

**La rappresentazione dei dati nel web: XML e JSON**

Rev Digitale 3.0 del 15/10/2019

**La rappresentazione dei dati nel web**

---

Charset .....	2
Internet Media Types .....	2
Gli oggetti LocalStorage e SessionStorage .....	3
XML .....	4
Navigazione di un albero XML .....	5
Vettori Associativi .....	8
Java Script Object Notation .....	8
Utilizzo dei metodi .....	11
Le specifiche di JSON .....	12
Vettori di Object .....	13
Note sugli Oggetti in js .....	14
L'oggetto MAP .....	15

## 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](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 **parificazione** (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 parser=new DOMParser();
var xml = "<bookstore> </bookstore>";
var xmlDoc=parser.parseFromString(xml,"text/xml");
```

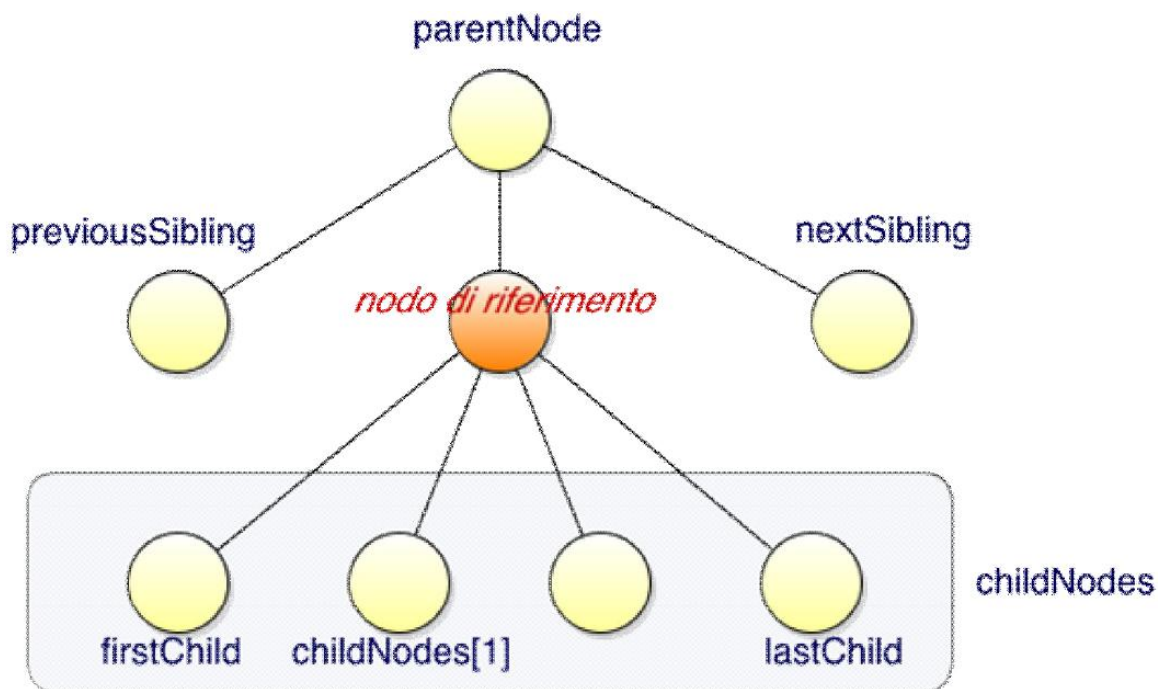
In questo modo viene creato un nuovo oggetto **xmlDoc** contenente soltanto la radice di un albero vuoto. Se invece la variabile **xml** contenesse l'intero albero precedente scritto come stringa, l'oggetto **xmlDoc** conterrebbe di conseguenza l'intero albero.

### 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.



### 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

### Accesso alla radice dell'albero

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

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

<code>.childNodes</code>	vettore dei nodi figli del nodo corrente. Accetta come parametro solo l' <b>indice numerico a base 0</b> Attenzione che però tiene in conto anche i white spaces.
<code>.children</code>	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.
<code>.firstChild</code>	primo figlio generico del nodo corrente
<code>.lastChild</code>	ultimo figlio generico del nodo corrente
<code>.parentNode</code>	padre del nodo corrente
<code>.previousSibling</code>	fratello precedente
<code>.nextSibling</code>	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 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

<code>.childNodes.length</code>	numero dei figli del nodo corrente (compresi i whitespace)
<code>.children.length</code> e <code>.childElementCount</code>	escludono i white spaces
<code>.hasChildNodes()</code>	indica se il nodo corrente ha nodi figli
<code>.nodeName</code>	legge/imposta il nodeName di un tag
<code>.tagName</code>	come sopra
<code>.nodeType</code>	legge/imposta il nodeType. Es: if (node.nodeType !== Node.TEXT_NODE)
<code>.nodeValue</code>	legge/imposta il nodeValue di un nodo foglia (TextNode)
<code>.textContent</code>	contenuto testuale (foglia) di un Element Node. Se un nodo con figli ha anche un TextContent, occorre tenerne conto nella navigazione
<code>.innerHTML</code>	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.removeAttribute(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.createElement("salve mondo"); // Nodo di tipo foglia  
riga.appendChild(foglia);
```

