La rappresentazione dei dati nel web: XML e JSON

Rev 3.4 del 30/10/2021

La rappresentazione dei dati nel web

La codifica della pagina: Charset

I file html vengono normalmente salvati in formato **UTF-8 senza BOM** (intestazione).
UTF-8 è un formato che salva :

- i caratteri ascii base (dallo 0 al 127) su un byte
- i rimanenti caratteri su 2 bytes in formato Unicode

L'intestazione (BOM) serve ad avvisare il lettore riguardo al formato del file, però può interferire con la gestione server per cui normalmente viene omessa ed il formato utilizzato viene scritto in testa al file tramite un apposito meta tag html:

```
<meta charset="UTF-8">
```

Il tipo di contenuto: Internet Media Types

Il **media-type** indica il tipo di informazioni contenute all'interno del file :

```
<meta http-equiv="content-type" content="text/html”>
```

IANA manages the official registry of **media types** The identifiers were originally defined in **RFC 2046**, and were called **MIME types** because they referred to the non-ASCII parts of email messages that were composed using the MIME specification (**Multipurpose Internet Mail Extensions**).
They are also sometimes referred to as **Content-types**.
Their use has expanded from **email** sent through SMTP, to other protocols such as HTTP, and others.
New media types can be created with the procedures outlined in **RFC 6838**.

Text Type, for human-readable text and source code.

text/plain: Textual data; Defined in RFC 2046 and RFC 3676
text/html: HTML; Defined in RFC 2854
text/css: Cascading Style Sheets; Defined in RFC 2318
text/xml: Extensible Markup Language; Defined in RFC 3023
text/csv: Comma-separated values; Defined in RFC 4180
text/rtf: RTF; Defined by Paul Lindner
Elenco completo : http://en.wikipedia.org/wiki/Internet_media_type

text/javascript JavaScript; Defined in and made obsolete in RFC 4329 in order to discourage its usage in favor of **application/javascript.** However, text/javascript is allowed in HTML 4 and 5 and, unlike application/javascript, has cross-browser support.
application/json dati JSON serializzati

The "type" attribute of the <script> tag in HTML5 is optional and there is no need to use it at all since all browsers have always assumed the correct **default,** even before HTML5.

Le stringhe su righe multiple

Le stringhe in JS, sia quelle racchiuse tra apici singoli sia quelle racchiuse tra apici doppi, possono essere scritte su righe multiple terminandole con il carattere backslash **** che però deve necessariamente essere l'ultimo carattere della riga, **senza eventuali spazi successivi**.

```
var s = “salve \
        mondo”;
```

Gli oggetti localStorage e sessionStorage

Per ragioni di sicurezza i browser impediscono a javascript l'accesso al file system della macchina locale. Cioè non è consentita una istruzione del tipo

```
var content = readFile("c:\cartella1\file2.txt");
alert (content);
```

Poiché talvolta una applicazione ha necessità di salvare delle informazioni sulla macchina locale, è stato aggiunto alle librerie java script un apposito oggetto denominato **localStorage** che consente ad ogni applicazione di salvare fino a 10 MB di dati sul **HD** del PC locale, all'interno di un'area gestita dal browser e non direttamente visibile all'utente.

Le variabili salvate all'interno del LocalStorage sono suddivise per **dominio.**
Cioè quando siamo connessi ad un certo dominio sono visibili ed accessibili soltanto le variabili relative a quel dominio (con un max di 10MB per dominio)

L'oggetto **sessionStorage** salva invece i dati soltanto nella memoria del browser, per cui vengono persi nel momento in cui il browser viene chiuso e risultano disponibili soltanto per la pagina che li ha creati.

All'interno degli oggetti localStorage e sessionStorage si possono salvare variabili (dette **item**) in formato chiave-valore. Le variabili vengono automaticamente salvate nell'area relativa al dominio corrente:

```
localStorage.setItem("key1", "value1");
var key1 = localStorage.getItem("key1");
localStorage.removeItem("key1");
localStorage.clear();  // ripulisce l'intera localStorage relativo al dominio
localStorage.length;   // restituisce il numero di chiavi memorizzate
```

Nota1

Attenzione al fatto che, avviando l'applicazione con webstorm, questo crea un server locale di accesso alle pagine. Per cui se il file all'interno del localStorage viene creato tramite una applicazione lanciata con webstorm, aprendo la stessa applicazione tramite file system con un doppio click, questa NON vedrà il file precedente all'interno del local Storage e viceversa.

