Richiami di Java Script

Rev 2.1 del 10/11/2025

Richiami sui principali metodi js di accesso al DOM

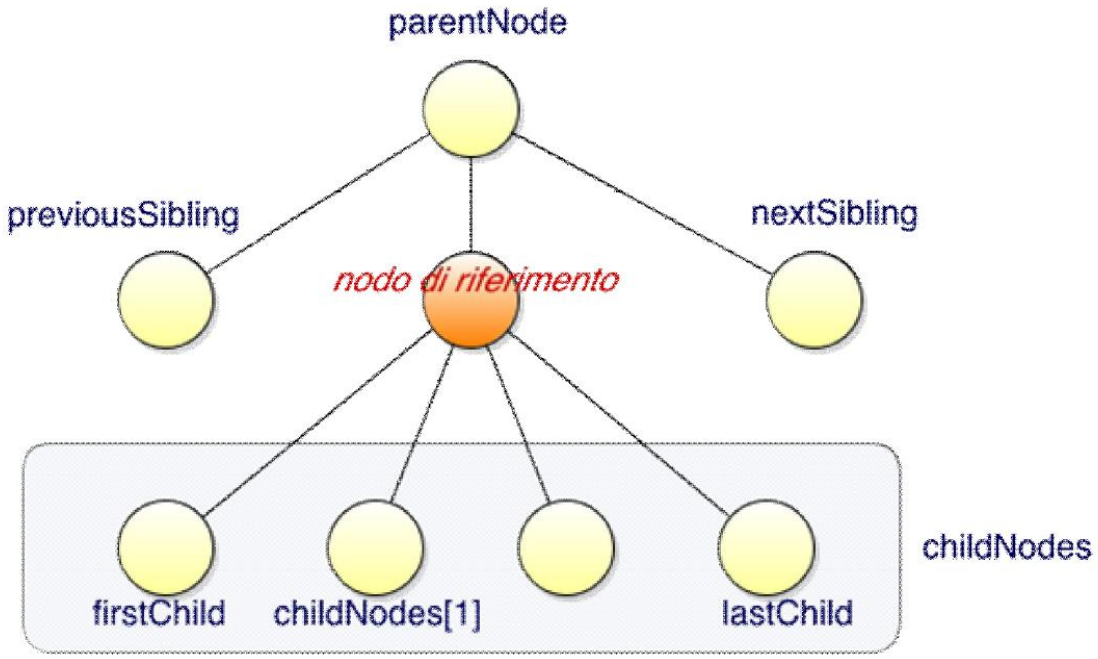
`document.`**`getElementById`** restituisce il singolo elemento, applicabile SOLO a document
`document.`**`getElementsByName`** restituisce un vettore di elementi, applicabile SOLO a document

`wrapper.`**`getElementsByTagName`** restituisce un vettore di elementi
`wrapper.`**`getElementsByClassName`** restituisce un vettore di elementi-
Applicabile anche su più classi contemporaneamente `getElementsByClassName ("class1 class2")`

`wrapper.`**`querySelector`**`("input[type=text]")` restituisce il primo input type=text
`wrapper.`**`querySelectorAll`**`("input[type=text]")` restituisce un vettore di input type=text
Il primo se non trova nulla restituisce undefined, il secondo se non trova nulla restituisce un vettore vuoto.

Molto comodi perchè, a differenza dei precedenti, accettano come parametro un qualunque selettore / pseudoselettore CSS. **Però molto più pesanti. Da usare una volta sola all'onload**

Struttura dell'albero HTML
Nell'immagine sotto rappresentata si ipotizza un puntatore attualmente posizionato sul "nodo di riferimento" visualizzato in rosso



`current.`**`children`** restituisce un vettore con tutti i tag **figli diretti** di current
Accetta come parametro SOLO un indice numerico : **`myFirstDiv = myWrapper.children[0]`**
`current.`**`childNodes`** come sopra però contempla anche i text nodes ed i comment nodes
`current.`**`children[i]`** i-esimo **figlio diretto**
`current.`**`firstElementChild`** primo **figlio diretto**
`current.`**`lastElementChild`** ultimo **figlio diretto**
`current.`**`children.length`** numero di **figli diretti**
`current.firstChild` come sopra però contempla anche i text nodes ed i comment nodes
`current.lastChild` come sopra però contempla anche i text nodes ed i comment nodes
`current.`**`nextElementSibling`** fratello successivo
`current.`**`previousElementSibling`** fratello precedente
`current.`**`next/previousSibling`** come sopra però contempla anche i text nodes ed i comment nodes
`current.`**`hasChildNodes()`** indica se il nodo corrente ha nodi figli
`current.`**`parentElement`** genitore di current
`current.`**`closest("div")`** ricerca il primo predecessore di tipo DIV (antenato) di current

`current.contains(span)` verifica se il tag puntato da **span** è un descendant di current, cioè se *span* coincide con *current* oppure se è un suo figlio o nipote.

`current.textContent` contenuto testuale del nodo (foglia)
`current.innerHTML` intero contenuto HTML

`current.clone(true)` clona current. True fa si che vangano clonati anche tutti i descendant
Attenzione che **nel clone vengono 'persi' eventuali handler di evento**
`current.remove()` rimuove current dal DOM
`current.removeChild(current.firstElementChild)`

`document.createElement("div")` crea un nuovo HTMLElement . **Disponibile SOLO su document**
`current.appendChild(newChild)` appende child in coda a current. **Lo taglia dalla posizione corrente**
Non accetta come parametro una stringa HTML, nel qual caso occorre utilizzare .innerHTML
`current.append(node1, node2, )` consente di appendere più figli in coda a current
`current.prepend(node1, node2,)` consente di appendere più figli davanti al primo nodo ci current

`current.insertBefore(newChild, current.children[0])` appende newChild come 1° figlio di current
`current.insertAdjacentElement(pos, newElement)` aggiunge un nuovo fratello/figlio
Il primo parametro **pos** può assumere i seguenti valori:
`'beforebegin'`: Before the targetElement itself (fratello precedente)
`'afterbegin'`: Just inside the targetElement, before its first child (primo figlio)
`'beforeend'`: Just inside the targetElement, after its last child (ultimo figlio)
`'afterend'`: After the targetElement itself (fratello successivo)

`current.matches(".class1, :first-of-type")` verifica if current matches one of two selectors restituisce true / false

`current.tagName` legge il nome del tag (es P)
`current.nodeName` come sopra ma vale anche per gli attributi
`current.nodeType` tipo di nodo sulla base della seguente tabella:

```
1 = ELEMENT_NODE = nodo vero e proprio
2 = ATTRIBUTE_NODE = attributo
3 = TEXT_NODE = nodo testuale, cioè foglia dell'albero
5 = WHITESPACE_NODE =    a capo; etc.
8 = COMMENT_NODE = commento.
9 = DOCUMENT_NODE = l'intero documento
```

NodeList e htmlCollection

querySelectorAll restituisce una collezione di tipo **NodeList**
getElementsByName/TagName/ClassName restituiscono una **htmlCollection**
Sono entrambi oggetti iterabili simili ad un Array ma **NON** supportano i metodi di javascript funzionale e nemmeno lo splice(). L'unica differenza è che la **NodeList** supporta il forEach mentre **htmlCollection** no

Per convertire **NodeList** e **htmlCollecion** in **Array** :

```
let vet = Array.from(NodeList)      // oppure
let vet = [...NodeList]
```

Metodi per la gestione degli attributi

Attributi statici html

```
current.getAttribute(name);
current.setAttribute(name, value);
current.hasAttributes( );
current.hasAttribute(attributeName);
current.removetAttribute(name);
```

Attributi dinamici javascript : si utilizza semplicemente l'operatore **puntino**.

