

Javascript: fondamenti, concetti, modello a oggetti

JAVASCRIPT

- Un linguaggio di **scripting**
Come tale è interpretato, non compilato
- Storia:
 - Definito in origine da Netscape (*LiveScript*) - nome modificato in Javascript dopo accordo con Sun nel 1995
 - Microsoft lo chiama JScript (minime differenze)
 - Riferimento: standard ECMAScript 262
- **Object-based** (ma non object-oriented)
 - *assomiglia alla "grande famiglia" C, Pascal, Java...*
 - *ma di Java ha "relativamente poco"*
- I programmi Javascript si inseriscono direttamente nel sorgente HTML delle pagine Web (*anche più di un programma nella stessa pagina*)

LA PAGINA WEB

```
<HTML>
  <HEAD> <TITLE>...</TITLE> </HEAD>
  <BODY>
    ...
    <SCRIPT Language="JavaScript">
      <!-- Per non confondere i vecchi browser
           ... il programma Javascript ...
      // Commento Javascript per non confondere browser -->
    </SCRIPT>
  </BODY>
</HTML>
```

Una pagina HTML può contenere più tag `<script>`.

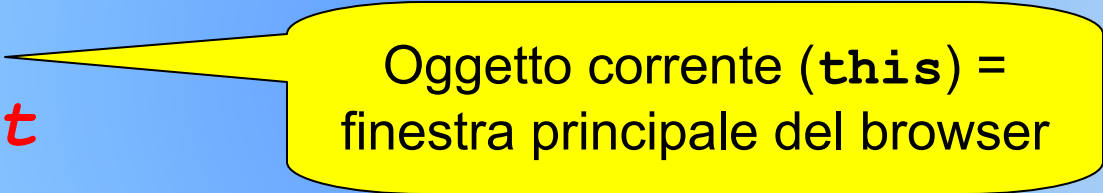
Document Object Model (DOM)

- Javascript è un linguaggio che fa riferimento al **Modello a Oggetti dei Documenti (DOM)**
- Secondo tale modello, *ogni documento ha la seguente struttura gerarchica:*

window

document

...



Oggetto corrente (**this**) =
finestra principale del browser

- Qui l'oggetto **window** rappresenta *l'oggetto corrente*, il sotto-oggetto *document* rappresenta *il contenuto*
- **window** $\Leftrightarrow$ pagina web (finestra) attuale
document $\Leftrightarrow$ *contenuto* della pagina web attuale

L'OGGETTO document

- L'oggetto **document** rappresenta *il documento corrente*, cioè *il contenuto* della pagina web attuale (*da non confondere con la finestra corrente!*)
- Sull'oggetto **document** è possibile invocare vari *metodi*: il metodo **write** stampa un valore a video
- stringhe, numeri, immagini, ...
- Esempi:

```
document.write("paperone")  
document.write(18.45 - 34.44)  
document.write('paperino')  
document.write('<IMG src="image.gif">')
```
- NB: this (riferito all'oggetto window) è sottinteso:
`document.write` equivale a this.`document.write`

L'OGGETTO `window` (1/2)

- L'oggetto **`window`** è *la radice della gerarchia DOM* e rappresenta *la finestra del browser*
- Fra i metodi dell'oggetto **`window`** vi è **`alert`**, che *fa apparire una finestra di avviso con il messaggio dato*
- **Esempio:**

```
x = -1.55; y = 31.85
```

```
alert("Somma di " + x + " e " + y + ": " + (x+y));
```

equivale a `this.alert(...)`

La funzione **`alert`** è ovviamente usabile anche in un link HTML.

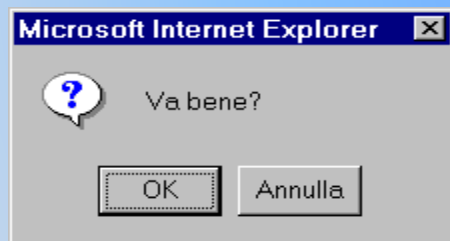
Alert non restituisce nulla
Se la si valuta → `undefined`



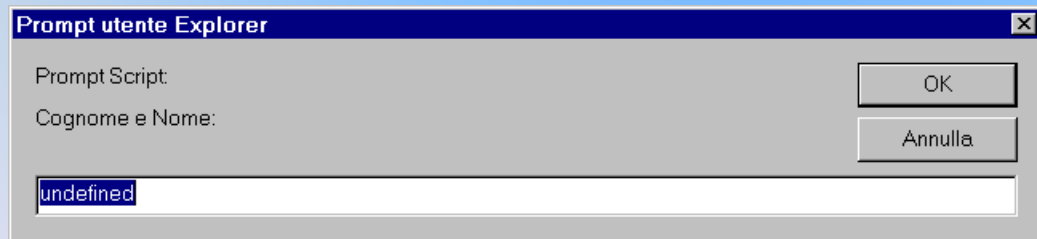
L'OGGETTO `window` (2/2)

Altri metodi dell'oggetto `window` sono:

- `confirm`, che fa apparire una *finestra di conferma* con il messaggio dato
- `prompt`, che fa apparire una *finestra di dialogo per immettere un valore*



`confirm` restituisce un valore *boolean* (*true* se si è premuto OK, *false* se si è premuto Annulla)



`prompt` restituisce un valore *string*

IL MODELLO DOM

Componenti principali di **window**

self

window

parent

top

navigator

plugins (array), **navigator**, **mimeType**s (array)

frames (array)

location

history

document

...segue intera gerarchia di sotto-oggetti...

L'OGGETTO document

Componenti principali di **document**

forms (array)

elements (array di *Buttons, Checkbox, etc etc...*)

anchors (array)

links (array)

images (array)

applets (array)

embeds (array)

Esempio: se il file HTML contiene un'immagine "image0":

document.image0.width

fa apparire la larghezza dell'immagine. Per cambiarla:

document.image0.width = 40

Javascript: stringhe

- Le **stringhe** possono essere delimitate sia da **virgolette** sia da **apici singoli**
- Se occorre annidare virgolette e apici, occorre alternarli - Ad esempio:

```
document.write('<IMG src="image.gif">')  
document.write("<IMG src='image.gif'>")
```

- Le stringhe si possono concatenare con +
`document.write("paolino" + 'paperino')`
- Le stringhe Javascript sono *oggetti dotati di proprietà*, tra cui **length** ...
- ...e di *metodi*, tra cui **substring(first, last)**
(*last* è l'indice del primo escluso, come in Java)

Javascript: Costanti e Commenti

- Le **costanti numeriche** sono sequenze di caratteri numerici non racchiuse da *virgolette* o *apici*
- *Il loro tipo è number*
- Le **costanti booleane** sono *true* e *false*
- *Il loro tipo è boolean*
- Altre costanti sono *null*, *NaN* e *undefined*
undefined indica **un valore indefinito**
- I commenti in Javascript sono come in Java:
// commento su riga singola
/ ... commento su più righe ... */*