Per cui o si utilizza sempre webstorm oppure si utilizza sempre il file system

Nota2

Per vedere se il browser supporta la localStorage si può utilizzare una delle seguenti condizioni:

```
if (typeof(localStorage) != "undefined") {
if('localStorage' in window && window['localStorage'] !== null) {
```

La trasmissione dei dati

I dati viaggiano attraverso la rete **sempre** in formato stringa.
Quando un client riceve una stringa di dati dalla rete, per poterli elaborare li deve trasformare in oggetto.
Quando invece deve trasmettere un oggetto in rete, prima di trasmetterlo lo deve trasformare in stringa.

- Il processo di trasformazione di una stringa in un oggetto si definisce **parsificazione** (parsing)
- Il processo inverso di trasformazione di un oggetto in stringa si definisce **serializzazione**

XML

Un documento XML è un documento strutturato a tag esattamente come i documenti HTML.
E' sostanzialmente costituito da un albero avente una **radice** (equivalente al tag **<html>**) con dei tag interni di **primo livello** che possono contenere al loro interno dei tag di **secondo livello** i quali possono contenere nodi di **terzo livello** e così via.

- Ogni tag può avere uno o più attributi.
- I valori terminali (cioè il contenuto finale di un nodo interno) sono detti **foglie** dell'albero

```
<bookstore>
      <book category="cooking">
            <title lang="en">Everyday Italian</title>
            <author>Giada De Laurentiis</author>
            <year>2005</year>
            <price>30.00</price>
      </book>
      <book category="children">
            <title lang="en">Harry Potter</title>
            <author>J K. Rowling</author>
            <year>2005</year>
            <price>29.99</price>
            </book>
      <book category="web">
            <title lang="en">XQuery Kick Start</title>
            <author>James McGovern</author>
            <author>Per Bothner</author>
            <author>Kurt Cagle</author>
            <author>James Linn</author>
            <author>Vaidyanathan Nagarajan</author>
            <year>2003</year>
            <price>49.99</price>
      </book>
      <book category="web" cover="paperback">
            <title lang="en">Learning XML</title>
            <author>Erik T. Ray</author>
            <year>2003</year>
            <price>39.95</price>
      </book>
</bookstore>
```

Parsificazione di una stringa XML in un oggetto XML

```
var xml = "<bookstore> ..... </bookstore>"

var parser=new DOMParser();
var xmlDoc=parser.parseFromString(xml,"text/xml");
```

In questo modo viene creato un nuovo oggetto **xmlDoc** con all'interno l'intera struttura contenuta nella variabile **xml** di tipo stringa. Se la variabile **xml** contiene l'intero albero precedente scritto come stringa, l'oggetto **xmlDoc** conterrà di conseguenza l'intero albero.

Serializzazione di un oggetto XML in stringa

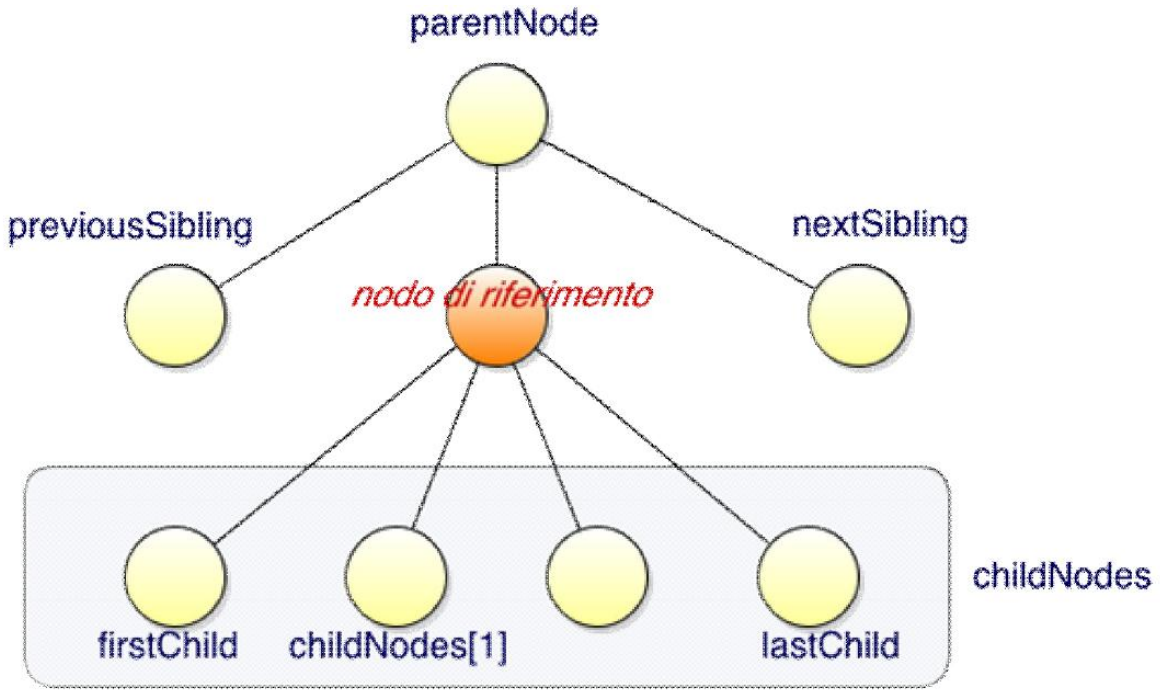
```
var serializer = new XMLSerializer();
var xml = serializer.serializeToString(xmlDoc);
```

