capire che è stato premuto un tasto

mentre il puntatore del mouse era posizionato all'interno del primo paragrafo dell'area in cui il browser stesso visualizza il primo paragrafo della nostra pagina ipotetica (costituita da un paragrafo, da un'immagine e da una tabella); così come il mouse è in grado di riconoscere che in un certo momento il puntatore del mouse è posizionato all'interno di una qualunque delle celle della nostra tabella.
A questo punto, stante la coscienza che il browser può avere (perché il browser ha la contezza di quelli che sono gli elementi strutturali che compongono una nostra pagina), quello che manca come componente per innescare il processo di esecuzione di un codice al verificarsi di un evento su un certo elemento, è lo strumento che consente, nel momento in cui noi programmatori definiamo il cotenuto della nostra pagina web, di dire :
"se quando questa pagina viene visualizzata, l'utente clicca all'interno di questo paragrafo (in un qualunque punto del block-box che identifica questo pararafo), esegui questa istruzione".
Ci manca cioè uno strumento che ci consenta di associare
il che cosa fare al verificarsi di un certo tipo di evento su un certo elemento della pagina.

Eventi principali

La maggior parte dei Tag HTML consente di attivare funzioni JS al verificarsi di tali eventi riferiti alle parti di documento da esse definite.

Come facciamo a dire noi programmatori della pagina HTML che vogliamo che quella funzione JavaScript venga eseguita al verificarsi dell'evento "click" su quel particolare paragrafo della pagina HTML ?

Lo possiamo fare attraverso una serie di attributi HTML mette a disposizioni come attributi *generici*, cioè che possono essere applicati a qualunque Tag HTML, che identificano il tipo di evento che in sede di visualizzazione va intercettato su quella componente HTML per la quale specifichiamo l'attributo.
Un'attrbuto in HTML ha quasi sempre un valore (tranne quelli booleani).

onBlur	onChange	onClick
onDblClick	onFocus	onKeyDown
onMouseDown	onMouseMove	onLoad
onMouseOver	onMouseUp	onMouseOut
onSelect	onSubmit	onReset
...	...	...

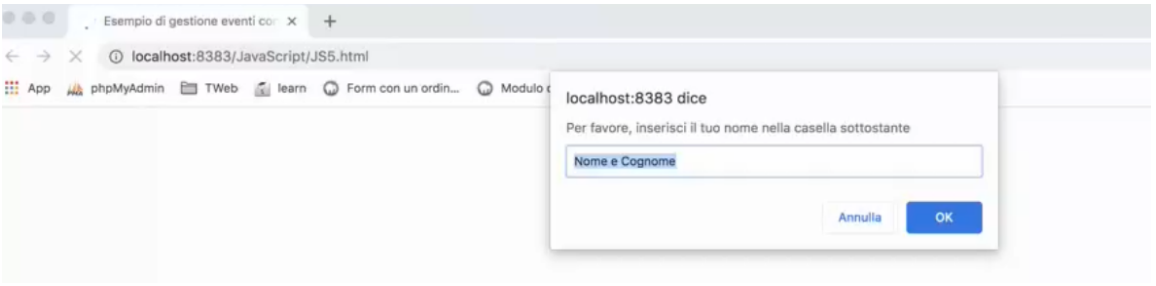
Questo set di attributi sopra è un set di attributi con valore, e il valore che questi attributi assumono è codice JavaScript, è il programma JavaScript da eseguire al verificarsi dell'evento che l'attributo identifica sull'elemento HTML della pagina per cui è definito l'attributo stesso.

- `onClick` : consente di assegnare una procedura elaborativa (codice JavaScript) al verificarsi del click del mouse sull'elemento HTML per cui l'attributo stesso è stato specificato.
- `onChange` : consente di associare una procedura JavaScript al cambiamento del valore di un elemento di input di una form.
- `onFocus` : identifica l'evento di acquisizione del *focus* di un elemento di una form. Il *focus* è il posizionamento del cursore di inserimento caratteri all'interno dell'elemento.
- `onKeyDown` : l'evento di click (tenuto premuto) sull'elemento per cui l'attributo è definito.
- `onDblClick` : il doppio click sull'elemento.
- `onBlur` : l'evento di perdita del focus di un elemento; è l'evento che si realizza quando, dopo aver inserito un valore in un campo di input di una form, io sposto il focus in un altro campo della form stessa, e quindi è l'evento che segnala il completamento della fase di inserimento del dato da parte dell'utente in un certo elemento di input della form.
- `onMouseDown` : il mouse passa sotto l'elemento.
- `onMouseOver` : il mouse passa sopra l'elemento.
- `onMouseOut` : il mouse esce dall'elemento.