Javascript: Espressioni

- Sono consentite sostanzialmente le stesse espressioni lecite in Java
 - espressioni numeriche: somma, sottrazione, prodotto, divisione (sempre fra reali), *modulo*, *shift*, ...
 - espressioni condizionali con `? ... :`
 - espressioni stringa: concatenazione con `+`
 - espressioni di assegnamento: con `=`
- Esempi:

```
document.write(18/4)
document.write(18%2)
document.write("paolino" + 'paperino')
```

Javascript: Variabili

- Le **variabili** in Javascript sono loosely typed
- *È possibile assegnare a una stessa variabile prima un valore stringa, poi un numero, poi altro ancora !*
- Ad esempio:

```
alfa = 19  
beta = "paperino"  
alfa = "zio paperone" // tipo diverso!!
```
- Sono consentiti *incrementi, decrementi e operatori di assegnamento estesi* (++, --, +=, ...)

Variabili e Scope

Lo **scope** delle **variabili** in Javascript è

- globale, per le variabili definite fuori da funzioni
- locale, per le variabili definite esplicitamente dentro a funzioni (compresi i parametri ricevuti)

ATTENZIONE: *un blocco NON delimita uno scope !*

- tutte le variabili definite fuori da funzioni, *anche se dentro a blocchi innestati*, sono globali

```
x = '3' + 2; // la stringa '32'
{
  { x = 5 } // blocco interno
  y = x + 3; // x denota 5, non "32"
}
```

Tipo dinamico

- Il tipo (dinamico) di una espressione (incluse le variabili) è ottenibile tramite **l'operatore *typeof***:

<code>typeof (18/4)</code>	dà <i>number</i>
<code>typeof ("aaa")</code>	dà <i>string</i>
<code>typeof (false)</code>	dà <i>boolean</i>
<code>typeof (document)</code>	dà <i>object</i>
<code>typeof (document.write)</code>	dà <i>function</i>

- Si parla di tipo *dinamico* perché, nel caso di variabili, **il tipo restituito da *typeof*** è quello corrispondente al *valore attuale* della variabile (o dell'oggetto...):

<code>a = 18;</code>	<code>typeof (a)</code> dà <u><i>number</i></u>
<code>...</code>	
<code>a = "ciao";</code>	<code>typeof (a)</code> dà <u><i>string</i></u>

Javascript: Istruzioni

- Le **istruzioni** devono essere separate una dall'altra o da un fine riga, o da un punto e virgola (similmente al Pascal e diversamente dal C o da Java)

- Esempio:

```
alfa = 19 // fine riga  
beta = "paperino" ; gamma = true  
document.write(beta + alfa)
```

- Come in Java, la concatenazione fra stringhe e numeri comporta la conversione automatica del valore numerico in stringa (*attenzione...*)

```
document.write(beta + alfa + 2)  
document.write(beta + (alfa + 2))
```

Javascript: Strutture di controllo

- Javascript dispone delle usuali strutture di controllo: **if, switch, for, while, do/while**
- In un **if**, le **condizioni booleane** esprimibili sono le solite (**=**, **!=**, **>**, **<**, **>=**, **<=**) con gli operatori logici AND (**&&**), OR (**||**), NOT (**!**)
- Anche i cicli **for, while, do/while** funzionano nel solito modo
- Esistono poi *strutture di controllo particolari*, usate per operare sugli oggetti:
 - **for ... in ...**
 - **with**

Javascript: Definizione di Funzioni

- Le **funzioni** sono introdotte dalla keyword **function**
- Possono essere sia **procedure**, sia **funzioni in senso proprio** (non esiste la keyword `void`)
- I parametri formali sono **senza dichiarazione di tipo**
- Al solito, il corpo va racchiuso in un blocco
- Esempi:

```
function sum(a,b) { return a+b }
```

funzione

```
function printSum(a,b) {  
  document.write(a+b)  
}
```

procedura (non ha
istruzione `return`)

Funzioni: chiamata

- Le **funzioni** sono chiamate nel solito modo, fornendo la lista dei **parametri attuali**
- Esempi:

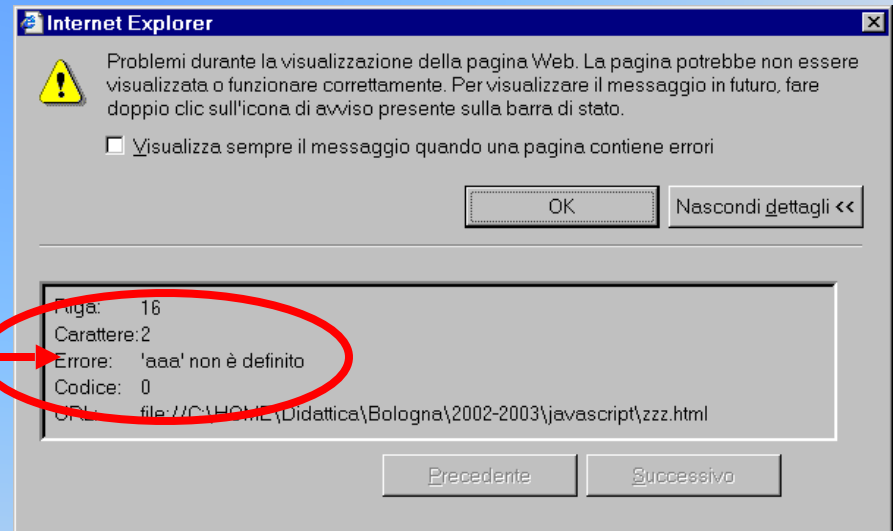
```
document.write("Somma: " + sum(18,-3) + "<br/>" )
```

```
printSum(19, 34.33)
```

tag HTML per andare a capo

- Se i tipi non hanno senso per le operazioni fornite, l'interprete Javascript dà **errore a runtime**

```
printSum(19, aaa)
```



Funzioni: parametri

- La lista dei **parametri attuali** può *non corrispondere in numero ai parametri formali* previsti.
- Se i **parametri attuali** sono **più** di quelli necessari, quelli extra sono *ignorati*
- Se i **parametri attuali** sono **meno** di quelli necessari, quelli mancanti sono **inizializzati ad undefined** (una costante di sistema)
- Il **PASSAGGIO** dei **PARAMETRI** avviene **per valore** (nel caso di oggetti, ovviamente si copiano riferimenti)
- **A differenza di C e Java, è lecito definire una funzione dentro un'altra funzione** (similmente al Pascal)

Variabili: tipi di dichiarazione

In Javascript, la **dichiarazione** delle variabili è

- **implicita o esplicita**, per variabili **globali**
- **necessariamente esplicita**, per variabili **locali**

La dichiarazione **esplicita** si effettua tramite la *parola chiave* **var**

```
var pippo = 19      // dichiarazione esplicita
```

```
pluto = 18           // dichiarazione implicita
```

- La dichiarazione **implicita** introduce sempre e solo variabili **globali**
- La dichiarazione **esplicita** ha un effetto che dipende da dove si trova la dichiarazione.