Navigazione di un albero XML

Un oggetto XML è un albero esattamente come il DOM di una pagina HTML ed è pertanto navigabile tramite gli stessi metodi e le stesse proprietà.

Per scorrere un albero XML occorre utilizzare un puntatore che punterà inizialmente alla root e che poi scorrerà via via lungo tutti i nodi dell'albero.

Nell'immagine sotto rappresentata si ipotizza che il puntatore sia attualmente posizionato sul "nodo di riferimento" visualizzato in rosso



Accesso alla radice dell'albero

```
var root = xmlDoc.documentElement;
var root = xmlDoc.childNodes[0];
var root = xmlDoc.getElementsByTagName("bookstore")[0];
```

I vari Tipi di nodo

I vari nodi possono essere di diversi tipi. I principali sono i seguenti :

1 = ELEMENT_NODE = nodo vero e proprio
2 = ATTRIBUTE_NODE = attributo
3 = TEXT_NODE = nodo testuale, cioè foglia dell'albero (valore di un nodo ELEMENT)
8 = COMMENT_NODE = commento
9 = DOCUMENT_NODE = l'intero xmlDoc subito dopo il parsing

Proprietà per la navigazione dell'albero (e che restituiscono sempre un Nodo)

.~~childNodes~~	vettore dei nodi figli del nodo corrente. Accetta come parametro solo **l'indice numerico a base 0** Attenzione che però tiene in conto anche i white spaces.
.children	simile al precedente ma è disponibile soltanto sui nodi di tipo ELEMENT e conteggia come figli soltanto i nodi di tipo ELEMENT. Cioè NON contempla i white spaces e nemmeno i nodi di tipo TEXT (foglie). Preferibile nella maggior parte dei casi.
.~~firstChild~~	primo figlio generico del nodo corrente
.~~lastChild~~	ultimo figlio generico del nodo corrente
.parentNode	padre del nodo corrente
.previousSibling	fratello precedente
.nextSibling	prossimo fratello del nodo corrente

Notare che tutti questi metodi valgono anche per la navigazione del DOM.
Ad esempio se abbiamo un puntatore a **table** e vogliamo accedere al **tbody** interno (in assenza di head) potremmo scrivere:

```
_tbody = _table.childNodes[1];
_tbody = _table.children[0];
```

`_table.childNodes[0]` è un white space

Proprietà per l'accesso alle informazioni del nodo

.childNodes.length	numero dei figli del nodo corrente (compresi i whitespace)
.children.length e .childElementCount	escludono i white spaces
.hasChildNodes()	indica se il nodo corrente ha nodi figli
.nodeName	legge/imposta il nodeName di un tag
.tagName	come sopra
.nodeType	legge/imposta il nodeType. Es: if (node.nodeType == Node.TEXT_NODE)
.nodeValue	legge/imposta il nodeValue di un nodo foglia (TextNode)
.textContent	contenuto testuale (foglia) di un Element Node. Se un nodo con figli ha anche un TextContent, occorre tenerne conto nella navigazione
.innerHTML	contenuto testuale completo di un qualunque nodo

Metodi per la gestione degli attributi

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

Metodi per l'aggiunta / rimozione dei nodi

parentNode.appendChild(childNode)	consente di appendere nuovi nodi in coda
parentNode.insertBefore(newNode, childNode)	aggiunge newNode davanti a childNode
parentNode.removeChild(childNode)	consente di eliminare un nodo

Metodi per la creazione di nuovi nodi

xmlDoc.createElement("nodeName")

```
var riga = xmlDoc.createElement("elemento");
root.appendChild(riga);
var foglia = xmlDoc.createTextNode("salve mondo");  // Nodo di tipo foglia
riga.appendChild(foglia);
```

Il metodo **createTextNode()** è simile a .innerHTML e consente di aggiungere al nodo corrente un intero albero / sottoalbero xml scritto sotto forma di stringa.