Il metodo `createElement()` 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['pippo'] = "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** consentirà 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

In Java Script i vettori associativi possono essere scritti anche in un modo alternativo molto comodo, cioè utilizzando la cosiddetta Java Script **Object Notation** :

Le righe precedenti avrebbero potuto essere scritte anche nel modo seguente del tutto equivalente :

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



in cui i vari campi sono scritti nel formato **nome : valore** e separati da una virgola. **Praticamente si definisce un oggetto a partire dal suo contenuto**, racchiudendo tra parentesi graffe le sue **proprietà ed anche i suoi metodi**. A differenza degli oggetti utilizzati nella programmazione tradizionale, gli Object Jva Script vengono dichiarati ed istanziati allo stesso momento. Non è necessario il new. In realtà la variabile contiene comunque un **riferimento** all'oggetto, esattamente come nella programmazione strutturata.

### 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 possono essere scritte indifferentemente

- senza virgolette (cioè come **identificatore**)
- **con le virgolette** (cioè come **stringa**), preferibile ed obbligatorio in jSon  
Le virgolette sono obbligatorie quando il nome non segue le regole per i nomi delle variabili (cioè se contiene spazi o caratteri speciali come ad esempio il trattino: es "Content-Type")
- I valori possono essere scritto come stringhe oppure utilizzando il nome di un'altra variabile.
- **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

### 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 diretto al valore delle proprietà

---

Si può accedere **direttamente** ai singoli valori in 2 modi:

```
var eta = studente['eta']; // sintassi dei vettori associativi (preferibile)
var eta = studente.eta;   // Object Notation (no PHP)
```

Nel primo caso la chiave di accesso **deve obbligatoriamente** essere scritta come stringa (**con le virgolette**). Lo scopo è quello di consentire un accesso parametrizzato tramite variabile che contiene il nome del campo a cui intendiamo accedere :

```
var myVar = "cognome";
var a = 1;
var nome = studente[myVar]; // "rossi"
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.

```
studente['indirizzo']="Fossano"; // vale anche per i metodi !
```

---

### Rimozione di una chiave

```
delete studente['eta']
```

---

### Sintassi di Dichiarazione di un Object

```
var persona = new Object(); // sconsigliato
var persona = {}; // più veloce
var persona = new Object({"nome" : "Mario", "cognome" : "Rossi"});
var persona = {"nome" : "Mario", "cognome" : "Rossi"};
var persona = new Array(); // sconsigliato
var persona = []; // più veloce
```

Attenzione che le ultime due righe non sono esattamente equivalenti alle precedenti. Anche se tutto sembra funzionare allo stesso identico modo, non è possibile ad esempio serializzare la variabile tramite il metodo `.stringify` che restituisce `[]`. In questo caso occorre fare un ciclo come per tutti i vettori.

---

### Scansione delle proprietà di un oggetto

E' possibile scandire il contenuto di un Object tramite il ciclo **for** **key** **in**:

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

Notare che **key**, a differenza di C#, 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 delle chiavi

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

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

## Il metodo forEach

---

```
Object.keys(studente).forEach(function(key) {  
    console.log('key': key, value: studente[key])  
});
```

Il ciclo forEach precedente è preferibile rispetto alla scansione **for** - **in** in quanto molto più veloce

E' utilizzabile anche sui vettori enumerativi:

```
var a = ['a', 'b', 'c'];  
a.forEach(function(element) {  
    console.log(element);  
});
```

## Le dimensioni di un object

---

Nel caso dei vettori associativi / object la proprietà **length** non è definita (restituisce sempre **zero**). Trattandosi di un oggetto non ha senso parlare della sua lunghezza, eventualmente intesa come numero di property presenti. Utilizzando però il vettore delle chiavi è possibile conteggiare il numero di property contenute nel vettore:

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

## Utilizzo dei Metodi

---

Per i metodi occorre scrivere **methodName: function() {}**, dove methodName rappresenta sostanzialmente un riferimento che punta al metodo. I Metodi, per poter accedere alle Properties della classe, devono **obbligatoriamente** utilizzare la parola chiave **this**. Un metodo può invece accedere ad altri metodi anche senza il **this** (esclusi però quelli dichiarati tramite prototype che richiedono il **this**). In pratica la cosa migliore è quella di utilizzare sempre il **this**.

Esempi.

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

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 di metodi

---

Un aspetto interessante e molto sfruttato dei **metodi** degli Object è quello di utilizzare all'interno dei metodi un **return this** finale. Questo fa sì 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

**Formato molto utilizzato per lo scambio di dati fra applicazioni client-server, costituito da uno o più Object serializzati.** Formato destinato a sostituire XML/XSLT, rispetto al quale è decisamente più leggero. Inoltre

- XML ha una struttura intrinseca gerarchica, dunque è preferibile per dati aventi una struttura gerarchica
- JSON ha invece 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).

Specifiche :

- 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** possono essere scritti anche in modo diretto (con o senza doppi apici)

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

**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

## Parsing e Serializzazione di uno stream JSON

Lo standard **ECMA Script 5** che definisce le specifiche di Java Script ha definito un nuovo oggetto **JSON** pienamente supportato dall'engine V8 di google chrome (e quindi anche da Node js).

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.

```
var s = '{ "name": "John Doe", "age": 42 }';  
var json = JSON.parse(s);  
alert(json.name);  
var jsonText = JSON.stringify(json);
```

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

```
var jsonText = JSON.stringify(json, null, "\t");
```

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 al meglio l'oggetto serializzato.

### Esempi di stringa jSon valide

---

```
var persona = { "nome" : "pippo" };

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

var persone = [
  { "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"
```

In entrambi i casi, al momento del `JSON.stringify()`, la variabile viene racchiusa all'interno di un apice singolo che verrà eliminato al momento del `JSON.parse()`.

Viceversa se si scrive `JSON.parse("ciao")` ==> il risultato sarebbe `ciao` ==> errore

Quando il nome della chiave è uguale al nome della variabile è possibile omettere la chiave, che assume automaticamente il nome della variabile.

```
{nome, cognome} === {"nome":nome, "cognome":cognome}
```

### Vettori di Object

Sono sostanzialmente analoghi ai vettori di record disponibili nei comuni linguaggi di programmazione

```
var myArray = [];

myArray [0] = {
  'name' : 'Mario',
  'surname' : 'Rossi',
  'age' : '33'
};

myArray [1] = {
  'name' : 'Giuseppe',
  'surname' : 'Verdi',
  'age' : '42'
};

for (i=0; i<myArray.length; i++)
  for (key in myArray [i])
    alert(key + ' = ' + myArray[i][key]);
```

### Ordinamento di un vettore di object sulla base di una certa chiave

Supponendo che **myArray** sia un vettore enumerativo di object e che questi object dispongano di una chiave **key1**, il seguente metodo consente di ordinare l'array sulla base del campo key1.

```
myArray.sort(function(a, b) {  
    let str1 = a.key1.toUpperCase();  
    let str2 = b.key1.toUpperCase();  
    if (str1 < str2)  
        return -1;  
    else if (str1 > str2)  
        return 1;  
    else return 0;  
});
```

### Note sugli oggetti js

- Non è possibile visualizzare il contenuto di un object usando un semplice **alert**. Gli elementi dell'array associativo sono trattati dal motore JavaScript come le proprietà di un oggetto, per cui è necessario eseguire una serializzazione.  
In realtà alcuni browser, facendo **alert(studente)**; mostrano tutte le voci, una per riga, con il relativo valore. Però all'interno dell'alert **NON** si possono eseguire concatenamenti, perché altrimenti invece del contenuto viene stampato l'indirizzo a cui punta il riferimento studente

- A differenza delle variabili primitive, gli Object vengono passati alle funzioni per riferimento. Questo vale anche per le assegnazioni che copiano il puntatore:

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

Dal momento che l'object viene passato per riferimento, le sottofunzioni possono modificare i valori delle Property

- Al costruttore Object è possibile passare il risultato di una qualsiasi espressione java script:

```
var N = new Object(72);  
var S = new Object("salve mondo");
```

Queste istanze creano un oggetto a partire da un dato primitivo (boxing, esattamente come in java). Object espone i metodi toString() e valueOf() che eseguono sostanzialmente l'unboxing():

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

## 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
```