Variabili: dichiarazione esplicita

- Fuori da funzioni, la parola chiave **var** è ininfluente: la variabile definita è comunque **globale**.
- All'interno di funzioni, invece, la parola chiave **var** ha un ben preciso significato: indica che la nuova variabile è **locale**, ossia ha come **scope la funzione**.
- All'interno di funzioni, una dichiarazione **senza** la parola chiave **var** introduce una variabile **globale**

```
x=6 // globale
function test(){
    x = 18 // globale
}
test()
// qui x vale 18
```

```
var x=6 // globale
function test(){
    var x = 18 // locale
}
test()
// qui x vale 6
```

Variabili: environment di riferimento

- Usando una variabile già dichiarata l'environment di riferimento è innanzitutto quello locale: solo se ivi la variabile non è definita si accede a quello globale.

`f = 3` *// fa riferimento alla `f` locale, se esiste
 *// o alla `f` globale, se non esiste locale**

- Esempio in ambiente globale

var `f = 4` *// `f` è comunque globale*
`g = f * 3` *// `g` è comunque globale, e vale 12*

- Esempio in ambiente locale (dentro a funzioni)

var `f = 5` *// `f` è locale*
`g = f * 3` *// `g` è globale, e vale 15*

Funzioni e Chiusure (1/3)

La natura interpretata di Javascript e l'esistenza di un ambiente globale pongono una domanda:

quando una funzione usa un simbolo non definito al suo interno, *quale definizione vale per esso?*

- la definizione che esso ha nell'ambiente in cui la funzione è *definita*, oppure
- la definizione che esso ha nell'ambiente in cui la funzione è *chiamata*?

Per capire il problema, si consideri l'esempio seguente.

Funzioni e Chiusure (2/3)

Si consideri il seguente programma Javascript:

```
var x = 20;  
function provaEnv(z) { return z + x; }  
alert(provaEnv(18)) // visualizza certamente 38  
function testEnv() {  
    var x = -1;  
    return provaEnv(18); // COSA visualizza ???  
}
```

Nella funzione testEnv *si ridefinisce il simbolo x*, poi si invoca la funzione **provaEnv**, che *usa il simbolo x* ...
ma QUALE x?

Nell' ambiente in cui provaEnv è definita, il simbolo x aveva un altro significato rispetto a quello che ha ora!

Funzioni e Chiusure (3/3)

```
var x = 20;  
function provaEnv(z) { return z + x; }  
function testEnv() {  
  var x = -1;  
  return provaEnv(18); // COSA visualizza ???  
}
```

Se vale l'ambiente esistente all'atto dell'uso di provaEnv, si parla di chiusura dinamica; se prevale *l'ambiente di definizione* di provaEnv, si parla di chiusura lessicale.

Come tutti i linguaggi moderni, **Javascript adotta la chiusura lessicale** → testEnv visualizza ancora 38 (non 17)

Funzioni come dati

In Javascript, le *variabili* possono *riferirsi a funzioni*:

```
var f = function (z) { return z*z; }
```

- la funzione in sé non ha nome (anche se può averlo)
- la funzione viene *invocata tramite il nome della variabile che si riferisce ad essa*:

```
var result = f(4);
```

- un assegnamento del tipo $g = f$ produce *aliasing*
- Questo permette in particolare di *passare funzioni come parametro ad altre funzioni*; ad esempio:

```
function calc(f, x) {return f(x); }
```

Se f cambia, $calc$ calcola una diversa funzione.

Funzioni come dati - Esempi

Test:

```
function calc(f, x) { return f(x) }
```

calc(Math.sin, .8) dà 0.7173560908995228

calc(Math.log, .8) dà -0.2231435513142097

Attenzione, però:

calc(x*x, .8) dà errore perché $x*x$ non è un oggetto funzione del programma

calc(funz, .8) va bene solo se la variabile *funz* fa riferimento a un costrutto *function*

calc("Math.sin", .8) dà errore perché *una stringa non è una funzione*

Non confondere *il nome di una funzione* con *la funzione*!

Funzioni come dati - Conseguenze

Dunque

- per utilizzare una funzione come dato occorre avere effettivamente un oggetto funzione, non solo il nome!
- per questo non si può sfruttare questa caratteristica per far eseguire una funzione di cui sia noto solo il nome, ad esempio perché letto da tastiera:

`calc("Math.sin", .8)` dà errore

- o di cui sia noto solo il codice, come nel caso:

`calc(x*x, .8)` dà errore

Il problema è risolvibile, ma non così:

- o si accede alla funzione tramite le proprietà dell'oggetto globale
- o si costruisce esplicitamente un "oggetto funzione" opportuno.

Oggetti

- Un **oggetto** Javascript è una *collezione di dati* dotata di nome: ogni dato è interpretabile come una proprietà
- Per accedere alle proprietà si usa, al solito, la "dot notation": *nomeOggetto.nomeProprietà*
Tutte le proprietà sono accessibili (salvo trucchi...)
- Un **oggetto** Javascript è costruito da una *speciale funzione, il costruttore, che stabilisce la struttura dell'oggetto e quindi le sue proprietà.*
- I costruttori *sono invocati mediante l'operatore new*
- In Javascript *non esistono classi*
Il nome del costruttore è quindi a scelta dell'utente
(in certo qual modo, indica implicitamente la "classe")

Oggetti: Definizione

- La **struttura** di oggetto Javascript viene **definita** dal **costruttore** usato per crearlo.
- **È all'interno del costruttore che si specificano le proprietà (iniziali) dell' oggetto, elencandole** con la dot notation e la keyword **this**

- **Esempi:**

identificatore `Point` globale

```
Point = function(i,j) {  
    this.x = i;  
    this.y = j;  
}
```

identificatore locale

```
function Point(i,j) {  
    this.x = i;  
    this.y = j;  
}
```

- La keyword **this** è **necessaria**, altrimenti ci si riferirebbe all'environment locale della funzione costruttore.