Creazione di un nuovo oggetto xmlDoc

```
var xmlDoc = document.implementation.createDocument("", "", null);
```

Il primo parametro indica un eventuale namespace da anteporre al documento
Il secondo parametro rappresenta un ulteriore prefisso opzionale
Il terzo parametro indica il tipo di documento (Document Type)

```
var root = xmlDoc.createElement("root");
xmlDoc.appendChild(root);
```

Esempio 1

```
var root = xmlDoc.documentElement;
alert(root.childNodes.length);  // 4

var book = root.childNodes[0];
var title = book.childNodes[0];
alert (title.textContent);  // Everyday Italian

for (var i=0; i<root.childNodes.length;i++){
     var book = root.childNodes[i];
     var category =  book.getAttribute("category");
     var title = book.childNodes[0].textContent;
     var authors = "";
     for (var j=0; j<book.childNodes.length;j++){
          var field = book.childNodes[j];
          if(field.nodeName == "author")
              authors += field.textContent + "; ";   }
     alert (title + '\n' + category + '\n' + authors);
}
```

Esempio 2

Si può accedere direttamente ai singoli nodi anche utilizzando i soliti metodi java script getElementById etc

```
var title = xmlDoc.getElementsByTagName("title");
for (var i=0;i<title.length;i++) {
    document.write(title[i].childNodes[0].nodeValue);
    document.write("<br>");
}
```

Creazione di nuovi nodi all'interno della pagina HTML

```
var tab = document.getElementById("gridStudenti");
var riga = document.createElement("tr");
tab.appendChild(riga);
```

Vettori Associativi

Sono vettori che al posto dell'indice numerico usano una **chiave** alfanumerica (nell'esempio pippo, pluto e minnie).

```
var vect = new Array(); // oppure var vect=[] raccomandato perchè più veloce
vect['pippp']  = "descrizione di pippo";
vect['pluto']  = "descrizione di pluto";
vect['minnie'] = "descrizione di minnie";
```

I valori dei vettori associativi vengono salvati mediante una tecnica di hash legata alla chiave, per cui il principale vantaggio rispetto ai normali array enumerativi è la **possibilità di accesso diretto tramite chiave** a qualsiasi campo del vettore. Nel caso dei vettori enumerativi, per trovare una informazione occorre eseguire una ricerca (tipicamente sequenziale) all'interno del vettore. Nel caso invece degli array associativi il campo di interesse (es 'minnie') può essere acceduto in modo diretto:

```
alert(vect['minnie']);
```

E' anche possibile, una volta definito il vettore, definire alcune celle tramite chiave, ed altre tramite indice, cioè in pratica creare un vettore misto in cui il ciclo **`for in`** consentirà di scandire le celle associative, mentre il ciclo **`for of`** consetirà di scandire le celle enumerative. I due gruppi sono completamente distinti. Approccio misto comunque assolutamente sconsigliato.

Una oggetto simile in C# sono i Dictionary

Nota

Internamente i vettori associativi possono essere visti come matrici a due colonne del tipo seguente:

```
var vect2 = new Array();
vect2[0] = new Array ('pippo',  'descrizione di pippo');
vect2[1] = new Array ('pluto',  'descrizione di pluto');
vect2[2] = new Array ('minnie', 'descrizione di minnie');
alert(vect2[0][2]);
```

In realtà alla base dei vettori associativi c'è una tecnica di 'hash' non presente nelle matrici per cui scrivere

```
vect1['pippo']  = "descrizione di pippo";
vect2[0] = new Array ('pippo',  'descrizione di pippo');
```

non è esattamente la stessa cosa. Nel primo caso la descrizione viene posizionata in una locazione ben precisa (non necessariamente sequenziale) strettamente dipendente dalla chiave.
Per cui, per quanto simili, facendo `alert(vect2['pippo'])` il risultato sarà undefined,