Attributi personalizzati : possono essere creati tramite dataset: `div.dataset.userName='pippo'`

Un dataset non è semplicemente una variabile javascript. In corrispondenza di ogni dataset viene creato un attributo html visibile nell'inspector ed utilizzabile all'interno di querySelector ed avente un prefisso **data-**. Il camelCase viene tradotto in trattino.

```
let div = document.querySelector("div[data-user-name='pippo']")
```

Selettori di attributo

I **selettori di attributo** permettono di individuare qualunque elemento del DOM sulla base dei suoi attributi HTML. *È importante notare che tutti i selettori CSS lavorano sempre sul DOM javascript, quindi vedono sia gli elementi presenti nel file HTML, sia quelli creati dinamicamente tramite JavaScript.*
Tuttavia i **selettori di attributo**, come [attr="val"], si basano **solo sugli attributi HTML cioè quelli definiti all'interno del file html oppure creati da javascript mediante .setAttribute().**
Non considerano invece le **proprietà DOM**, che possono avere valori diversi rispetto agli attributi.

```
input[type=text]
input[type="radio"]
input[name=txtNome]
input[name="D'Alessio"]
input[checked]
select[multiple]
```

Attenzione a **NON lasciare spazi** davanti alle quadre
Le **virgolette** sono facoltative. Devono **essere usate** se il value dell'attributo contiene spazi o apici singoli.
Ad esempio : `<input name="D'Alessio">`

Attributi booleani

Per agli attributi booleani (ad esempio **checked**) esistono due diverse sintassi:
[checked] che riflette la l'attributo statico checked
:checked che invece è una pseudoclasse che riflette la corrispondente proprietà del DOM.

Ovviamente le psudoclassi sono preferibili, però ne esistono ***solo*** 5 in tutto :

:checked	
:disabled	
:enabled	*(uguale opposto a disabled)*
:required	*(campi obbligatori)*
:read-only	
:selected	*(solo jQuery !! in javascript è mappato come :checked)*

Nel caso dell'attributo ***multiple*** di un listBox, non esiste la pseudo classe ma poiché l'attributo viene praticamente sempre settato a livello statico nel file html, si può tranquillamente usare ***[multiple]***

I selettori di attributo possono anche essere applicati in forma combinata. Esempi:

```
querySelector("input[type=radio][name=opt1]:checked")
querySelectorAll("input[type=radio][data-user-name=4Bstudent]")
alert(querySelector('input[name="optGenere"]:checked').value)
```

Notare il seguente esempio: `querySelectorAll(":disabled")` restituisce tutti gli elementi attualmente disabilitati. In javascript per ri-abilitarli tutti occorre eseguire un ciclo.
Viceversa in jQuery è possibile applicare un effetto ad una intera collezione, per cui sarebbe consentita una istruzione del tipo: `querySelectorAll(":disabled").disabled = false`

CSS Functions

```
:not(selettore_Secondario)   // Es input[type=radio]:not(:checked)
#menu li:has(ul)             // Gli elementi che contengono il descendant indicato
:contains(testo);            // solo jQuery. Restituisce quei tag che contengono il testo indicato
```

Pesudoclassi nth-child e nth-of-type

:first-child // Primo figlio **generico** (indipendentemente dal suo tipo)
:first-of-type // Primo figlio **del suo tipo**

:first-child restituisce true se l'elemento corrente **gode della proprietà di essere primo figlio (primogenito) del proprio genitore**, qualunque sia il genitore, e indipendentemente dal proprio tipo.

```
<div id="wrapper">
    <p> …… </p>
    <p> <input ……> </p>
    <input ……>
</div>
```

Il primo `<input>` presenta **:first-child** uguale a true perché è in assoluto il primo figlio di <p>
Il primo `<input>` presenta **:first-of-type** uguale a true perché, rispetto al proprio padre, è il primo input.
Il secondo `<input>` presenta **:first-child** uguale a false perché dentro wrapper, prima di lui, ci sono altri figli
Il secondo `<input>` presenta **:first-of-type** uguale a true perchè, all'interno di wrapper, è il primo input

:nth-child(i) // i-esimo figlio **generico** (indipendentemente dal tipo) (**a base 1**)
:nth-of-type(i) // i-esimo elemento **del suo tipo** (**a base 1**)

:last-child // Ultimo figlio **generico** (indipendentemente dal tipo)
:last-of-type // Ultimo elemento **del suo tipo**

#wrapper :nth-child(i) // i-esimo elemento **generico** contenuto in wrapper
#wrapper :nth-of-type(i) // i-esimo elemento **del suo tipo** contenuto in wrapper
:only-child // Figlio unico

Note

(1) Si consideri la seguente struttura:

```
<label>  <input type="radio" name="opt1" value="a">  </label>
<label>  <input type="radio" name="opt1" value="a">  </label>
```

```
input[type=radio][name=opt1]:nth-of-type(1)
```

non restituisce il primo radio button avente name=opt1, ma ***TUTTI*** i tag input che soddisfano alla proprietà di essere `:first-of-type` (ad esempio perché ognuno è il primo figlio di una label) e che hanno anche type=radio e name=opt1.

(2) Allo stesso modo

```
#wrapper p:first-of-type
```

non restituisce il primo tag `<p>` interno a #wrapper, ma tutti i tag `<p>` interni a #wrapper che soddisfano alla proprietà di essere `:first-of-type,` perché ad esempio ciascuno è inserito all'interno di una propria label.

(3) **:nth-child(i) e :nth-of-type(i)** non sono applicabili alle classi ma SOLO ai tag

even e odd

Gli pseudoselettori **:nth-child() e :nth-of-type()** oltre all'indice i-esimo accettano come parametro anche i valori: **even** tutti gli elementi pari - **odd** tutti gli elementi dispari.
Questi valori **sono utilizzabili soltanto all'interno di** **nth-child** e **nth-of-type.**
E' anche disponibile una firma del tipo **nth-of-type(3n - 2)**
prende tutti i multipli di 3 e sottrae 2, cioè: 1, 4, 7, 10, 13, etc

Puntualizzazioni su Form e Controlli : lettura e scrittura del value

in lettura

Nel caso di **select a selezione singola** restituisce automaticamente il **value della voce selezionata**.
Se il value della voce selezionata è vuoto restituisce il testo html della voce selezionata

Nel caso dei **radio button**, che sono esclusivi, tramite lo pseudo selettore ***:checked*** si può accedere all'unica voce selezionata e poi leggerne il value. Per i radio non selezionati si può usare ***:not(:checked)*** e poi eseguire un ciclo.

Nel caso dei **check box**, si può accedere alle voci selezionate tramite lo pseudo selettore ***:checked***.
Dopo di che occorre necessariamente eseguire un ciclo. Per i non selezionati ***:not(:checked)***

Nel caso di **select a selezione multipla** la soluzione migliore è quella di scorrere le voci selezionate tramite un ciclo su **selectedOptions** :

```
for (let option of lst.selectedOptions)
```

oppure
```
let selectedOptions = lst.querySelectorAll("option:checked")
```
oppure
```
for (const option of lst2.children)
  if(option.selected)
    msg += option.value + "\n";
```

In jQuery il metodo val() restituisce un vettore enumerativo contenente i value di tutte le voci selezionate. Questo NON vale invece per i checkbox dove il metodo val() restituisce normalmente il value della PRIMA voce selezionata

in scrittura

Nel caso del **ListBox a selezione singola** l'assegnazione del value provoca la selezione dell'elemento indicato. Se l'elemento non viene trovato il listbox verrà deselezionato (come selectedIndex=-1)

Nel caso di **radioButtons, checkBox** e **ListBox a selezione multipla** occorre necessariamente scorrere tutti gli elementi ed andare a settare opportunamente il campo **.checked** o **.selected**

```
for (const option of lst.querySelectorAll("option"){
   option.selected = false
   if(option.value=="a" || option.value=="c")
      option.selected = true
}
```

In jQuery è anche possibile passare al metodo **val()** un **vettore enumerativo di values** che, in tal caso, provoca la selezione di tutti gli elementi aventi il **value** indicato e la deselezione automatica delle altre voci `$(':checkbox').val(['chk-1','chk-2']);`
Anche nel caso dei **radio button** (che sono esclusivi) occorre comunque 'passare' il value della voce da selezionare all'interno di un vettore enumerativo lungo 1. Se si passano più value viene selezionato l'ultim

La gestione degli eventi

La selezione di un elemento da codice NON provoca MAI la generazione degli eventi, che dovranno eventualmente essere richiamati da codice

Esempio:
Dopo aver selezionato da codice una voce in un listbox occorre richiamare esplicitamente anche l'evento

```
list.addEventListener("change", esegui)
list.selectedIndex=2
esegui()
```

Evoluzioni di Java Script

1995 Netscape JavaScript
1996 **Standardizzazione ECMA** con il nome **ES** (**ECMA Script**), comunemente javascript
ECMA (European Computer Manufacturers Association) è un'associazione fondata nel 1961 e dedicata alla standardizzazione nel settore informatico e dei sistemi di comunicazione.
Dal 1994 viene chiamata ECMA International
1998 **ES2**

. . . . . .