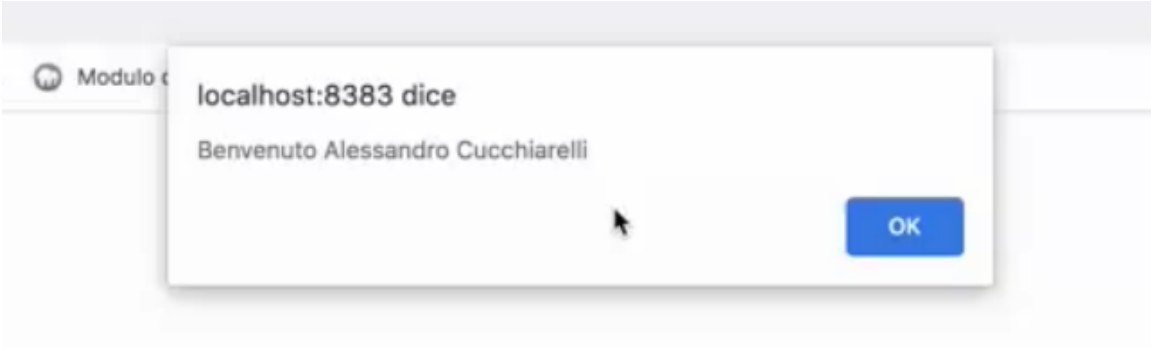
- `onLoad` : evento che identifica il termine del caricamento da parte del browser della visualizzazione della pagina, cioè il termine del processo che ha consentito al browser (a partire dal codice HTML che ha ricevuto dal server) di visualizzare la pagina stessa.
- `onSubmit` : click sul bottone "*Submit*" di una form.
- `onReset` : click sul bottone "*Reset*" di una form.
- `onSelect` : selezione di una opzione all'interno di una form.

▼ Esempio JS#5

Durante la fase di caricamento della pagina si apre questa finestra *modale,* una finestra cioè che interrompe l'esecuzione dei processi del browser in attesa che l'utente completi l'informazione che va inserita nella finestra e clicchi o sul bottone "Annulla" o sul bottone "OK". *localhost:8383* è il server da cui la richiesta arriva.



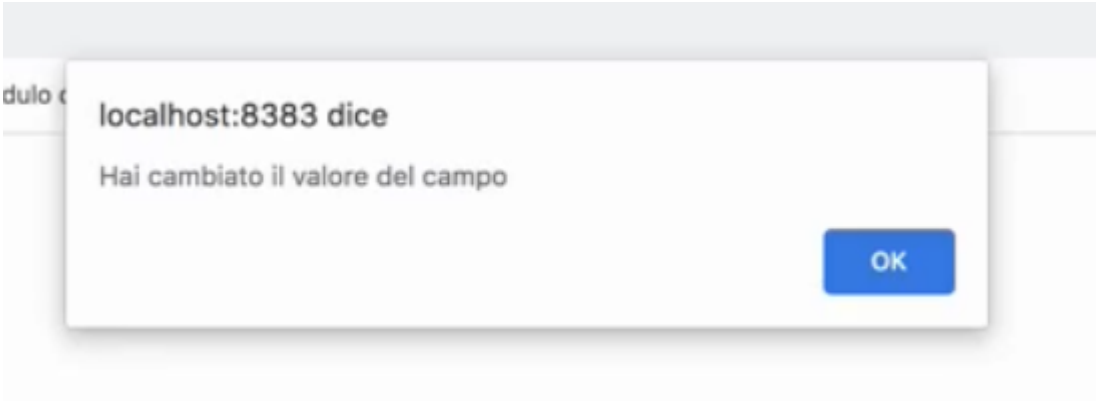
Una volta inseriti i dati richiesti appare una nuova finestra :