JSON : Java Script Object Notation

Per la scrittura di un vettore associativo esiste anche una sintassi alternativa, molto compatta, che è la cosiddetta **Java Script Object Notation** (abbreviato JSON),
Le righe precedenti avrebbero potuto essere scritte anche nel modo seguente del tutto equivalente , in cui i vari campi sono scritti nel formato **`chiave : valore`** e separati da una virgola

```
var vect = {
   "pippo" : "descrizione di pippo",
   "pluto" : "descrizione di pluto",
   "minnie" : "descrizione di minnie"
};
```

Questa sintassi è abbastanza simile alle **`struct`** del C, però mentre una struttura definisce soltanto il tracciato di un record in cui **i dati** verranno inseriti successivamente, nel caso dei JSON struttura e dati vengono definiti contemporaneamente. **Praticamente si definisce un oggetto a partire dal suo contenuto, racchiudendo tra parentesi graffe le sue proprietà ed anche i suoi metodi.**
Si tratta sostanzialmente di un Object che viene dichiarato ed istanziato allo stesso momento.
Anche senza il new, la variabile vect contiene comunque (ovviamente) un **riferimento** all'oggetto.

Terminologia e sintassi

Si chiamano:

- **chiavi** le stringhe che rappresentano il nome della proprietà,
- **valori** gli elementi associati alle chiavi.

In molti casi una struttura come questa è chiamata *hashmap*, ed è caratterizzata da coppie chiave-valore

- **Le chiavi devono essere scritte con le virgolette doppie** (in taluni casi possono essere utilizzate anche le virgolette semplici o addirittura l'omissione delle virgolette, ma è fuori standard)
- **Per le chiavi non è consentito utilizzare il nome di una variabile.** Anche se si omettono le virgolette, il testo viene comunque sempre interpretato come stringa che definisce il nome del campo
- **I valori** possono essere scritti come stringhe, numeri, booleani oppure utilizzando il nome di un'altra variabile.

Esempio : anagrafica di uno studente

```
var studente = {
    "nome" : "mario",
    "cognome" : "rossi",
    "eta" : 16,
    "studente" : true,
    "images" : ["smile.gif", "grim.gif", "frown.gif", "bomb.gif"],
    "hobbies" : [],                    // vettore al momento vuoto
    "pos": { "x": 40, "y": 300 },      // oggetto annidato

    "stampa" :   function () { alert("Hello " + this.nome); },
    "fullName" : function () { return this.nome + " " + this.cognome; }
};
```

Accesso in lettura ai campi di un JSON

Si può accedere **al valore di una chiave** tramite la sintassi dei vettori associativi (parentesi quadre) oppure tramite la tipica sintassi degli oggetti con il puntino come separatore di campo.

```
var eta = studente["eta"]; // sintassi dei vettori (più generale)
var eta = studente.eta;    // sintassi degli oggetti (no PHP)
```

Il primo caso presenta il **vantaggio** di consentire un accesso parametrizzato al campo. Cioè, invece di specificare direttamente il nome del campo, è possibile utilizzare una variabile che contenga il nome del campo a cui intendiamo accedere :

```
var myVar = "cognome";
var nome = studente[myVar];      // "rossi"
var a = 1;
var nome = studente["nome"+a];   // accede al campo nome1
```

Se si cerca di accedere in lettura ad una chiave inesistente, il risultato sarà **null**

```
if(studente[key]!=null)  alert(studente[key]);
```

oppure :

```
if(key in studente)  alert(studente[key]);
```

oppure :

```
if(studente.hasOwnProperty(key))  alert(studente[key]);
```

Accesso in scrittura e aggiunta di nuove chiavi

```
studente["eta"]=18;  // sovrascrive il valore precedente
```

Per aggiungere una nuova chiave **NON è ammesso l'utilizzo del metodo `.push()`** (che vale solo per i vettori enumerativi) ma è sufficiente accedere in scrittura ad una chiave inesistente e la nuova chiave verrà aggiunta all'oggetto con il corrispondente valore. Vale anche per i metodi !

```
studente["indirizzo"]="Fossano";
studente.indirizzo="Fossano"; // sintassi non ammessa in TypeScript per creare nuove chiavi
```

Rimozione di una chiave

```
delete studente["eta"]
```

Dichiarazione di un Array (Associativo o Enumerativo) e di un JSON

```
1) var persona = new Array();  // vettore generico (associativo o enumerativo)
   var persona = [];  // equivalente

2) var persona = new Object();   // JSON (vettore associativo)
   var persona = {};  // equivalente
```

Le due sintassi non sono completamente equivalenti.
Per i JSON è preferibile la 2°, riservando la 1° per i vettori enumerativi.