2008 **Chrome V8 Engine** (scritto in C++)
2009 **ES5** (aggiunge fra le altre cose lo strict mode, le librerie JSON, i **web workers**, le promises)
2009 **Nodejs** per la programmazione lato server

2015 **ES6** o ES2015 – **programmazione funzionale**: iteratori, arrow functions, ES-Modules
2016 ES7 o ES2016 – assestamento dei cambiamenti introdotti nella precedente edizione.

Java Script Funzionale

In ES6 sono state introdotte molti *'metodi funzionali'* per svolgere operazioni su un gruppo di elementi. Queste funzioni sostituiscono i cicli, le ricerche, etc e forniscono prestazioni ritenute migliori.

Metodi di programmazione funzionale applicabili a vettori enumerativi e stringhe

Tutti i metodi seguenti applicati ad un vettore enumerativo richiamano **n** volte la funzione anonima passata come parametro iniettando di volta in volta uno dei record (o valori scalari) costituenti la collezione.
Si tratta di un approccio funzionale alternativo più veloce rispetto alla tipica scansione di un vettore enumerativo con un ciclo FOR OF.

Il metodo `forEach( )`

Applicato a un vettore esegue automaticamente la scansione dell'intero vettore. **A differenza di FOR OF che è applicabile a qualsiasi elemento iterabile, FOR EACH è applicabile SOLO agli elementi di tipo Array.** Riceve come parametro una funzione di callback richiamata in corrispondenza della scansione di ogni singolo elemento. A questa funzione vengono iniettati sia il valore dell'elemento sia il suo indice (facoltativo). **I parametri sono passati per valore, dunque se modifico item, i nuovo valori NON saranno visibili al di fuori del metodo medesimo.** Il metodo forEach è asincrono e non ritorna nulla.

```
vect.forEach(function(item, i) {
      console.log(item)
})
```

I metodi `find()` e `findIndex( )`

Sono molto simili. Entrambi eseguono una scansione del vettore individuando il primo record che soddisfa alla condizione impostata. Però:

- **findIndex** restituisce la **posizione** della prima occorrenza che soddisfa la condizione (-1 in caso di record non trovato)
- **find** restituisce **l'intero record** della prima occorrenza che soddisfa la condizione (**undefined** in caso di record non trovato)

Esempio: Sia **students** un vettore enumerativo di studenti aventi ciascuno una **matricola**

```
let pos = students.findIndex(function(student) {
      return student.matricola == 'AL636GZ'; });
students.splice(pos, 1)
```

findIndex è più completo rispetto al tradizionale **indexOf**.
indexOf consente di ricercare un intero record all'interno di un vettore di record (ovviamente se i puntatori coincidono), mentre **findIndex** consente di eseguire la ricerca sulla base di uno o più campi

Il metodo `filter( )`

Restituisce un vettore enumerativo contenente una **copia** di **tutti** i record che soddisfano al criterio di filtro:

```
let vet = students.filter(function (student) {
     return student.residenza == "Fossano"
})
```

Rappresenta una interessante alternativa allo splice per eliminare un record:

```
students = students.filter(function(x) {
      return student.matricola != 'AL636GZ'; })
```

Il metodo `map( )`

Il metodo **map** applica la funzione anonima ricevuta come parametro ad ogni elemento dell'array restituendo un **nuovo array** avente in linea di massima **la stessa lunghezza** dell'array iniziale, in cui ogni elemento è il risultato dell'elaborazione sul'elemento corrispondente nell'array iniziale. Note:

- L'array iniziale non viene modificato
- La funzione non viene eseguita su eventuali celle prive di valore

```
let materials = ["Hydrogen", "Helium", "Lithium", "Beryllium"];

let lengths = materials.map(function(item) {         // [8, 6, 7, 9]
    return item.length;
});
```

// Invece di una funzione anonima si può passare direttamente una funzione esistente che verrà richiamata per ogni item e a cui verrà iniettato l'item stesso :

```
let numbers = [4, 9, 16, 25];
let ris = numbers.map(Math.sqrt);      // [2, 3, 4, 5];
```

Il seguente esempio restituisce un vettore enumerativo contenente gli ID di tutti i checkbox della pagina.
Il metodo querySelectorAll restituisce una NodeList a cui non è applicabile il metodo .map()

```
let ids = Array.from(document.querySelectorAll("input[type=checkbox]"))
        .map(function(item) {
               return item.id;
         })
```

Anche le collezioni jQuery espongono un metodo `.map()` analogo al precedente, il quale però ritorna non un semplice vettore enumerativo ma una collezione jQuery

Spread Operator …

La sintassi **Spread** (indicata tramite tre puntini consecutivi) permette di suddividere un vettore enumerativo o un vettore associativo nei suoi elementi costitutivi.

Array enumerativi

```
let parts = ['mele', 'pere']
let fruits = ['arance', 'ciliegie' ...parts, , 'uva']
// risult  = ['arance', 'ciliegie' 'mele', 'pere', 'uva']
```

Attenzione però che se uno degli elementi è a sua volta un vettore annidato, questo viene restituito così com'è.

```
let parts  =  ['a', 'b', ['c', 'd']]
   ...parts  ->  'a', 'b', ['c', 'd']
```

Altro esempio: calcolo del massimo all'interno di un vettore di numeri:

```
const numbers = [1, 2, 3, 4, etc];
let max = Math.max(...numbers)
```

Array associativi (json)

Applicato ad un json, restituisce singolarmente tutte le coppie chiave/valore:

```
let obj1 = { foo:'bar', x:42 };
let obj2 = { foo:'baz', y:13 };
let  mergedObj = { ...obj1,  ...obj2 };
// risult =     { foo: "baz", x: 42, y: 13 }
```

Il secondo valore di `foo` sovrascrive il primo (non possono infatti esistere due chiavi uguali)

Lo spread operator consente anche di clonare un json esistente

```
let clonedObj = { ...obj1 };
```

Questa istruzione è simile a clonedObj = obj1 però, mentre clonedObj = obj1; copia semplicemente il puntatore, l'utilizzo dello spread operator **crea un nuovo oggetto**.
Attenzione però che nel caso di oggetti annidati viene copiato il puntatore, per cui se modifico una property annidata in un oggetto, la modifica si ripercuote anche sull'altro oggetto.