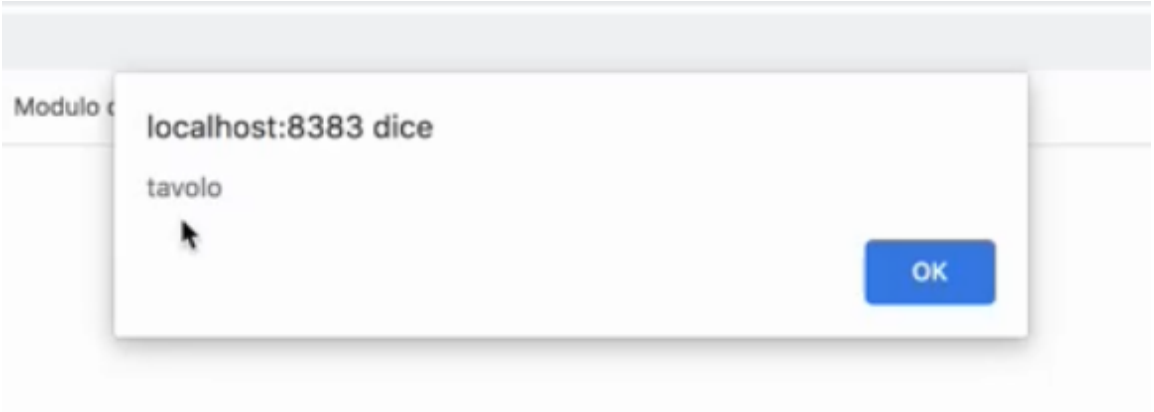
Infine, premuto il bottone "OK", si procede alla visualizzazione della pagina (come riportato nel CodePen).

https://codepen.io/Romandini/pen/QWdKBKE?editors=1010

La perdita del focus dell'elemento di input, oppure la modifica del valore inserito all'interno del campo, determinano la produzione di un altro messaggio da parte della pagina :



Una volta rimepito il campo e premuto il bottone "Visualizza il contenuto del campo", appare la seguente finestra (ho inserito la stringa "*tavolo*" all'interno del campo) :



Browser Object Model (BOM)

Quelle che abbiamo visto in precedenza sono azioni che vanno ad agire non sul file HTML che è stato inviato dal server e ricevuto dal client ed è stato interpretato e trasformato in una rappresentazione interna al browser sulla quale il browser stesso può operare attraverso l'esecuzione di codice JavaScript che consente di manipolare la struttura della pagina HTML.

In altri termini,
quando una pagina viene caricata dal browser, il contenuto della pagina (espresso in codice HTML) viene trasformato in una serie di strutture dati (nella memoria del browser) che poi sono quelle a partire dalle quali il processo di visualizzazione (e anche il processo di visualizzazione dinamica dei contenuti) viene attivato.

Questo modello di rappesentazione delle informazioni che il browser utilizza nella sua operatività è noto come **BOM** (*Browser Object Model*) : al termine del caricamento del documento HTML che riceve, il browser definisce una serie di oggetti a struttura predefinita nell'ambiente di esecuzione JavaScript che rappresentano i contenuti della pagina stessa.

Gli oggetti principali sono :

- *Document* : modello in memoria del browser che rappresenta i contenuti della pagina.
- *Location* : rappresenta l'URL della pagina che è visualizzata nella finestra del browser che stiamo analizzando.
- *History* : definisce la storia di navigazione tra le diverse pagine che su quella certa pagina si è realizzata.
- *Navigator* : definisce le proprietà del browser (nome, versione, ...) che sono utili nel momento in cui la pagina si deve adattare a browser diversi. Entra in gioco per esempio quando la nostra pagina contiene delle regole di stile che sono diverse per i diversi browser.

Sono oggetti JavaScript che vengono valorizzati dal browser all'atto di caricamento della pagina e che poi possono essere manipolati attraverso codice JavaScript che noi inseriamo all'interno della nostra pagina.