Oggetti: Costruzione

- Per **costruire oggetti** si applica ***l'operatore new a una funzione costruttore***:

```
p1 = new Point(3,4) ;  
p2 = new Point(0,1) ;
```

- **ATTENZIONE:** l'argomento di new non è il nome di una classe, è solo il nome di una funzione costruttore.
- A partire da Javascript 1.2, si possono **creare oggetti** anche ***elencando direttamente le proprietà con i rispettivi valori*** mediante una **sequenza di coppie nome:valore** separate da virgole:

```
p3 = { x:10, y:7 }    (... ma di che "classe" è ?)
```

La sequenza di coppie è racchiusa fra parentesi graffe.

Oggetti: Accesso alle proprietà

- Poiché *tutte le proprietà di un oggetto sono pubbliche e accessibili*, per accedere basta la "dot notation:

`p1.x = 10;` *// da (3,4) diventa (10,4)*

(in realtà, esistono anche proprietà "di sistema" e come tali non visibili, né enumerabili con gli appositi costrutti)

- IL COSTRUTTO **with**

Nel caso si debba accedere a parecchie proprietà o metodi *di un medesimo oggetto*, per evitare di ripetere il nome dell'oggetto conviene usare il **costrutto with**

with (p1) x=22, y=2

equivale a p1.x=22, etc

with (p1) {x=3; y=4}

equivale a p1.x=3, etc

Aggiunta e rimozione di proprietà

- *Le proprietà specificate nel costruttore non sono le uniche che un oggetto può avere: sono semplicemente quelle iniziali*
- *È possibile aggiungere dinamicamente nuove proprietà semplicemente nominandole e usandole:*

`p1.z = -3;` *// da {x:10, y:4} diventa {x:10, y:4, z: -3}*

NB: non esiste il concetto di classe come "specifica della struttura (fissa) di una collezione di oggetti", come in Java o C++

- *È anche possibile rimuovere dinamicamente proprietà, mediante l'operatore delete :*

`delete p1.x` *// da {x:10, y:4, z: -3} diventa {y:4, z: -3}*

Metodi per (singoli) oggetti

- Definire *metodi* è semplicemente *un caso particolare dell'aggiunta di proprietà*
- Non esistendo il concetto di classe, un metodo viene definito per uno specifico oggetto (ad esempio, **p1**):
non per tutti gli oggetti della stessa "classe" !
- Ad esempio, per definire il *metodo* **getX** per **p1**:

```
p1.getX = function() { return this.x; }
```

Al solito, *this* è necessario per evitare di riferirsi all'environment locale della funzione costruttore.

*E se si vuole definire lo stesso
metodo per più oggetti ... ?*

Metodi per più oggetti

- Un modo per definire lo stesso metodo per più oggetti consiste nell' *assegnare tale metodo ad altri oggetti*.
Ad esempio, per definire il *metodo getX* anche in *p2*:

p2.getX = p1.getX

ATTENZIONE: non si sta *chiamando* il metodo *getX*, lo si sta *referenziando* → non c'è l'operatore di chiamata ()

- **ESEMPIO D'USO**

```
document.write( p1.getX() + "<br/>" )
```

L'operatore di chiamata () è necessario al momento dell'uso (invocazione) del metodo.

Metodi per una "classe" di oggetti

- In assenza del concetto di classe, assicurare che oggetti "dello stesso tipo" abbiano *lo stesso funzionamento* richiede un'opportuna metodologia.
- Un possibile approccio consiste nel **definire tali metodi dentro al costruttore**:

```
Point = function(i,j) {  
    this.x = i;  this.y = j;  
    this.getX = function(){ return this.x }  
    this.getY = function(){ return this.y }  
}
```

- Esiste però un *approccio più efficace*, basato sul concetto di prototipo (si veda oltre)

Metodi: Invocazione

- L'operatore di chiamata `()` è quello che effettivamente *invoca* il metodo.

- Ad esempio, definito il metodo:

```
p1.getX = function() { return this.x; }
```

per invocarlo gli si applica l'operatore di chiamata `()`:

```
document.write( p1.getX() + "<br/>" )
```

- **ATTENZIONE:** se si invoca un *metodo inesistente* si ha *errore a run-time* (metodo non supportato)
- **NB:** se l'interprete Javascript incontra un *errore a run-time*, non esegue le istruzioni successive e *spesso non visualizza alcun messaggio d'errore!*

Simulare proprietà "private"

- Anche se *le proprietà di un oggetto sono pubbliche*, si possono **"simulare" proprietà private**, tramite variabili locali della funzione costruttore. Ad esempio:

```
Rettangolo = function() {  
    var latoX, latoY;  
    this.setX = function(a) { latoX = a }  
    this.setY = function(a) { latoY = a }  
    this.getX = function() { return latoX }  
    this.getY = function() { return latoY }  
}
```

La keyword **this** rende visibili a tutti le quattro proprietà setX, setY, getX e getY, mentre latoX e latoY sono visibili solo *nell'environment locale della funzione costruttore* e dunque, di fatto, "private".

Variabili e metodi "di classe"

- In assenza del concetto di classe, le *variabili di classe* si possono modellare ad esempio come **proprietà dell' "oggetto costruttore"** (che è una funzione - e quindi un oggetto - esso stesso). Ad esempio:

```
p1 = new Point(3,4);  Point.color = "black";
```

ATTENZIONE: la notazione completa Point.color è *necessaria anche* se la definizione compare dentro al costruttore stesso, poiché **color** da sola definirebbe una *variabile locale* della funzione costruttore, *non una proprietà dell'oggetto costruttore*!

- Analogamente si possono modellare come **proprietà dell' "oggetto costruttore"** anche i *metodi di classe*:

```
Point.commonMethod = function(...) { ... }
```

Oggetti Function

Ogni funzione Javascript è un oggetto, costruito sulla base del **costruttore Function**

- implicitamente, tramite il **costrutto function**
 - gli argomenti sono i parametri formali della funzione
 - il corpo (codice) della funzione è racchiuso in un blocco

Esempio: `square = function(x) { return x*x }`

- esplicitamente, tramite il **costruttore Function**
 - **gli argomenti sono tutte stringhe**
 - *i primi N-1 sono i nomi dei parametri della funzione*
 - *l'ultimo è il corpo (codice) della funzione*

Esempio:

`square = new Function("x", "return x*x")`

Oggetti Function

Ogni funzione Javascript è un oggetto, costruito sulla base del **costruttore Function**

- **implicitamente** Costruisce dentro al programma Javascript
 - gli argomenti sono *tutte stringhe*
 - il codice → valutato una sola volta (compilazione)
→ più efficiente, ma non flessibile.