Lo Spread Operator è molto comodo anche per convertire una NodeList javascript (o una collezione jQuery) in un vettore enumerativo puro, come ottima alternativa al metodo Array.from() :

```
let vet = [...NodeList]
```

Il metodo `Array.from ( )`

Il metodo Array.from() converte in Array l'oggetto passato come parametro (ad esempio una NodeList)

Esiste però anche una seconda firma costituita da due parametri dove:

- Il primo parametro è un vettore enumerativo iterabile
- Il secondo parametro è una funzione di map che agisce sul primo vettore e restituisce un nuovo vettore

```
console.log(Array.from( [1, 2, 3],  function(x) { return x*2 }  ))
// Expected output: Array [2, 4, 6]
```

La funzione di callback in realtà può ricevere due parametri : il valore e l'indice (facoltativo)

```
Array.from( [1, 2, 3],  function(x, i) { return x*2 }  )
```

Il primo parametro è obbligatorio. Se non si intende farne uso, si assegna normalmente il nome underscore

```
Array.from( [1, 2, 3],  function (_, i) { console.log(i) }  )
```

Come primo parametro si può anche passare un json indicante la lunghezza del vettore.
Il seguente esempio crea una stringa casuale di 20 caratteri minuscoli:

```
let str = Array.from({length:20},
                     function(){return String.fromCharCode(random(26,98))})
        .join('')
```

// oppure in modo 'tradizionale' :

```
let str [...(new Array(20))].map( function(){
String.fromCharCode(random(26,98)) .join('')
```

new Array(20) crea un array di lunghezza 20, ma con tutti elementi "vuoti" (<empty>) **non iterabile.**
Le posizioni esistono ma non sono definite
Lo **spread operator ...** crea 30 elementi lineari tutti undefined
Le **quadre** finali ricostruiscono il vettore che sarà ora un vettore reale (quindi iterabile) con 20 celle reali tutte contenenti undefined.

Notare che in javascript richiamare Array() con o senza **new** NON cambia assolutamente nulla.
Il risultato è sempre lo stesso: un array di lunghezza 20 con slot vuoti (<empty>) non iterabile

L'oggetto Set

Gli oggetti SET sono delle '**collezioni' iterabili**, sostanzialmente vettori enumerativi, **in cui ogni valore può ripetersi una ed una sola volta.** Possono contenere qualsiasi item (scalari, stringhe, json, vettori).

Se si trasforma un vettore in oggetto Set, **tutte le voci replicate vengono rimosse**

```
let vet = [5, 6, 7, 6, 8, 6, 9, 7]
let set = new Set(vet)

set.add(9)            // non fa nulla perché il 9 è già presente
console.log(set)      // Set(5) { 5, 6, 7, 8, 9 }
```

A questo punto si può ritrasformare l'oggetto SET in normale vettore :

```
let vet2 = [... set]
```

Per avere la dimensione di un oggetto SET, non si usa la property *.length* ma la property ***.size***
E' possibile scorrere normalmente l'oggetto SET con un ciclo ***FOR-EACH***.
Non sono invece supportati gli altri metodi funzionali, nel qual caso occorre convertire il SET in Array

Il metodo `sort( )`

Dato un vettore di JSON, per ordinarlo si può utilizzare il metodo **.sort()** specificando come parametro il nome del campo su cui eseguire l'ordinamento, ed anche il verso crescente/decrescente.
La seguente procedura esegue un ordinamento del vettore sulla base del nome dei singoli studenti:

```
students.sort(function(item1, item2) {
        let str1 = item1.nome.toUpperCase();
        let str2 = item2.nome.toUpperCase();
        if (str1 < str2)
            return -1;
        else if (str1 > str2)
            return 1;
        else return 0;
    });
```

Per fare l'ordinamento inverso è sufficiente invertire il return all'interno delle due IF

Il metodo `replace( )`

Sostituisce la sottostringa contenuta nel primo parametro con quella contenuta nel secondo parametro.
E' anche possibile:

- Sostituire un **intero JSON** con un altro JSON (solo se il puntatore del primo JSON coincide con il puntatore all'interno del vettore)
- passare come primo parametro una espressione regolare che indica tutte le occorrenze da sostituire con il secondo parametro

Inoltre in **ES6** è possibile passare come secondo parametro una funzione di callback alla quale, ad ogni scansione, viene iniettato come parametro una della occorrenze individuate dall'espressione regolare impostata come primo parametro. Questa funzione può elaborare / modificare ciascuna occorrenza ritornando ad ogni scansione il nuovo valore.

```
str.replace(/\b\w/gi,    function (m) { return m.toUpperCase(); });
str.replace(/\b[a-z]/gi, function (m) { return m.toUpperCase(); });
```

equivalenti. Convertono in maiuscolo la lettera iniziale di ciascuna parola contenuta all'interno della stringa

La seguente riga invece evidenzia tutte le occorrenze della parola indicata.

```
let parolaDaCercare = "……"
let regEx = new RegExp(parolaDaCercare, "gi");
textBox.value = textBox.value.replace(regEx, (match) => {
  return `<mark>${match}</mark>`})
```

Per eliminare invece l'evidenziatore:

```
let regEx = new RegExp("<mark>|</mark>", "gi");
textBox.value = textBox.value.replace(regEx, (match) => {return ""});
```

Il metodo `.flat( )`

Trasforma un vettore di vettori in un vettore con tutti gli elementi al primo livello. Può agire anche su più livelli di annidamento.

```
const vet = [0, 1, 2, [3, 4]];
console.log(vet.flat());      // [0, 1, 2, 3, 4]
```

Comodo ad esempio quando vari oggetti presentano un campo vettoriale comune (es hobbies) e si vuole ottenere un vettore con tutti i valori di quel campo (vettore che potrà poi essere reso univoco tramite Set)

```
let hobbies = []
for(let student of students)
  hobbies.push(student.hobbies)   // push di un intero vettore
hobbies = hobbies.flat()
let set = new Set(hobbies)
```

Il metodo `.flatMap( )`

Identico a .map() ma in più esegue un flat() finale (nel caso in cui la map() sostituisca una cella scalare con un piccolo vettore enumerativo).

Il metodo `some( )`

Restituisce **`true`** se **almeno un elemento dell'array** soddisfa ai criteri della condizione impostata.
Evita un ciclo for sull'array. Non modifica l'array.

```
const array = [1, 2, 3, 4, 5];
console.log(array.some(function (item) {
      return item%2 == 0 })
```

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**". (questo anche nel caso di un vettore).

Per verificare di quale object si tratta, si può utilizzare l'operatore **instanceof** che, a differenza di typeof(), non è una funzione ma un semplice operatore come ad esempio ==

```
if (myVar instanceof FormData) ...
if (myVar instanceof Array) ...
```

Oppure si può utilizzare la seguente istruzione che restituisce l'object type come semplice stringa:

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

Per vedere se è un array si può anche utilizzare `if(Array.isArray(myVar))`

Il metodo animate()

```
const animation = element.animate(
  { transform: "translateX(600px)"},
  { duration:3000, fill:"forwards"})
await animation.finished;     // restituisce una promise risolta al termine dell'animazione
animation.commitStyles();
//animation.cancel();
```

- Il **primo parametro** indica la proprietà CSS da animare.
  E' possibile animare qualsiasi proprietà css numerica. Anche più di una contemporaneamente :

```
id1.style.position = "relative"; // o "absolute"
id1.style.left = "0px";  // valore iniziale obbligatorio
const animation = id1.animate(
  { left:"600px", "marginTop":"80px"},
  { duration:3000, fill:"forwards"})
await animation.finished;
animation.commitStyles();
//animation.cancel();
```

  E' anche possibile animare più property in sequenza, passando come prima parametro un vettore enumerativo di json, ciascuno con un proprio offset temporale. Se l'offset non viene specificato, viene equamente suddiviso fra i vari valori.

```
element.animate( [{opacity:1}, {opacity:0.1, offset: 0.7}, {opacity:0}],2000)
```

- Il **secondo parametro** può essere semplicemente un tempo in sec oppure si può specificare un secondo json come indicato sopra. L'opzione `fill:"forwards"` serve a rendere persistente l'animazione, nel senso che l'oggetto rimane nella sua posizione finale, altrimenti ritornerebbe alla posizione iniziale. Insieme a `fill:"forwards"` è consigliato l'utilizzo di **`animation.commitStyles()`** che va a settare le nuove property CSS.