Scansione dei campi di un JSON

Poiché l'accesso ai campi di un JSON può sempre essere eseguito in modo diretti attraverso le chiavi, la scansione dei campi di un JSON si rende necessaria molto raramente, ad esempio per eseguire delle stampe o dei log (che tra l'altro possono essere fatti anche utilizzando semplicemente la serializzazione).

In ogni caso una eventuale scansione può essere eseguita tramite il ciclo **`for`** `key` **`in`**:

```
for (var key in studente)
    alert(key + ' = ' + studente[key]);
```

Nota a differenza di C#, `key` non è un oggetto, ma una semplice stringa che contiene il nome della chiave. **Non** è pertanto possibile scrivere `alert(key.value)`
ma occorre scrivere `alert(key + ' = ' + studente[key]);`

Il vettore enumerativo delle chiavi

Il metodo statico **`Object.keys`** restituisce un vettore enumerativo di tutte le chiavi presenti all'interno di un vettore associativo:

```
var keys = Object.keys(studente);
var chiave = keys[0];
var valore = studente[chiave];
```

Le dimensioni di un object

Nel caso dei vettori associativi / object la proprietà **length** non è definita (restituisce *undefined*)**.** Infatti, trattandosi di un oggetto, non ha senso parlare della sua lunghezza. Volendo però valutare il numero dei campi presenti, si può utilizzare il **vettore enumerativo delle chiavi** che, essendo un vettore enumerativo, dispone della proprietà length:

```
if(Object.keys(studente).length != 0)  // oppure in jQuery;
jQuery.isEmptyObject({});   // true
```

Parsing e Serializzazione di uno stream JSON

Il metodo statico **JSON.parse(str)** consente di parsificare una stringa jSON convertendola in oggetto
Il metodo statico **JSON.stringify(obj)** consente di serializzare un oggetto o un vettore di oggetti nella stringa corrispondente.

```
// server
var json = {"name": "John Doe", "age": 42}
var s = JSON.stringify(json);
// ------ trasmissione dei dati al client ----------
// client
alert(s);
var json = JSON.parse(s);
```

In realtà il metodo **stringify** presenta la seguente sintassi completa:

```
var jsonText = JSON.stringify(json, null, 2);
```

Il 2° parametro è detto **rimpiazzo** e consente di applicare un filtro sui campi da visualizzare (o rimpiazzarli con altri valori)
Il 3° parametro è detto **formattatore** e consente di formattare il json in un formato **human readable** in cui ogni livello interno presenta un livello di indentazione pari al valore indicato (nell'esempio 2 chr)

Passaggio per riferimento e confronto fra JSON

A differenza delle variabili primitive, gli Object vengono passati alle funzioni per riferimento.
Questo vale anche per le assegnazioni che copiano il puntatore e NON l'intero record:

```
var studente2 = studente1;     // copia il riferimento
studente2.nome = "enrico";     // anche studente1.nome conterrà "enrico";
```

Allo stesso modo NON **è possibile eseguire un confronto diretto fra due json**
Il confronto verrebbe eseguito sui puntatori e NON sul contenuto !

Utilizzo e concatenamento dei Metodi

Per i metodi occorre scrivere **methodName: function(){},** dove methodName rappresenta sostanzialmente un riferimento che punta al metodo.

Esempi.

```
studente.stampa();       // Hello Mario
alert(studente.fullName());
```

I Metodi, per poter accedere a Proprietà e Metodi della classe, devono **obbligatoriamente** sempre utilizzare la parola chiave **`this`**.

Data una variabile che punta ad un metodo

```
var method =studente['stampa'];
```

sono consentite tutte le seguenti sintassi:

```
alert(method);              // Stampa il codice del metodo, cioè:
                               function() { alert("Hello" + this.name); }
alert(typeof(method));      // function
method();                   // Hello World
studente['stampa']();       // Hello World. Sintassi usata nel dispatch
```

Concatenamento

Un aspetto interessante e molto sfruttato dei **metodi** degli Object è quello di utilizzare all'interno dei metodi un **return this** finale. Questo fa si che il metodo, dopo aver eseguito le proprie operazioni, ritorni l'oggetto stesso, così che il chiamante possa richiamare in cascata un altro metodo della stessa classe:

```
studente.elabora().stampa().rilascia();
```

Le specifiche JSON

JSON rappresenta il formato standard attualmente utilizzato nello scambio di dati fra client e server. Rispetto ad XML, JSON infatti è decisamente più leggero.

Inoltre JSON ha una struttura tabellare e quindi risulta preferibile per dati aventi struttura tabellare, come lo sono quasi sempre i dati provenienti da un DB (SQL o noSQL).