Esempio: `square = function(x) { return x*x }`

- **esplicitamente**, tramite il **costruttore Function**
 - gli argomenti sono *tutte stringhe*
 - i primi
 - l'ultimo

Costruisce a partire da STRINGHE
→ valutato ogni volta
→ meno efficiente, ma *molto flessibile!*

Esempio:

`square = new Function("x", "return x*x")`

Funzioni come dati – Revisione (1/4)

Riconsideriamo la funzione che invocava funzioni:

```
function calc(f, x) { return f(x) }
```

```
calc(Math.sin, .8)    dà 0.7173560908995228
```

```
calc(Math.log, .8)    dà -0.2231435513142097
```

Essa funzionava solo purché l'argomento fosse un oggetto funzione, provocando **errore negli altri casi:**

```
calc(x*x, .8)    dà errore perché  $x*x$  non è un oggetto  
                  funzione del programma
```

```
calc("Math.sin", .8)    dà errore perché una stringa  
                        non è una funzione
```

Ora possiamo gestire anche questi casi, ricorrendo a una costruzione dinamica tramite il costruttore *Function*

Funzioni come dati – Revisione (2/4)

Costruzione dinamica tramite il costruttore *Function* :

- nel caso di funzioni di cui è noto solo il codice:

```
calc(x*x, .8)
```

dà errore

```
calc(new Function("x", "return x*x"), .8)
```

dà 0.64

- nel caso di funzioni di cui è noto solo il nome:

```
calc("Math.sin", , .8)
```

dà errore

```
calc(new Function("z", "return Math.sin(z)"), .8)
```

dà 0.7173560908995228

Funzioni come dati – Revisione (3/4)

Da qui a generalizzare, il passo è breve:

```
var funz = prompt("Scrivere f(x) : ")  
var x = prompt("Calcolare per x = ? ")  
var f = new Function("x", "return " + funz)
```

Ora, **l'utente può digitare nella finestra di dialogo**

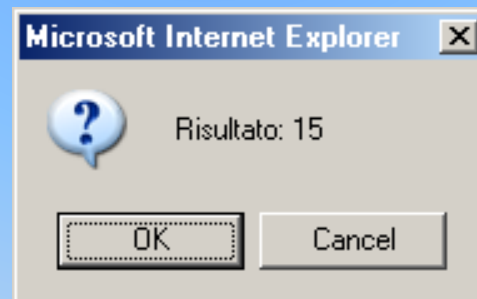
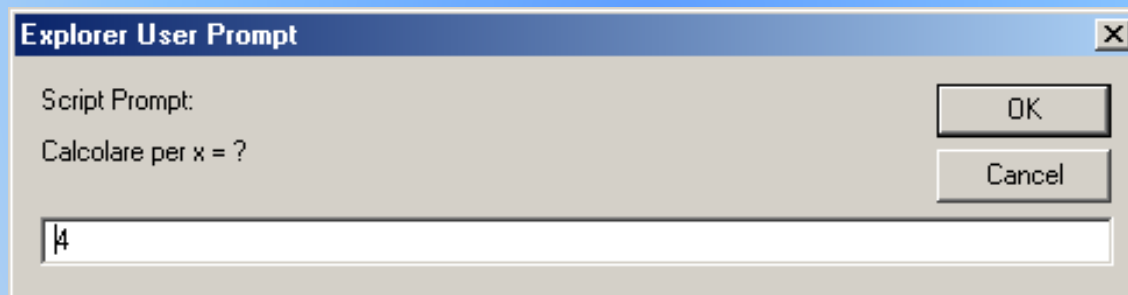
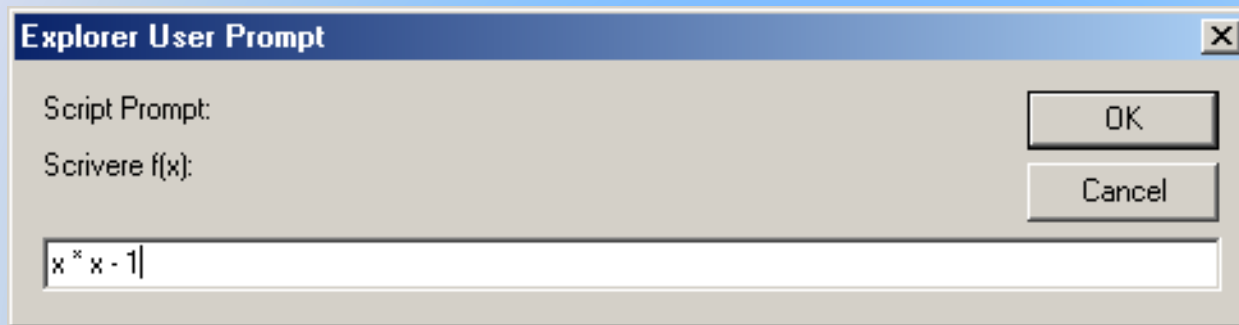
- il testo della funzione desiderata (*attenzione a cosa scrive...*)
- il valore del punto in cui calcolarla

e causarne l'invocazione tramite un *meccanismo riflessivo* !

Il risultato può essere mostrato ad esempio con un'altra finestra di dialogo:

```
confirm("Risultato: " + f(x))
```

Funzioni come dati – Revisione (4/4)



Funzioni come dati: un problema

Per Javascript, i valori immessi da prompt sono *stringhe*
Quindi, *finché è possibile si opera sul dominio stringhe.*

Conseguenza: *x+1 concatena invece di sommare!*

- se l'utente introduce come funzione *x+1...*
- ...per *x=4* ottiene come risultato *41 !!*

Come provvedere?

Possibili soluzioni:

- far scrivere all'utente la funzione in modo "non ambiguo"
con conversione esplicita di tipo: `parseInt(x)+1`
- imporre da programma una conversione esplicita di tipo
dal dominio stringhe al dominio numerico dopo il prompt:

```
var x = parseInt(prompt("Calcolare per x = ? "))
```

Funzioni come dati: un problema

Per Javascript, i valori immessi da prompt sono *stringhe*
Quindi, *finché è possibile si opera sul dominio stringhe.*

Conseguenza: *x+1 concatena invece di sommare!*

- se l'utente introduce come funzione x+1...
- ...per x=4 ottiene come risultato 41 !!

Come provvedere?