`animation.cancel()` consente di riportare l'oggetto nella sua posizione iniziale antecedente l'animazion

fadeIn

```
div.style.transition='opacity 1s';
function esegui() { div.style.opacity=0;
                    setTimeout(() => div.style.opacity=1, 1000)   }
```

Delegated Events

Considerazioni sulla associazione degli eventi

A differenza delle proprietà CSS definite all'interno di un foglio di stile che vengono applicate anche a posteriori sui nuovi elementi creati dinamicamente, **l'associazione degli eventi ad un elemento del DOM deve essere fatta soltanto DOPO che l'oggetto è stato creato ed appeso al DOM**.
Se l'elemento viene rimosso dal DOM e riappeso, l'associazione con l'evento va persa !!

La seguente riga, che si suppone posizionata al form_load, agisce SOLO per gli elementi esistenti

```
for (let btn of buttons)
   btn.addEventListener("click", function(){
```

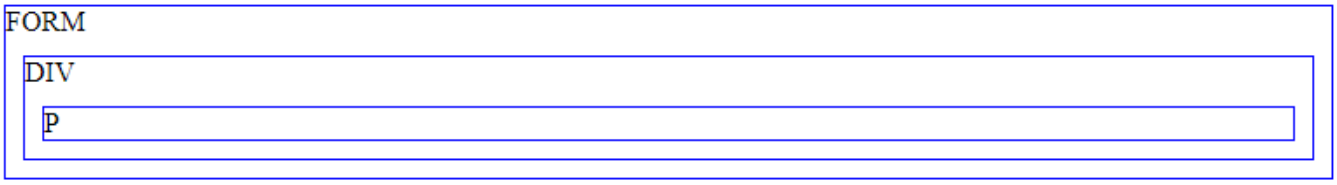
Se nuovi button verranno creati successivamente l'associazione non verrà eseguita.

Per gli elementi creati dinamicamente di solito si associa l'evento durante la creazione stessa, dopo averlo appeso al DOM. Se però il medesimo addEventListener viene eseguito più volte, nel caso delle procedure anonime l'associazione ad evento viene eseguita più volte, per cui poi l'evento si verificherà più volte.

I **delegated Events** rappresentano una interessante alternativa per associare gli eventi ad un elemento esistente per poi **delegarli agli elementi interni** (esistenti o che verranno creati dinamicamente in un secondo tempo)

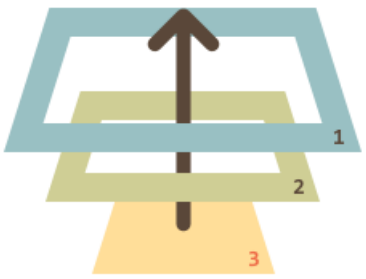
Directly bound events and Event Bubbling

Si consideri la seguente struttura html




```
<form onclick="alert(this.name)">
   FORM
   <div>
      DIV
      <p> P </p>
   </div>
</form>
```

Se si clicca sulla form **o su un qualsiasi elemento interno** la alert() verrà sempre eseguita.
Infatti se si verifica un evento su un elemento interno, **questo evento viene sempre propagato verso l'alto** (**Event Bubbling**) a tutti i suoi genitori, fino ad arrivare al tag ***form*** e poi al tag ***body***.



1° Vantaggio: definendo l'evento sul tag FORM anziché sugli elementi interni, **l'evento verrà eseguito anche nel caso di elementi interni aggiunti dinamicamente in un secondo tempo.** Notare però che nell'esempio ***this*** rappresenta SEMPRE la form, indipendentemente da dove è stato fatto il click.

Nota Sulla base di quanto detto, se si modifica il codice precedente con l'aggiunta di due eventi interni

```
<form onclick="alert('form')">
   FORM
   <div onclick="alert('div')">
       DIV
       <p onclick="alert('p')">P</p>
   </div>
</form>
```

Se si clicca sulla form compare il messaggio **form**
Se si clicca su tag DIV compaiono i messaggi **div form**
Se si clicca su tag P compaiono i messaggi **p div form (cioè in *bubbling phase*)**

Delgated Events

Si supponga di nuovo di gestire l'evento soltanto sulla form esterna
Nell'esempio precedente, indipendentemente da dove si clicca, **this** farà sempre riferimento all'elemento a cui l'evento è direttamente collegato, cioè al tag ***form***.

Però, quando si clicca su un elemento interno, insieme all'evento, **viene SEMPRE propagato anche il puntatore all'elemento che ha scatenato l'evento**, puntatore che sarà **disponibile all'interno di *event.target*** Per cui sostanzialmente diventa possibile **delegare** la gestione dell'evento all'elemento interno su cui l'evento si è verificato (**capturing phase**). All'interno del gestore è sufficiente controllare event.target e gestire l'evento SOLO SE è avvenuto all'interno di un elemento che ci interessa

```
form1.addEventListener("click", function(event){
      if(event.target.tagName == "p")
         alert(event.target.textContent)
})
```

event.target puntatore all'elemento che ha scatenato l'evento.
event.currentTarget puntatore all'elemento primario (esterno) a cui è collegato l'evento (cioè ***this***)

Vantaggi:

- **Come detto, al momento della definizione dell'evento è sufficiente che esista il contenitore** esterno. **Gli elementi interni possono essere aggiunti dinamicamente in un secondo tempo**.
- Un altro notevole vantaggio è legato alle prestazioni. Supponendo di avere una tabella con 1000 righe, il seguente codice definisce 1000 associazioni di evento, una per ogni **tr**

```
let rows = document.querySelectorAll("tr")
for (let row of rows)
    row.addEventListener( "click", function() {
        console.log(this.textContent)  })
```

Il seguente codice definisce invece **una sola associazione di evento** su **table** demandando poi la gestione dell'evento agli elementi interni:

```
table.addEventListener( "click", function(event) {
    if(event.target.tagName == "tr")
        console.log(event.target.textContent)
});
```

Arrow Functions e Binding del this

Le Arrow Function, basate sull'operatore **lamda,** sono nate in ES6 **per**

- **semplificare la scrittura delle funzioni di callback all'interno di una classe**,
- **migliorare la gestione del `this`.**

Sintassi Generale delle Arrow Function

La sintassi delle arrow function **omette** la parola chiave **function,** sostituita dall'operatore **lamda** ed i parametri vengono anteposti all'operatore lamda medesimo. La sintassi generale prevede infatti:

- le **parentesi tonde** intorno alla lista dei parametri da passare alla funzione di callback
- l'operatore **lamda**
- le **parentesi graffe** per delimitare il corpo della funzione

Si consideri ad esempio una funzione anonima che implementa la somma di due numeri:

```
var somma = function(x, y) {
      return x + y; };
```

con la sintassi delle arrow function diventa:

```
var somma = (x, y) => {return (x + y)}
```

L' **invocazione** di una arrow function è del tutto identica a quella di una normale funzione :

```
var totale = somma(3, 2);
```

Casi Particolari

- Se il corpo è costituito da una singola istruzione, si possono omettere ***sia* le parentesi graffe** per individuare il corpo della funzione ***sia* la scrittura diretta del return**. Ciò che viene ritornato dall'espressione interna (o dall'eventuale sottofunzione) viene automaticamente ritornato al chiamante. Se però si mettono le parentesi graffe, allora ci vuole anche il return

  ```
  (x, y) => x + y;
  (x, y) => {return x + y;}
  ```

- Se la funzione prevede un solo parametro, si possono omettere le parentesi tonde. esempio:
  ```
  x => x * 2
  ```

- Se la funzione non prevede alcun parametro, l'uso delle parentesi tonde è obbligatorio :
  ```
  () => "Hello world!"
  ```

Utilizzo all'interno delle funzioni di callback

Grazie alla'estrema compattezza, le arrow function si prestano bene ad essere utilizzate nelle callback

Consideriamo ad esempio il seguente codice tradizionale:

```
let numeri = [18, 13, 24];
numeri.forEach(function(valore) {
   console.log(valore);
});
```

Possiamo semplificare la sua sintassi riscrivendolo nel seguente modo:

```
numeri.forEach(valore => console.log(valore));
```

Il Binding del this

La peculiarità delle arrow function è che **non ridefiniscono il `this` che mantiene pertanto il significato che aveva all'interno dello scope genitore a monte della arrow function medesima**.