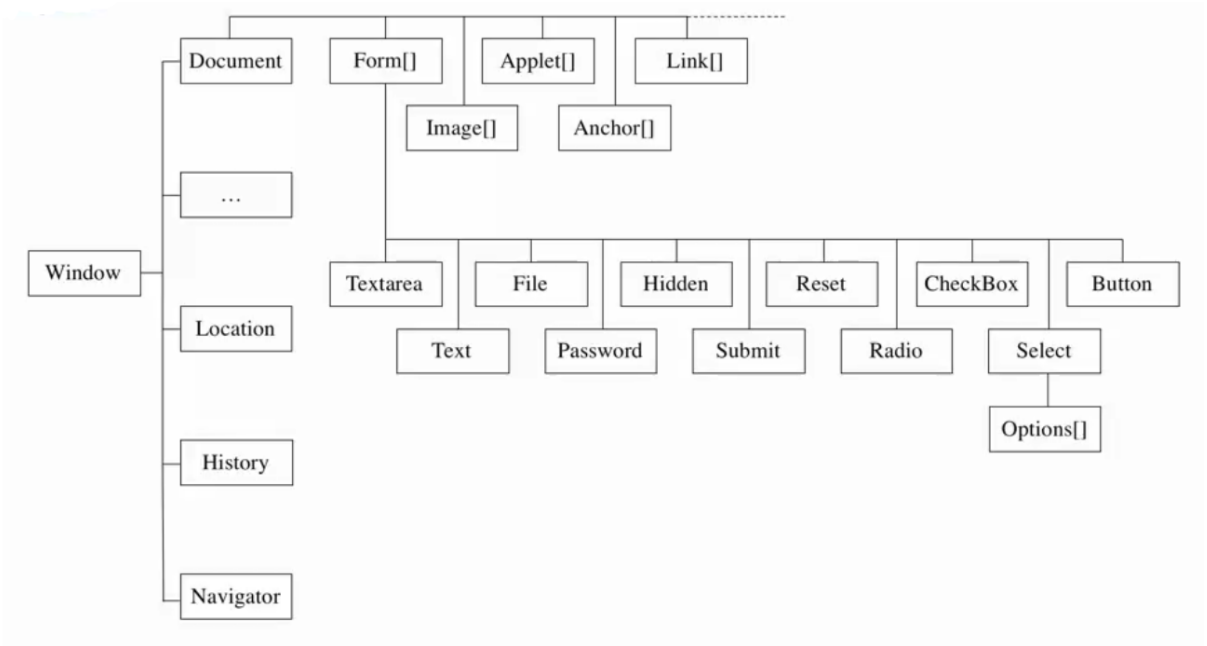
Il Browser, al termine del caricamento di un documento, associa una serie di oggetti JS al contenuto della pagina.

A questi oggetti si può accedere tramite script JS per manipolare il contenuto della pagina visualizzata.

Gerarchia oggetti JS

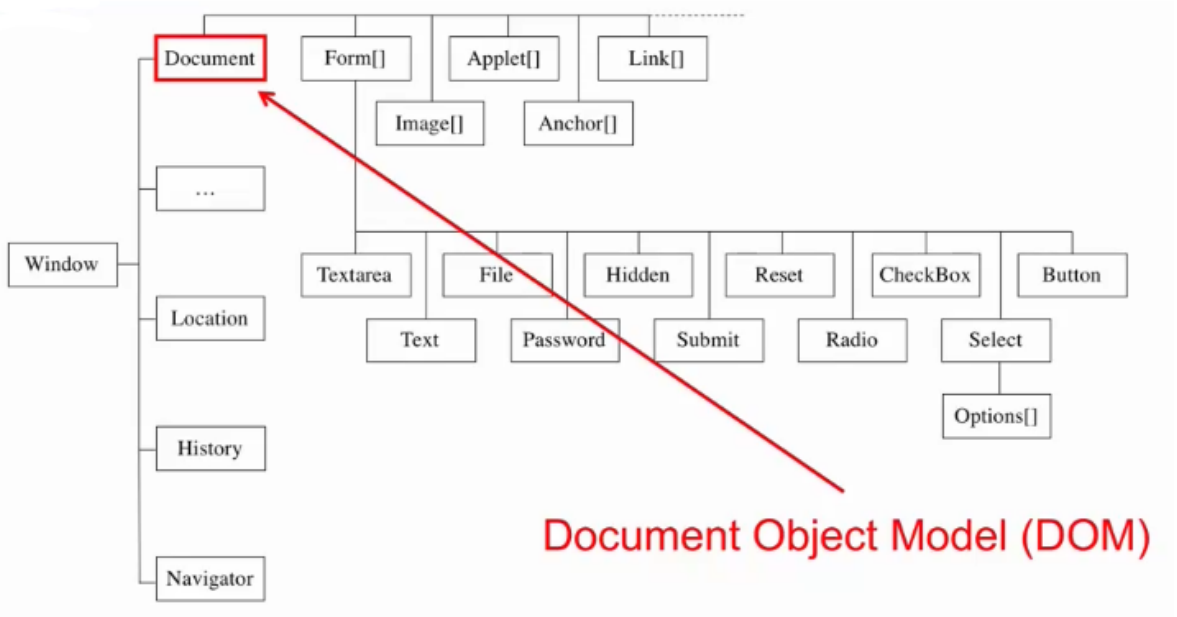
Tra gli oggetti definiti nel *BOM* esiste un meccanismo di legame di tipo gerarchico, strutturato ad albero.