Possibili soluzioni:

Ora `typeof (x)` è `number`

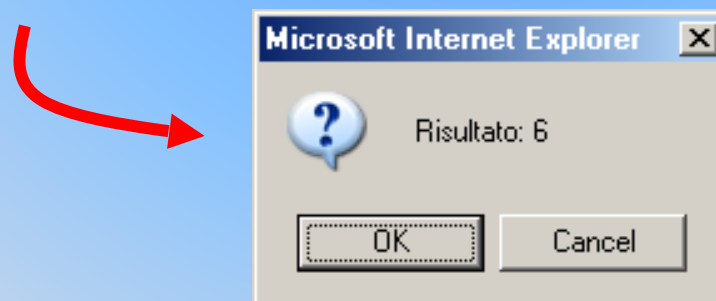
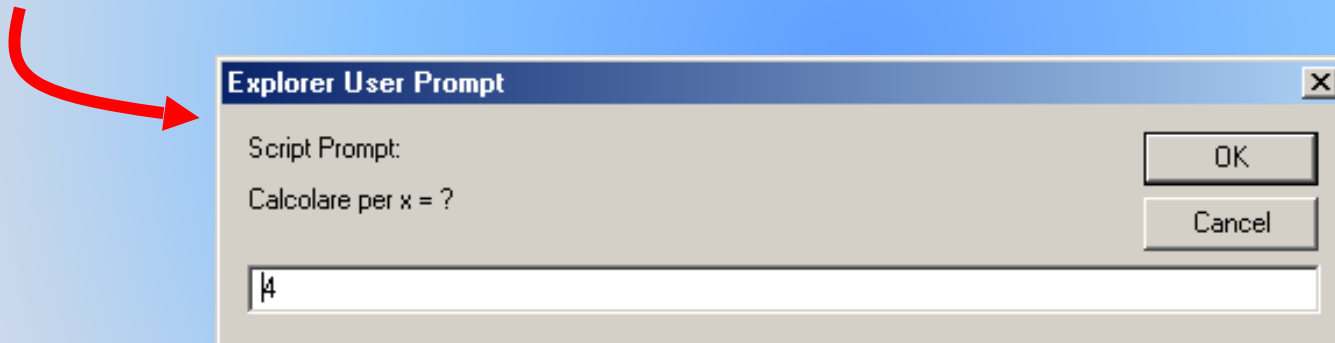
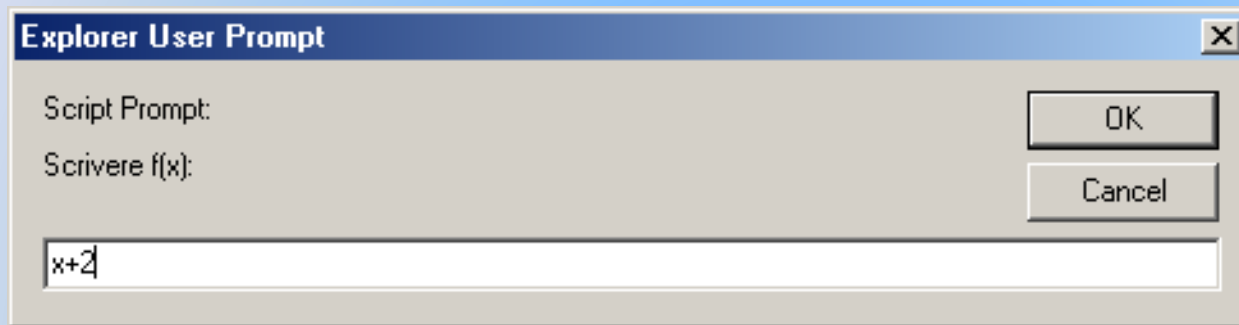
→ si applicano per default
gli operatori del dominio
numerico

funzione in modo "non ambiguo"
di tipo: `parseInt(x)+1`

...ma conversione esplicita di tipo
dal dominio stringhe al dominio numerico dopo il prompt:

```
var x = parseInt(prompt("Calcolare per x = ? "));
```

Funzioni come dati – Output finale



Oggetti Function: proprietà

Proprietà statiche: *(esistono anche mentre non esegue)*

- *length* - numero di **parametri formali** (attesi)

Proprietà dinamiche: *(mentre la funzione è in esecuzione)*

- *arguments* - array contenente i **parametri attuali**
- *arguments.length* - numero dei **parametri attuali**
- *arguments.callee* - la funzione in esecuzione stessa
- *caller* - il chiamante (null se invocata da top level)
- *constructor* - riferimento all'oggetto costruttore
- *prototype* - riferimento all'oggetto prototipo

Oggetti Function: metodi

Metodi invocabili su una funzione

- *toString* - una stringa fissa (ma si può cambiare...)
- *valueOf* - *ritorna la funzione stessa*
- *call* e *apply* - chiamano questa funzione sull'oggetto passato come primo argomento, fornendole i parametri specificati *il cui formato differenzia call da apply*

Esempi

- `funz.apply(ogg, arrayDiParametri )`
equivale concettualmente a `ogg.funz(parametri)`
- `funz.call(ogg, arg1, arg2, ... )`
equivale concettualmente a `ogg.funz(arg1, arg2, ..)`

call & apply: esempio 1

Definizione dell'oggetto funzione:

```
test = function(x, y, z) { return x + y + z }
```

Invocazione nel contesto corrente di `test(3,4,5)`:

```
test.apply(obj, [3,4,5] )
```

array di costanti

```
test.call(obj, 8, 1, -2 )
```

I parametri sono opzionali: se non ne esistono, `apply` e `call` assumono ovviamente la medesima forma.

NOTA:

in questo esempio, l'oggetto destinatario del messaggio (`obj`) è irrilevante perché la funzione `test` invocata non fa alcun riferimento a `this` nel suo corpo.

call & apply: esempio 2

Questa funzione invece fa riferimento a this:

```
prova = function(v) { return v + this.x }
```

Quindi, l'oggetto destinatario del messaggio diventa rilevante perché **determina l'environment di valutazione di x**

1° caso

```
x = 88  
prova.call(this, 3)
```

Restituisce 3 + 88 = 91

2° caso

```
x = 88  
function Obj(u) {  
  this.x = u  
}  
obj = new Obj(-4)  
prova.call(obj, 3)
```

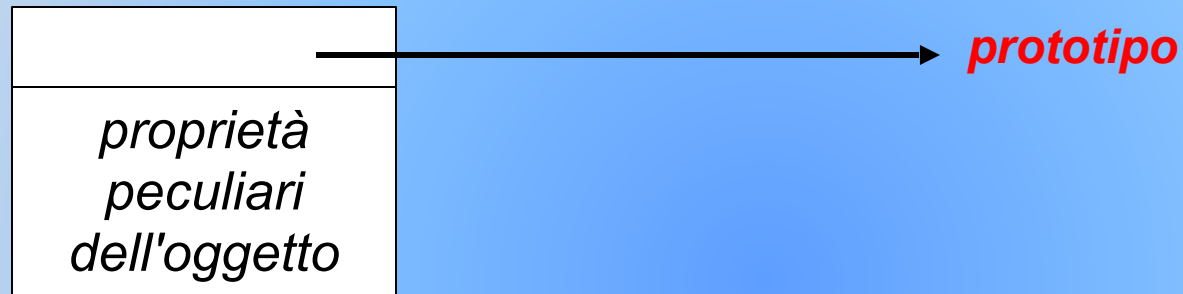
Restituisce 3 + -4 = -1

Prototipi

- In Javascript, **ogni OGGETTO ha sempre un prototipo** che ne specifica le ***proprietà di base***:
 - il prototipo è esso stesso *un oggetto*
 - se P è l'oggetto prototipo dell'oggetto X, tutte le proprietà dell'oggetto prototipo P appaiono anche come proprietà di X
 - sempre che l'oggetto X non le ridefinisca
- Ogni COSTRUTTORE ha inoltre sempre un prototipo di costruzione espresso dalla proprietà prototype
 - serve a definire le *proprietà degli oggetti da lui costruiti (cioè degli oggetti della sua "classe"....!)*
 - per default, coincide con l'altro
 - ma, mentre l'altro è fisso, questo può essere cambiato
 - ciò consente di ottenere *ereditarietà "prototype-based"*