Visibilità del this in assenza delle arrow function

- All'interno di una generica funzione **`this`** rappresenta sempre un generico "Spazio delle funzioni" all'interno del global object window, per cui se si tenta di accedere ad un qualunque campo (es **`this.name`**) il risultato sarà sempre **`undefined`**. Lavorando in "**`strict mode`**" (use strict), **`this`** sarà lui stesso **`undefined`**.
- All'interno di un **metodo di una classe**, **`this`** consente di accedere a Proprietà e Metodi della classe (*usato all'interno del costruttore consente di creare nuove proprietà*)
- All'interno di un **metodo di evento** linkato ad un oggetto attraverso un bind (**`.addEventListener`** in js oppure **`.on`** in jQuery), **il `this` viene ridefinito** e rappresenta **l'oggetto al quale è associato l'evento**.

Che cosa cambia con le arrow function

Nel terzo caso precedente (**cioè quando mi trovo all'interno di un metodo di evento**) se si usano le arrow function il this NON viene ridefinito e continua a puntare allo scope che aveva prima della definizione dell'evento. Molto comodo se mi trovo all'interno di una classe e voglio poter continuare ad accedere ai membri della classe.

Esempio tradizionale senza arrow function : il this viene ridefinito

```
let _div =document.querySelector("div")
class User {
    nome="pippo"
    constructor() {    }
    visualizza() {
        _div.animate({"width":"200px", "height":"200px"}, 1000,
          function(){
            // this viene ridefinito e rappresenta _div
            this.style.backgroundColor = "red"    // OK perché this === _div
            this.nome="pluto"                     // NOK
        })
    }
}

let user = new User()
user.visualizza()

btnLeggi.addEventListener("click", function(){
    console.log(user.nome)  // pippo
})
```

Stesso Esempio con arrow function: il this NON viene ridefinito

Le arrow functions, come detto, non ridefiniscono il `this` che mantiene pertanto il significato che aveva all'interno dello scope genitore a monte della arrow function medesima.

Proviamo allora a riscrivere il codice precedente facendo uso delle arrow function
Il risultato si ribalta completamente:

```
let _div =document.querySelector("div")
class User {
      nome="pippo";
      constructor() { }
      visualizza() {
            _div.animate({"width":"200px", "height":"200px"}, 1000,
              ()=>{
                  // this NON viene ridefinito e punta alla classe
                  this.style.backgroundColor = "red"     // NOK
                  this.nome="pluto"                      // OK  this === class
            })
      }
}

let user = new User()
user.visualizza()
btnLeggi.addEventListener("click", function(){
      console.log(user.nome)  // pluto
})
```

Conclusioni

In conclusione le arrow function da un lato semplificano la programmazione orientata agli oggetti e **vanno benissimo se utilizzate all'interno di una classe**, ma dal lato opposto creano complicazioni indesiderate in altri casi, come ad esempio :

- nella callback di un evento
- all'interno di un json (pur essendo il json l'analogo di una classe istanziata)

In questi casi le arrow function non ridefiniscono il **`this`** che punterà all'oggetto esterno rispetto all'oggetto in cui si sta utilizzando il this

Esempio di utilizzo all'interno di un json

```
var obj = {
    n: 13,
    b: function() {
       console.log(this.n, this)    // this ridefinito punta a oggetto corrente
    }
    c: () => {
        console.log(this.n, this)  // this NON viene ridefinito punta a window
    }
}
obj.b();      // 13,  Object
obj.c();      // undefined, Window
```

Le classi in java script

Concetto di Function

Le **funzioni** sono oggetti (di tipo Function) **invocabili,** cioè a cui è possibile passare dei valori che non vengono 'salvati' ma elaborati, e ritornano sempre un valore, eventualmente **undefined**.
Come qualsiasi altro Object, le Functions possono essere assegnate ad una variabile e possono essere passate come parametro ad un'altra funzione.

Per definire una funzione esistono due sintassi equivalenti:

```
let myFunction = function (name, surname){ }   // function expression
    //   oppure
function myFunction (name, surname){ }         // function declaration
```

Concetto di Classe

- JS per sua natura è un linguaggio non Object Oriented, il che lo rende molto più leggero.
- Nei linguaggi ad oggetti si distingue tra la definizione di classe (il template) e l'istanza di classe, in cui il contenuto dei dati viene definito in fase di istanza.
- Viceversa In java script **le classi sono semplicemente "funzioni speciali" che consentono di "creare" un JSON** mediante una sintassi alternativa rispetto alla scrittura diretta del json stesso.

  I **vantaggi** relativi all'utilizzo delle classi rispetto alla scrittura diretta di un JSON sono:
  - La possibilità di gestire un controllo più fine sull'assegnazione dei valori
  - La possibilità di creare **più** JSON diversi a partire da una stessa funzione "costruttrice"

Dichiarazione delle classi in ES5

Le classi in javascript vengono definite tramite una normalissima generica funzione che dovrà però essere invocata NON direttamente ma tramite l'operatore **new**

```
function User(name, surname){                  // function declaration
   this.name = name;                           // creo una Property name
   this.surname = surname;                     // creo una Property surname
   this.setName=function(s){this.name=s};      // creo un method
}

let user = new User("Henry", "Ford");
```

In pratica **il new crea un nuovo JSON vuoto e fa sì che, all'interno della funzione, il `this` punti a questo json.**

Tutti i campi creati con l'operatore `this` definiscono una chiave all'interno del json ed assumono il significato di Proprietà e Metodi della classe

In pratica la funzione User() assume il significato di **costruttore** dell'Oggetto User

Note:

- Se si omette il **`this`** davanti al nome del campo, il campo diventa sostanzialmente una variabile PRIVATA all'interno della funzione e non viene creata nessuna property
- **`User`**, essendo una normale **Function,** di per se può anche essere richiamata normalmente **senza il new**. Però, nel momento in cui viene richiamata senza il new, **`this`** risulterà undefined, per cui tutte le istruzioni che iniziano con il this andranno in syntax error

L'istruzione Prototype

A differenza dei JSON scritti direttamente, gli Oggetti istanziati tramite l'utilizzo di una classe sono **estensibili, cioè diventa possibile in qualunque momento estendere la classe (mediante la parola chiave `prototype`) con l'aggiunta di nuovi metodi che saranno automaticamente accessibili da tutte le istanze**

```
User.prototype.sayHi = function() {
    alert(this.name);
}
user.sayHi();
```

Il prototype deve essere applicato al **nome della classe** ed è poi utilizzabile in tutte le istanze.

Note

1. I metodi per accedere ai membri della classe (proprietà e metodi) devono sempre utilizzare il **`this`**
2. I metodi, oltre che accedere alle property e agli altri metodi della classe, possono anche accedere a semplici funzioni definite nello stesso file al di fuori della classe. In questo caso occorre omettere il this. Conviene inserire all'interno della classe tutti i metodi che devono essere visibili all'esterno (**metodi pubblici**) e definire come semplici funzioni tutti quei metodi che devono essere utilizzati solo all'interno della classe (**metodi privati**).
3. A livello operativo definire un metodo "all'interno della classe" oppure "all'esterno tramite prototype", NON cambia assolutamente nulla. Però:
   - I metodi definiti tramite il **this** all'interno della classe (es. `setName()`) vengono replicati all'interno di ogni singola istanza.
   - I metodi definiti tramite **prototype** **non** vengono replicati all'interno di ogni singola istanza**.** Vengono invece memorizzati all'interno di un'area comune; a tutti gli effetti faranno parte delle istanze, (e dunque potranno accedere ai membri di istanza tramite il this), ma il loro codice non viene replicato in tutte le istanze. Un po' come avviene normalmente per i metodi nella programmazione Obiect Oriented

   Un altro aspetto importante nell'utilizzo del prototype è che tutte le modifiche apportate al prototype vengono propagate anche agli oggetti già allocati.