XML continua a rimanere preferibile per dati aventi una struttura gerarchica

Le specifiche JSON sono definite all'interno dello standard **ECMA Script 5** (ES5).
Le principali regole sono le seguenti:

- Non è ammesso usare solo numeri come nome di chiave (*ovviamente*)
- I **nomi dei campi** DEVONO essere scritti come stringhe (cioè racchiusi tra doppi apici).
- Per le stringhe (chiavi e valori) NON sono ammessi gli apici singoli ma SOLO i **doppi apici**
  Ad esempio Express accetta SOLO stringhe json scritte in questo modo. Altri ambienti (es jQuery) accettano anche chiavi senza virgolette e ammettono l'utilizzo delle virgolette semplici.
- **Numeri** e **Booleani** devono essere scritti modo diretto (senza doppi apici). Attenzione che, aggiungendo i doppi apici, NON sono più NUMERI o BOOLEANI, ma diventano STRINGHE

Dati supportati: *http://www.json.org/json-it.html*

`null`
`interi, reali, booleani (true e false)` scritti senza i doppi apici. Per i decimali si utilizza il puntino. E' ammessa anche la virgola ma in tal caso occorre utilizzare gli apici doppi
`stringhe` **racchiuse da doppi apici**
`array(`sequenze di valori, separati da virgole e racchiusi in parentesi quadre`);`
`object(`sequenze **coppie** chiave-valore separate da virgole e racchiuse in parentesi graffe`)`
`array of objects` e qualunque altra forma composita

Esempi di jSon validi

```
var person = {
      "name"      : "Nicolas",
      "age"       : 22,
      "date"      : [01, 09, 1992],   // vettore
      "student"   : true,
      "info"      : {"web": "myPage.it", "mail": "myMail@vallauri.edu"}
};
```

```
var  persons = [
   { "name" : "Nicolas",  "age" : 22  },
   { "name" : "Piero",    "age" : 29  },
   { "name" : "Gianni",   "age" : 20  }
];
```

Anche le seguenti variabili sono considerati oggetti JSON validi:

```
var n = 5
var s = "salve mondo"
```

questo perché in realtà viene eseguito un **boxing** automatico del tipo:

```
var N = new Object(5);
var S = new Object("salve mondo");
```

In entrambi i casi, al momento del **`JSON.stringify()`**, l'intero contenuto verrà racchiusa all'interno di un ulteriore apice singolo, apice singolo che verrà poi rimosso al momento del `JSON.parse()`.

Unboxing

```
var s = N.toString();      // "72"
var n = N.valueOf();       //  72
```

Nota

Viceversa `var s = 'salve mondo'` non è considerato un JSON valido

Creazione automatica delle chiavi

Quando si vuole creare una chiave avente lo stesso nome della variabile da utilizzare per il value, per definire il JSON è possibile omettere la chiave, che assume automaticamente il nome della variabile.

`{nome, cognome}` equivale a `{"nome":nome, "cognome":cognome}`

jsonformatter

Il sito **`jsonformatter`** consente di validare l'esatta sintassi di una qualunque stringa jSon.

Il sito `http://www.httputility.net/`**`json-minifier`**`.aspx` consente interessanti conversioni di dati da XML a JSON e viceversa

Vettori di Object

Sono sostanzialmente analoghi ai **vettori di record** disponibili nei linguaggi strutturati, però nel caso dei json ogni record può avere una sua struttura diversa dalla struttura degli altri record:

```
var myArray = [];
myArray [0] = {"name":"Mario",  "age" : 33 };
myArray [1] = {"name":'Manlio", "residenza":"Fossano", "student":true};

for (let item of myArray)
     for (key in item)
          alert(key + ':' + item[key]);
```

Scansione di un vettore enumerativo di object

Il ciclo for-of con indice e item

Spesso, quando si utilizza un ciclo for-of, risulta necessario disporre anche dell'indice numerico i dell'item corrente. Per poter accedere anche all'indice occorre impostare il ciclo su **vet.entries**(). Molto comodo.

Esempio

```
let vet =["item1", "item2", "item3", "item4", "item6"]
for(const [i, item] of vet.entries())
    console.log(i, item)
```

Il ciclo forEach

Fa parte dei metodi del cosiddetto "java script funzionale" introdotto in ES6
E' ritenuto molto più veloce rispetto al ciclo ***for-of*** tradizionale.