Quella riportata in foto è la struttura del BOM definito ed implementato nei primi browser; attualmente la struttura del BOM è molto più complessa.



Questa struttura ha come radice *Window* che rappresenta la finestra (o la scheda) del browser. Tutto quello che sta sotto all'oggetto Window rappresenta il contenuto della finestra stessa.

Document è l'oggetto componente di Window (il figlio di Window) che descrive il documento contenuto nella pagina, cioè descrive tutte le componenti che abbiamo espresso nel file HTML che sono contenute nelle sezioni body ed head del nostro oggetto.
Nell'oggetto Document esistono delle sotto-componenti predefinite per rappresentare tutte le form contenute nella mia pagina; la maggior parte di questi elementi è indicata come un array sotto forma di oggetto, perché in un documento non è detto che ci sia un unico oggetto form, oppure image, oppure ancora, o applet. Ognuno di questi oggetti a sua volta può essere articolato in sotto-componenti.
Ogni elemento che può essere inserito all'interno di una pagina HTML (che è associato ad una Tag HTML) in questo schema generale viene associato ad un singolo oggetto; ogni volta che il browser carica una pagina, al termine del caricamento mappa i contenuti espressi dall'HTML in un'istanza di questo albero.

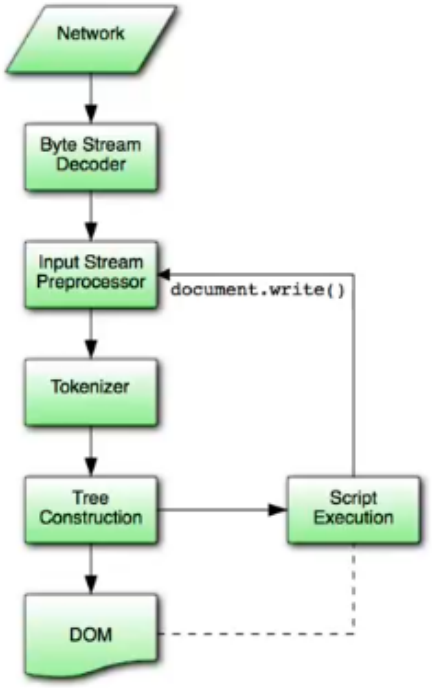


L'oggetto Document è noto come **DOM** (*Document Object Model*).

Negli esempi precedenti, quando creiamo un riferimento a `document` , in realtà stiamo facendo riferimento all'oggetto Document che il browser ha creato interpretando il codice HTML che gli è stato fornito dal server.

Poichè la radice dell'albero non è Document ma Window, allora ci si aspetta che il riferimento dichiarato prima sia scritto come `window.document` ; ma poiché tutti gli oggetti discendono da Window, JavaScript ci dà la possibilità di non specificare l'oggetto radice (che è sempre lo stesso, Window).

HTML Parsing - Creazione DOM



- **Byte Stream Decoder**
  Individuazione della codifica usata nel documento per i caratteri (UTF-8, ISO-8859-5, …)
- **Input Stream Preprocessor**
  Conversione dei codici in caratteri UNICODE
- **Tokenizer**
  Accorpamento dei caratteri in termini (parole)
- **Tree Construction**
  Creazione della gerarchia (albero) del DOM

Dal network il browser riceve il documento HTML oggetto della risposta che il server invia al client a seguito di una richiesta del client.