Dichiarazione delle classi in ES6

ES6 introduce una nuova sintassi per la gestione delle classi tramite la parola chiave **`class`** che rappresenta una semplice **sintassi alternativa** a quella già esistente in ES5 e NON un nuovo modello di ereditarietà orientata agli oggetti, come nei linguaggi Object Oriented.
La parola chiave **`class`** è semplicemente un 'abbellimento estetico' della sintassi precedente.

Come in ES5 **le classi sono semplicemente "funzioni speciali" che, al momento del new, consentono di "creare" un JSON** mediante una sintassi alternativa rispetto alla scrittura diretta. Rispetto alla scrittura diretta del JSON, le classi consentono un maggior controllo sui tipi e sugli accessi R / W

In ES6 la classe precedente può essere riscritta nel modo seguente:

```
class User {
  name=""      // property
  constructor(name) {
     this.name = name;
  }
  sayHi() {
     alert(this.name);
  }
}

let user = new User("John");
console.log(user.name)
user.sayHi()
```

- All'interno della classi si possono definire sia **proprietà** sia **metodi**.
  Entrambi possono facoltativamente essere terminati da punto e virgola
- Non sono ammessi separatore fra i campi di una classe, nemmeno la virgola.
- **Davanti alle Property occorre omettere sia `var` sia `let`.**
- **Davanti ai Method occorre omettere `function`**

- Il codice scritto all'interno di una classe viene automaticamente gestito in **`strict mode`**, cioè è obbligatoria la dichiarazione delle variabili.
- In javascript davanti a proprietà e metodi non sono ammessi i **qualificatori di visibilità** public e private che per default sono tutti **public**. Se si vuole rendere private una property o un metodo occorre utilizzare il # davanti al nome del campo: **`#sayHi()`**
- La dichiarazione esplicita delle **Property** non è obbligatoria. Se nell'esempio precedente si omettesse la Property **`name`**, essa verrebbe automaticamente creata dal costruttore in corrispondenza dell'istruzione `this.name = name`
- **I metodi** (essendo dichiarati internamente mediante prototype) **non** vengono replicati all'interno di ogni singolo JSON ma vengono invece memorizzati all'interno di un'area comune della funzione in modo che tutte le istanze possano accedervi.

Traduzione della classe precedente in codice ES5

In pratica la parola chiave **class** fa due cose:

- **Definisce una funzione avente il nome della classe e che rappresenta un riferimento alla funzione definita tramite la parola chiave constructor**
- Crea una funzione .prototype per ogni metodo elencato all'interno della classe.

```
let User = function (name){
   this.name = name;
}

User.prototype.sayHi = function() {
   alert(this.name);
}
```

```
// main
let user = new User("John");
console.log(user.name)
user.sayHi();
```

Note:

- A differenza di ES5, una funzione definita tramite **class** non può essere richiamata senza il new. Il **new**, come detto, è quello che fisicamente crea e restituisce il JSON con all'interno tutte le property definite all'interno della classe.
- Se all'interno della classe non si specifica il metodo costruttore, viene automaticamente inserito un costruttore vuoto, come se avessimo scritto `constructor() {}`
- Anche nella sintassi ES6 continua ad essere consentito l'utilizzo del **prototype**

```
class User { }
var user = new User()
User.prototype.test = 5;
alert(user.test);    // 5
```

Metodi Statici

I metodi statici possono essere definiti direttamente all'interno della classe tramite la parola chiave **static**

```
class User {
  static staticMethod() { alert("Static Method");  }
}
```

Oppure anche "fuori" dalla classe nel modo seguente (senza il prototype):

```
User.staticMethod = function () { alert("Static Method") };
```

Il richiamo deve avvenire mediante il nome della classe:

```
User.staticMethod();
```

Definizione delle Property di una classe tramite Getters / Setters

Per definire le property della classe è anche possibile utilizzare i metodi **getter** e **setter** che servono per controllare gli accessi alle property. Notare come sia **get** che **set** trasformano in Property quello che in apparenza è un metodo (**name**)

```
class User {
   constructor(name) {
      this._name = name;    // se esiste, invoca il setter
   }
   get name() {
      return this._name;
   }
}

   set name(value) {
     if (value.length < 4)
        alert("Name is too short.");
     else
       this._name = value;
   }
```

```
let user = new User("John");
alert(user.name); // John
user.name = 'Jack'
user = new User(""); // Name is too short.
```

Import Export di un modulo

Storicamente esistevano in javascript due diversi standard per l'import di una libreria:

- Lo standard **CommonJS** usato lato **server** e caratterizzato dall'utilizzo dei metodo **`require()`**
- Lo standard **UMD** (Universal Module Definition) usato lato **client** con link del modulo direttamente dal file html (es bootstrap, jquery, axios)

Con ES6 (2015) è stato introdotto un nuovo standard detto **ES-Modules** che ha poco alla volta rimpiazzato i due standard precedente. Due le principali innovazioni:

- Mentre i due precedecessori erano entrambi strutturati in modo **sincrono** (blocco dell'applicazione fino al termine del caricamento del modulo), **ES-Modules** carica i moduli dinamicamente in modo **asincrono** con una callback che rende disponibile il modulo all'applicazione nel momento in cui serve. Molto più prestante.
- Con i due precedecessori occorreva importare sempre l'intera libreria, mentre **ES-Modules** consente di importare la libreria intera oppure soltanto i metodi che servono in modo da non appesantire troppo l'applicazione.
- Inoltre il metodo **`import`** deve necessariamente essere richiamato **all'inizio del file** prima di qualsiasi altra istruzione.

Import di un modulo in vanilla javascript

Un file index.js che intende richiamare un modulo esterno, deve essere dichiarato nell'html con l'opzione **`type='module'`**. Questo fa si che il browser, dopo aver letto il file html, richieda al server anche o moduli contenuti all'interno di index.js

```
<script type="module" src="./index.js"></script>
```

Notare che :

- Mentre una libreria **UMD** è a visibilità globale, cioè può essere utilizzata in qualsiasi file .js appartenente al progetto, una libreria **ESM** NON può essere importata nel file html e può essere utilizzata SOLO nel file.js che esegue l'import. Se più file.js necessitano di dover utilizzare quella libreria, dovranno ognuno importarla per conto proprio
- Ogni file.js che esegue una ***import*** dovrà essere dichiarato nell'html come **`type='module'`**.
- I moduli **ESM** non condividono lo scope, per cui due file dichiarati **`type='module'`** non si vedono fra loro
- i moduli ESM, per motivi di sicurezza, funzionano **solo** se la pagina viene servita da un **server HTTP** (non tramite file://). Quindi, se si apre il file HTML direttamente con doppio click, si ottiene un errore del tipo: "*Access to script at ... from origin 'null' has been blocked by CORS policy*" .