Esempio 1:

```
let vet =["item1", "item2", "item3", "item4", "item6"]
vet.forEach(function(item) {
    console.log(item);
});
```

Esempio 2:

```
Object.keys(studente).forEach(function(key){
    console.log(key + ":" + studente[key])
});
```

Scansione di oggetti annidati

Come detto i campi di un JSON possono essere a loro volta JSON annidati anche su più livelli.

Nel momento in cui si vuole visualizzare in modo tabellare un JSON in cui le informazioni relative al record da visualizzare provengono da un vettore enumerativo interno, occorre eseguire un ciclo for annidato per ogni vettore enumerativo che si incontra a partire dalla struttura più esterna.

```
for (let item1 of mainArray) {
     for (let item2 of innerArray)
```

dove

`item1` consentirà di accedere alle chiavi della struttura di primo livello
`item2` consentirà di accedere alle chiavi della struttura di secondo livello

Ordinamento di un vettore enumerativo di object sulla base di una chiave

Supponendo che **myArray** sia un vettore enumerativo di object e che questi object dispongano di una chiave **myKey**, il seguente metodo consente di ordinare l'array sulla base del campo myKey

```
myArray.sort(function(record1, record2) {
        let str1 = record1.myKey.toUpperCase();
        let str2 = record2.myKey.toUpperCase();
        if (str1 < str2)
            return -1;
        else if (str1 > str2)
            return 1;
        else return 0;
    });
```

Appendice: L'oggetto MAP

Object e Map sono basate sullo stesso principio: salvano i dati in formato key:value

Object:
`{1: 'smile', 2: 'cry', 42: 'happy'}`

Map:
`{[1, "smile"], [2, "cry"], [42, "happy"]}`
Ogni item della map è costituito da un vettore enumerativo di due elementi: la chiave ed il valore

Differenze fra Object e Map

- Negli **Object** la chiave può essere soltanto un numero intero o una stringa. Nelle **Map** può essere qualunque cosa: un vettore, un altro object, etc.
- Nelle **Map** gli elementi sono ordinati sulla base dell'ordine di inserimento. Negli **Object** no.
- **Map** è una sottoclasse derivata da **Object**. Object rappresenta invece la superclasse da cui eredita Map. Per cui Map è una istanza di Object, mentre Object NON è una istanza di Map
- Un **Object** può essere istanziato in diversi modi:
  ```
  var obj = {};        //Empty object
  var obj = {id:1, name:"Test object"};
  var obj = new Object(); //Empty Object
  ```
- Una **Map** può essere istanziata soltanto tramite costruttore che si aspetta come parametro un vettore enumerativo di coppie `[key,value]`
  ```
  var map = new Map(); //Empty Map
  var map = new Map([[1,2],[2,3]]); // map = {1=>2, 2=>3}
  ```

Quando è preferibile Object e quando Map

- L'**object** è la scelta ideale quando c'è bisogno di una struttura semplice e veloce, perché la creazione di un oggetto e l'accesso alle proprietà tramite una chiave specifica sono **molto più veloci** della creazione di Mappa e del relativo accesso
- L'**object** è preferibile anche laddove è necessario applicare una logica separata a una singola proprietà / elemento della collezione

```
var obj = {
    id: 1,
    name: "It's Me!",
    print: function(){
        return `Object Id: ${this.id}, with Name: ${this.name}`;
    }
}
```

Questo non è fattibile con le Map

- **Map** è preferibile in scenari che richiedono molte aggiunte e rimozioni di nuove chiavi e per grandi quantità di dati. Inoltre Map, a differenza di Object, mantiene l'ordine delle chiavi e quindi garantisce prestazioni stabili in tutti i browser.

Principali metodi dell'oggetto Map

```
map.get(1)        // 2
map.has(1);       // return true if key exists
map.set(4,5);     // {1=>2, 2=>3, 4=>5}
var ok = map.delete(1);  //{ 2=>3, 4=>5}
console.log(ok); // true
map.clear();      // remove ALL elements from a Map object
console.log(map.size);  // number of keys
```

Scansione di una Map

Le mappe sono iterabili, cioè supportano il metodo for of

```
for (const item of map){
    console.log(item);
    // Array[2,3]
    // Array[4,5]
}
```

oppure

```
for (const [key,value] of map){
    console.log(`key: ${key}, value: ${value}`);
    // key: 2, value: 3
    // key: 4, value: 5
}
```

oppure

```
map.forEach((value, key) => console.log(`key:${key}, value:${value}`))
    // key: 2, value: 3
    // key: 4, value: 5
```