Il documento HTML passa per un primo modulo che si occupa del *Byte Stream Decoder,* ovvero quel processo che fa sì che la sequenza di 0 e 1 venga trasformata in una sequenza di codici che rappresentano il contenuto della pagina da interpretare. Questa prima fase di analisi dipende dal tipo di codifica che viene usata per la rappresentazione dell'informazione all'interno del file e ,in funzione del tipo di codifica, vengono scelte delle logiche diverse con cui associare le sequenze di 0 e 1 dei codici che rappresenteranno i caratteri contenuti nel nostro file. La principale codifica usata è l'UTF-8, che associa i caratteri degli elementi del file che verranno interpretati al livello di HTML a sequenze di byte che vanno da un minimo di 1 a un max di 4. Un singolo carattere della codifica UTF-8 può essere rappresentato da una sequenza di 8 o 32 byte. Noi abbiamo bisogno di trasformare la sequenza binaria in un carattere che segue le specifiche Unicode, che definisce il modo in cui una sequenza di byte viene associato a un carattere all'interno di una tabella predefinita che è la tabella Unicode, estensione della tabella ASCII, che contiene circa un milione di differenti simboli.

Queste sequenze di byte vengono passati all'*Input Stream Preprocessor*, che prende queste sequenze di byte e le associa ai caratteri corrispondenti secondo la tabella Unicode.

All'uscita di questo blocco abbiamo una sequenza di caratteri che va trasformata in una sequenza di "token" e se ne occupa il *Tokenizer* dove non abbiamo più <, H, T, M, L > ma abbiamo <html>.

Con questo output utilizzato dal modulo *Tree Construction* per creare il DOM della pagina. Se durante la fase di costruzione del DOM viene individuato da parte dell'htmlparser una porzione di codice JavaScript questo codice viene mandato in esecuzione e se produce un output questo ritorna nella fase di *Input Stream Preprocessor* perché i caratteri prodotti vengono associati a simboli unicode e poi a token che vanno a comporre il contenuto dinamico della pagina.

Proprietà e oggetti del DOM

WINDOW

Proprietà	Metodi
name	alert()
parent	close()
self	confirm()
status	focus()
top	open()
...	prompt()
	...

Proprietà:

- *name*: nome associato alla finestra (title in html)
- *parent*: definisce se la nostra finestra è generata da un'altra finestra quale la generata
- *self*: riferimento alla finestra stessa
- *status*: proprietà che caratterizza la barra di stato
- *top*: all'interno di una gerarchia ci dice qual è quella che le ha generate tutte

Funzioni elaborative:

- `close()` : consente di chiudere in maniera programmatica la finestra stessa oltre che un'altra finestra che è stata generata dalla finestra che esegue il metodo close
- `confirm()` : messaggio in popup che contiene frase con due bottone di accettare o rifiutare
- `focus()` : metodo che mette in primo piano la finestra che in quel momento specifichiamo come parametro
- `open()` : consente a una finestra di aprirne un'altra

▼ Esempio JS#6

https://codepen.io/Romandini/pen/poREObV?editors=1000

Una volta premuto sul bottone, si aprirà un'altra scheda contentente un'immagine, come riportato sotto :

Dopo 5 secondi tale finestra contenente l'immagine si chiuderà automaticamente.

DOCUMENT

Nella gerarchia degli oggetti DOM (accessibile da JavaScript), l'oggetto Document è quello che caratterizza il contenuto del documento visualizzato nella finestra del browser. Costituisce una mappatura in memoria di tutti gli elementi strutturali del file HTML che il browser ha caricato e che si appresta a

visualizzare. Dentro l'oggetto Document troviamo le form, le immagini, le liste, le ancore, ... , tutto ciò che contiene il documento HTML stesso.

Le proprietà principali sono:

Proprietà	Metodi
lastModified	write()
title	writeln()
URL	createElement()
forms	getElementById()
images	getElementByName()
...	querySelector()
	...

Proprietà:

- *lastModified*: consente di stabilire (se c'è la comunicazione necessaria tra server e client) qual è la data di modifica del domumento.
- *title*: titolo associato al documento
- *URL*: URL associato al documento
- *forms, images, ...* : rappresentano i vari singoli elementi stutturali del nostro documento

Le proprietà possono essere viste come variabili che assumono dei valori.