Due semplicissimi http Server sono :

- **npx** che una utility di nodejs (da installare a livello globale) da eseguire a linea di comando tramite ***`npx serv .`***
- Installare in Visual Studio Code l'utility **Live Server** di Ritwick Dey che può essere eseguito in modo contestuale (con tasto destro in qualsiasi punto del file html) oppure tramite apposto comando in basso a destra sulla barra di stato (**Go Live**, porta 5500)

Scrittura di un modulo: l'istruzione export

Per esportare una variabile, una funzione o una classe, è sufficiente aggiungere l'istruzione export davanti al nome dell'oggetto da esportare:

```
export let months = ['Jan','Feb','Mar','Apr','Aug','Sep','Oct','Nov','Dec'];
export const PI = 3.14;
export function sayHi(name) { return "Hi " + name }
export class User { }
```

oppure l'export può essere eseguito totalmente o parzialmente a fine pagina

```
export {PI, sayHi}     // {"PI":PI, "sayHi":sayHi }
```

che equivale a scrivere esplicitamente ciò che in realtà viene fatto in automatico dal compilatore.

Cioè**, indipendentemente dalla sintassi utilizzata**, **ciò che che viene 'esportato' è sempre un json** avente come **chiav**i il nome di tutte le variabili / funzioni contrassegnate come export.

Import di un modulo

```
import {PI, sayHi, months, User }  from "./modulo.js"
```

cioè viene creata una varibile **PI** avente come valore il contenuto della chiave **PI** all'interno del modulo
Idem per le altre chiavi,

Dal momento che viene esportata la classe User e non l'istanza, il main dovrà provvedere lui all'istanza:

```
let user = new User()
```

export default

Un modulo può esporre un unico **export default** che non può essere associato ad una costante o ad una variabile, ma soltanto ad una funzione oppure, meglio, ad una classe.

```
export default class User { }
```

Il main potrà importare questa ***default class*** assegnandogli un nome arbitrario:

```
import myUserClass, {PI, months, sayHi}  from "./modulo"
```

Nota

Molto spesso le librerie espongono le loro principali funzionalità

- **sia** come metodi della ***default class***
- **sia** come funzioni indipendenti

Ad esempio:

```
    import fs from 'fs';
    fs.readfile("myFile")

// oppure
    import { readFile } from 'fs';
    readfile("myFile")
```

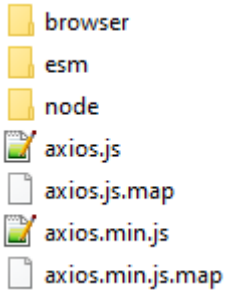
In quest'ultimo caso i metodi importati possono essere utilizzati in modo diretto senza dover anteporre il nome della libreria.

Distribuzione delle librerie

- Se si intende utilizzare la libreria in modo 'tradizionale' con link dall'html occorre scaricare la versione **UMD** (o utilizzare un relativo CDN Content Delivery Network, Rete di Distribuzione dei Contenuti)
- Se si intende utilizzare la libreria come 'modulo' con link dal file javascript occorre scaricare la versione **ESM** (o utilizzare un relativo CDN). Per in questa modalità occorre un server http.

I due principali siti di distribuzione delle librerie javascript sono **cdnjs/cloudflare** e **jsDelivr** che rendono disponibili sia il download sia il CDN.

Il download viene di solito distribuito come **package npm**, che è il formato utilizzato da nodejs.
In figura è riportata l'alberatura della libreria ***axios@1.13.2*** che è una libreria per l'invio delle richieste ajax. Nella sottocartella **`/dist`** si trova la libreria UMD disponibile nelle due versioni completa e minimizzata, mentre nell'ulteriore sottocartella **`/esm`** di trova la versione ESM disponibile sempre nelle due versioni completa e minimizzata



Al di là delle minimizzazioni (a capo, indentazioni e commenti) le librerie sono spesso scritte usando un codice ottimizzato ma poco leggibile. Il file **.map** abbinato alla libreria è una **source map**, cioè una "mappa" che collega il codice trasformato al codice sorgente originale, in modo da consentire un più agevole debugging del codice stesso. Ad esempio

```
function(a,b,c){return a.b(c)?...}
```
diventa
```
function getData(id) {
    return fetch(`/api/${id}`);
}
```

Il file .map che viene collegato alla libreria in modo automatico senza necessità di ulteriori link. In coda alla libreria c'è una riga del tipo `//# sourceMappingURL=axios.min.js.map` che dice al browser di scaricare anche il file .map

jsDelivr

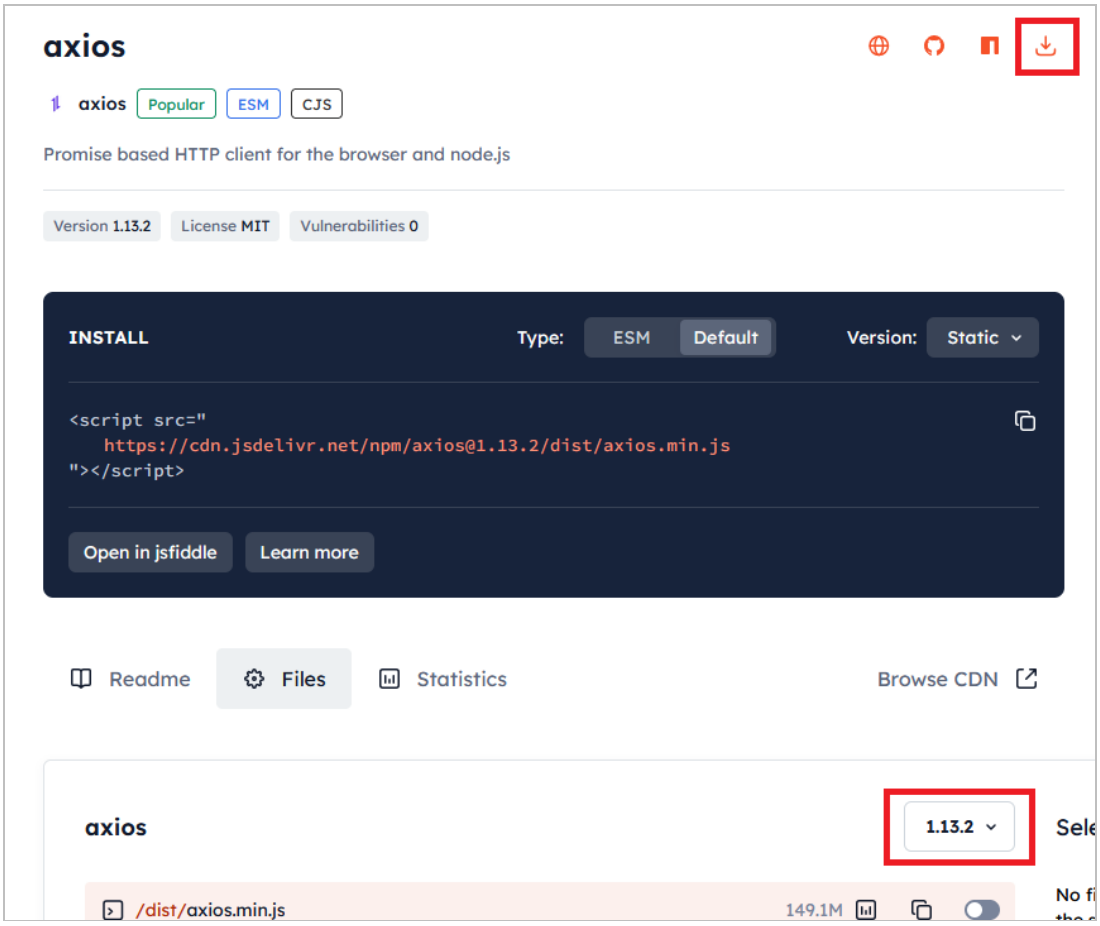
Una volta individuata la libreria, è possibile:

- Scegliere la versione (in basso)
- Scaricare l'intero npm package (in alto)
- Copiare il link UMD (indicato come default)

```
<script src="https://cdn.jsdelivr.net/npm/axios@1.13.2/dist/axios.min.js">
</script>
```

- Copiare il link ESM

```
import axios from 'https://cdn.jsdelivr.net/npm/axios@1.13.2/+esm'
```



Il **+esm** finale è corretto. Viene interpretato direttamente da jsdelivr