Prototipi: Architettura

Oggetto



Costruttore



Prototipi di base

Non esistendo classi, Javascript fornisce una serie di **costruttori predefiniti** il cui **prototype** fa da prototipo per tutti gli oggetti di quel certo "tipo" :

- il costruttore **Function** ha come **prototype** il proto-tipo di tutte le **funzioni**
- il costruttore **Array** ha come **prototype** il prototipo di tutti gli **array**
- il costruttore **Object** ha come **prototype** il prototipo di tutti gli **oggetti definiti dall'utente** costruiti con **new**

Altri **costruttori predefiniti** sono **Number**, **Boolean**, **Date**, **RegExp** (per le espressioni regolari)

Il prototipo è memorizzato internamente nella proprietà di sistema (non visibile né enumerabile) `__proto__`

Tassonomia di Prototipi

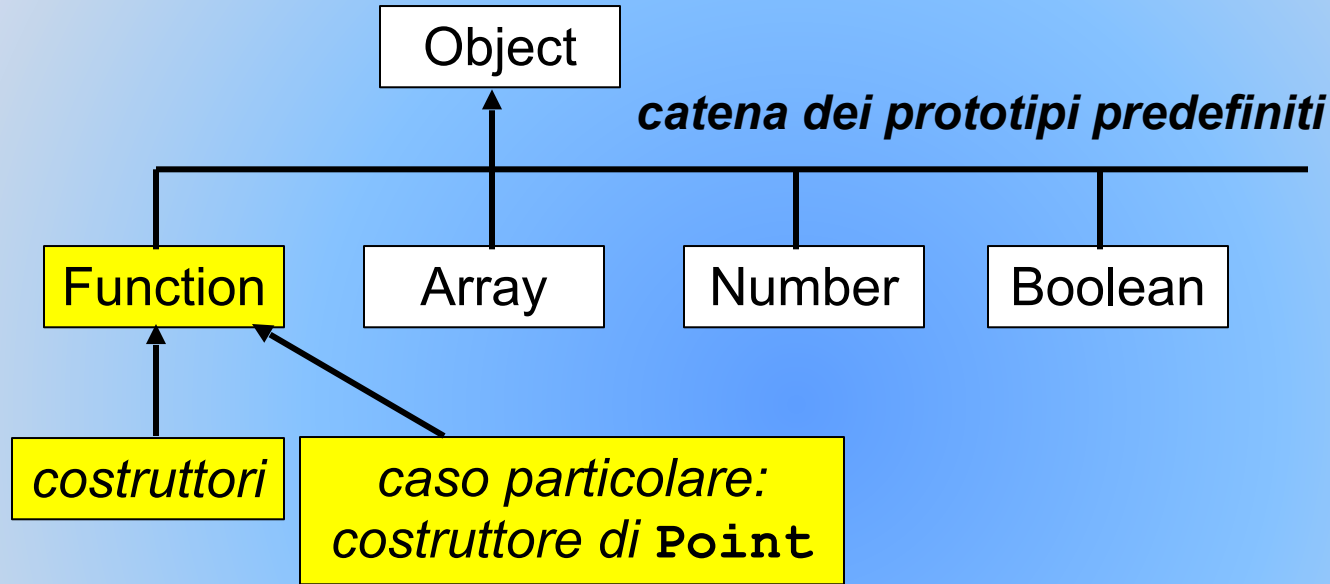
- Poiché i **costruttori sono essi stessi oggetti**, **hanno a loro volta un prototipo!**
- Si crea perciò una **"tassonomia di prototipi"**, alla cui radice vi è il prototipo di **Object**

Il **prototipo di Object** definisce le proprietà:

- **constructor** – la funzione che ha costruito l'oggetto
- **toString()** – il metodo per "stampare" l'oggetto
- **valueOf()** – ritorna il tipo primitivo sottostante

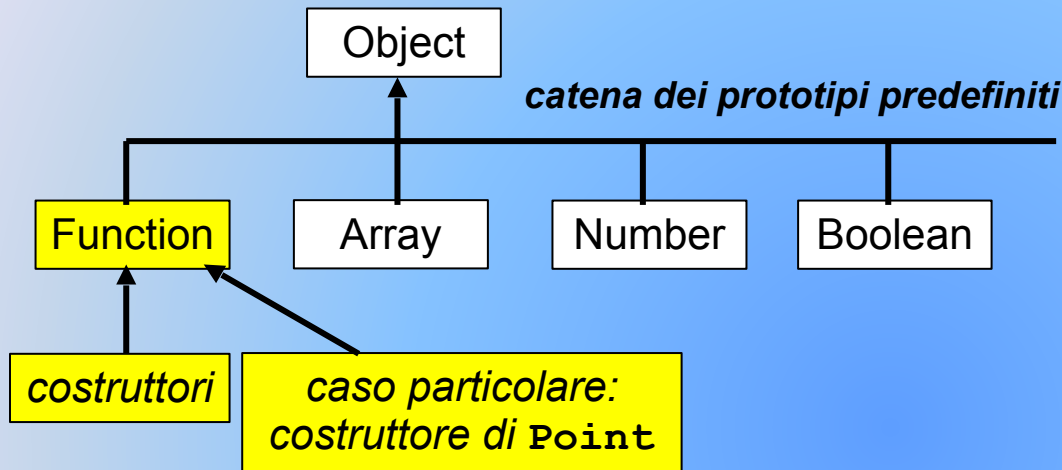
Dunque, **ogni oggetto (funzioni e costruttori compresi!)** gode di queste proprietà.