Funzioni elaborative:

- `write()`: genera una stringa
- `writeln()`: genera una stringa terminato con un carattere di 'a capo' → in realtà viene interpetrato come uno spazio
- `createElement()`: ci permette di creare un elemento nella pagina (paragrafo, immagine, tabella...) direttamente a livello di DOM e non a partire dal file. Posso quindi manipolare la pagina iniziale continuando a eliminare e creare elementi direttamente dal livello del DOM.

- `getElementById()`: Ci consente di selezionare un qualunque elemento strutturale della pagina attraverso l'attributo *id* (qualora sia definito) dell'elemento stesso.
- `getElementByName()`: Ci consente di selezionare un qualunque elemento strutturale della pagina attraverso l'attributo *name* dell'elemento stesso.
- `querySelector()`: Ci consente di selezionare un qualunque elemento strutturale della pagina attraverso il selettore CSS dell'elemento stesso. Ad esempio gli elementi `<h1>` a partire da loro selettore.
  Nota: se il selettore seleziona più elementi (ad esempio passando la tag `<p>`) in questo caso ritorna il primo elemento del selettore selezionato

▼ Esempio JS#7

Variante dell'esempio JS#5. Introduce una prima tecnica di programmazione avanzata in JS che ci consentirà di muovere di spostare il codice JS dalla sezione <html> alla sezione <script>, eliminando l'uso degli elementi *"on"* (onchange, onclick, ...) all'interno del codice HTML stesso. Comportamento analogo alla divisione dei mondi tra CSS e HTML, al fine di mantenere il codice leggibile e pulito.

Nel DOM ogni elemento ha anche, rappresentate come proprietà dell'oggetto, gli attributi `onchange` e `onclick` (piuttosto che gli altri attributi "on") che rappresentano la controparte in memoria sottoforma di modello ad oggetti degli attributi HTML `onchange` e `onclick` (piuttosto che gli altri attributi "on").

Ho la possibilià di includere all'interno del codice JS associato alla pagina un'istuzione che, senza bisogno di andare a definire a livello HTML l'attributo `onclick`, vada a settare lin maniera programmatica la proprietà corrispondente all'attributo `onclick` HTML nel DOM (che è una proprietà dell'oggetto `input type="button"` in questo caso). In questo modo scompare l'attlributo `onclick` a livello HTML e compare la chiamata alla proprietà `onclick` all'interno del codice JS del programma.

È importante l'accortezza che questo tipo di operazione debba essere fatto solo dopo che il documento è stato caricato e quindi che è stato costruito il DOM; solo quando è terminata la fase di interpretazione del codice HTML avremo che il documento avrà una sua corrispondenza in memoria, e quindi all'interno del mio DOM sarà presente l'elemento e l'attributo.

https://codepen.io/Romandini/pen/KKaQdwV?editors=1000

Non c'è più una funzione `benvenuto()`, questo viene sostituito dalla funzione `onload`, assegniamo alla proprietà `onload` il risultato della funzione `function()` corrispondente alla vecchia funzione `benvenuto()` eliminando il codice JS all'interno dell'HTML. Analagomente abbiamo fatto la stessa cosa per `onchange` e `onclick`.

La stringa che scrivo all'interno della text box viene aggiunta al documento e non è infatti presente nel codice HTML.

▼ Esempio JS#8

Variante dell'esempio dell'ordinazione, con l'aggiunta di una richiesta di conferma al momento del click sul bottone "Annulla".

https://codepen.io/Romandini/pen/VwPQvwB?editors=1000

▼ Esempio JS#9

Cliccando sui vari header appaiono o scompaiono dei contenuti senza la necessita di comunicazione tra client e server. Inizialmente setto l'attributo CSS `display` a `none` per poi modificare l'attributo al momento del click del mouse, con logica binaria (appare/scompare). All'interno del codice i vari sottocapitoli sono già presenti e visualizzabili all'interno degli stumenti per sviluppatori nell'elenco degli elementi.

https://codepen.io/Romandini/pen/MWJQawb?editors=1000

Ogni elemento del DOM che corrisponde ad un elemento contenuto nel file HTML è un oggetto che ha predefinito una proprietà `style`, questa proprietà è a sua volta un oggetto che mappa come sue proprietà tutti gli attributi CSS associati all'elemento della pagina.
Dopo aver trovato l'oggetto
`style` dell'elemento cliccato, vado a modificare l'attributo `display`

dell'oggetto `style` dell'elemento cliccato, rendendolo da visibile a invisibile e viceversa.

▼ Esempio JS#10

Esempio sull'uso del `querySelector()`.
Tabella sensibile al passaggio del puntatore del mouse, primo evento intercettato.
Il secondo evento è la gestione del click sulle varie celle della tabella.

> https://codepen.io/Romandini/pen/yLgveqQ?editors=1000

Tra i selettori CSS abbiamo visto i selettori `E:link` e `E:visited` che rappresentano rispettivamente gli elementi con tag `E=<a>` relativi ad un link non ancora visitato o già visitato. Un selettore analogo (`E:hover`) ci consente di individuare gli elementi sui quali è posizionato il puntatore del mouse.
Il colore ciano al passaggio del puntatore si ottiene sempliemente con delle regole CSS.

Attraverso il `querySelector("td:hover")` possiamo evitare di inserire un metodo `onclick` per ogni elemento. Se un si verifica un evento in una cella (l'utente clicca) l'evento si verifica sull'intera cella (dato che la cella fa parte della tabella). Associamo un unico evento all'intera tabella, ma si estrae poi il valore della cella cliccata attraverso `querySelector()`

`document.querySelector("td:hover").innerHTML`: contenuto dell'elemento HTML che viene selezionato dal selettore `td:hover` del nostro documento. `td:hover` identifica la cella (`td`) sulla quale era posizionato (`hover`) il puntatore del mouse al momento del click.

FORM

Scendiamo ancora di livello Window → Document → Form

Proprietà	Metodi
action	reset()
method	submit()
name	
elements[]	
length	
...	

Proprietà:

Sono attributi della type form che vengono trasformate in proprietà:

- *action*: attibuto della type form, definisce la risorsa lato server che va attivata passandogli i valori della form per l'eleborazione lato server dei contenuti stessi
- *method*: metodo con cui i dati della form vengono inoltrati
- *name*: valore dell'attributo name della type stessa
- *elements[]*: array nel quale vengono memorizzati gli elementi che contiene la form
- *lenght*: numero di elementi che la form contiene

Metodi:

- `reset()`: cambiare il modo in cui la nostra form risponde al bottone di reset
- `submit()`: cambiare il modo in cui i dati vengono inoltrati al server

▼ **Esempio JS#11**

Al momento del click sul bottone "Ordina" i dati non partono direttamente al server, bensì vengono estratti e visualizzati in una box.
In questo caso
`onsubmit` visualizza è sempre `false`, non ci interessa inviare i dati.

Estraiamo i dati di ogni coppia nome-valore della form, nel caso in cui ci sia una check box estraiamo i dati solo se spuntato, inoltre eliminiamo i valori di

Submit e Azzera che non ci interessano. Attraverso `elements[i]` scandiamo tutti i vari elementi aggiungendo ogni volta l'elemento desiderato in `datiOut`

https://codepen.io/Romandini/pen/oNBEbPb?editors=1000

MATH

È un oggetto predefinito JS al quale sono associate (come metodi) le funzioni matematiche di JS.

Proprietà	Metodi		
E	abs()	exp()	random()
LN10	acos()	floor()	round()
LN2	asin()	log()	sin()
PI	atan()	max()	sqrt()
SQRT2	ceil()	min()	tan()
...	cos()	pow()	...

▼ **Esempio JS#12**

https://codepen.io/Romandini/pen/dyNdGjM?editors=1000

▼ **Esempio TicTacToe**

Programmato per non perdere. Questo documento non ha bisogno di interazioni con il server.

Struttura pricipale

Header di livello 1 con il titolo, una tabella con 9 celle numerate da 0 a 8, un pulsante per resettare la partita e una `<div>` dove sarà visualizzato dinamicamente il risultato

Logica generale

Creare un handler che attende il click su una cella da parte dell'utente, a quel punto l'handler segnerà il simbolo dell'utente su quella cella, registrerà che quella cella è occupata dell'utente, deciderà la mossa del computer, registrerà che quella cella è occupata dal computer e visualizzerà la mossa del computer e poi si fermerà attendendo un altro click. Tutto all'interno dell'handler della `function()` corrispondente a `onclick`

https://codepen.io/fabiocarosi/pen/QWdmJoN