Tassonomia di Prototipi



- Al prototipo di **Function** si agganciano tutte le funzioni, e in particolare tutti i costruttori
- Tale prototipo definisce proprietà come `arguments`, tipiche di tutte le funzioni (inclusi i costruttori) ed "eredita" le proprietà del prototipo di `Object` (es. `constructor`)

Esperimenti



Il metodo predefinito `isPrototypeOf()` verifica se un oggetto ne ha un altro nella sua catena di prototipi (SOLO Microsoft JScript)

Test:

`Object.prototype.isPrototypeOf(Function)` *true*

`Object.prototype.isPrototypeOf(Array)` *true*

`Function.prototype.isPrototypeOf(Point)` *true*

`Object.prototype.isPrototypeOf(Point)` *true*

Point è un costruttore,
quindi anche una funzione

... ma è anche un oggetto

La proprietà prototype

- Il prototipo di costruzione esiste *solo per i costruttori*
 - esiste comunque: se non ne è stato definito uno specifico, coincide con il prototipo di base
- Il **prototipo di costruzione** definisce le *proprietà di tutti gli oggetti costruiti da quel costruttore*: in pratica, definisce le proprietà di una "classe" di oggetti.

Per definire un prototipo di costruzione specifico occorre:

- definire un oggetto che faccia da prototipo con le proprietà desiderate
- assegnarlo alla proprietà prototype del costruttore.

NB: la proprietà *prototype* si può cambiare dinamicamente, ma la modifica riguarda *solo gli oggetti creati da quel momento in poi*.

Esempio

Supponendo di avere:

```
Point = function(i,j) {  
    this.x = i;  
    this.y = j;  
}
```

Attenzione: la forma

```
function Point()
```

non rende globale l'identificatore Point, creando problemi se il prototipo è aggiunto da un altro environment da cui Point non si veda

si vuole associarvi un prototipo che definisca le funzioni `getX` e `getY`.

1) Definire il costruttore per l'oggetto che deve fungere da prototipo:

```
GetXY = function() {  
    this.getX = function() { return this.x }  
    this.getY = function() { return this.y }  
}
```

2) Crearlo ed assegnarlo alla proprietà `prototype` del costruttore `Point`:

```
myProto = new GetXY(); Point.prototype = myProto
```

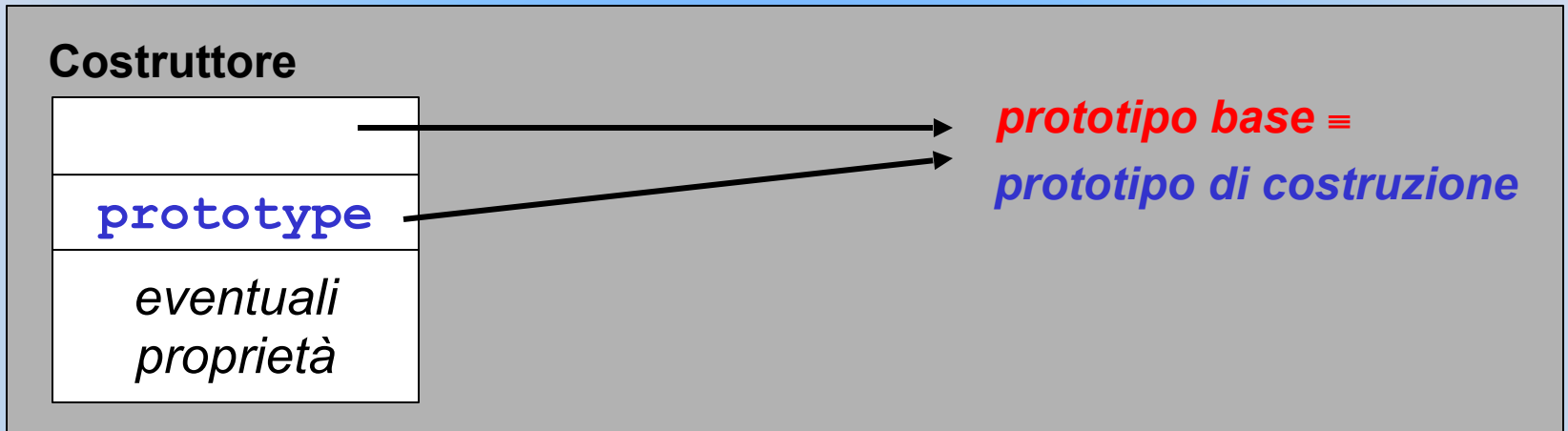
3) Sui nuovi oggetti `Point` si possono invocare `getX` e `getY`:

```
p4 = new Point(7,8); document.write(p4.getX())
```

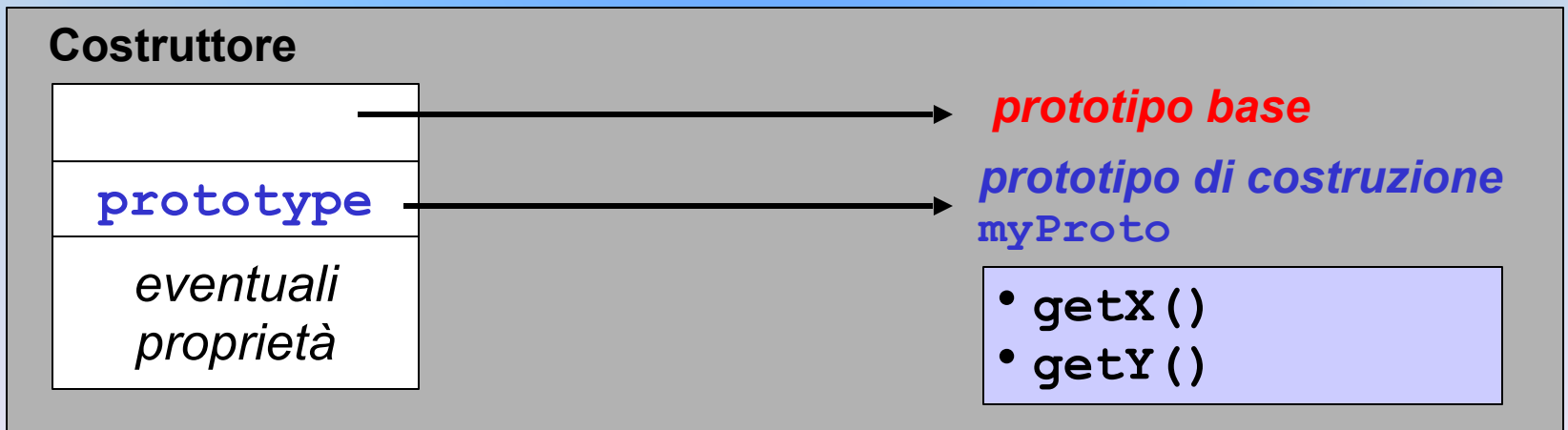
NB: `getX` e `getY` non si possono invocare su oggetti *preesistenti*!

Architettura

PRIMA

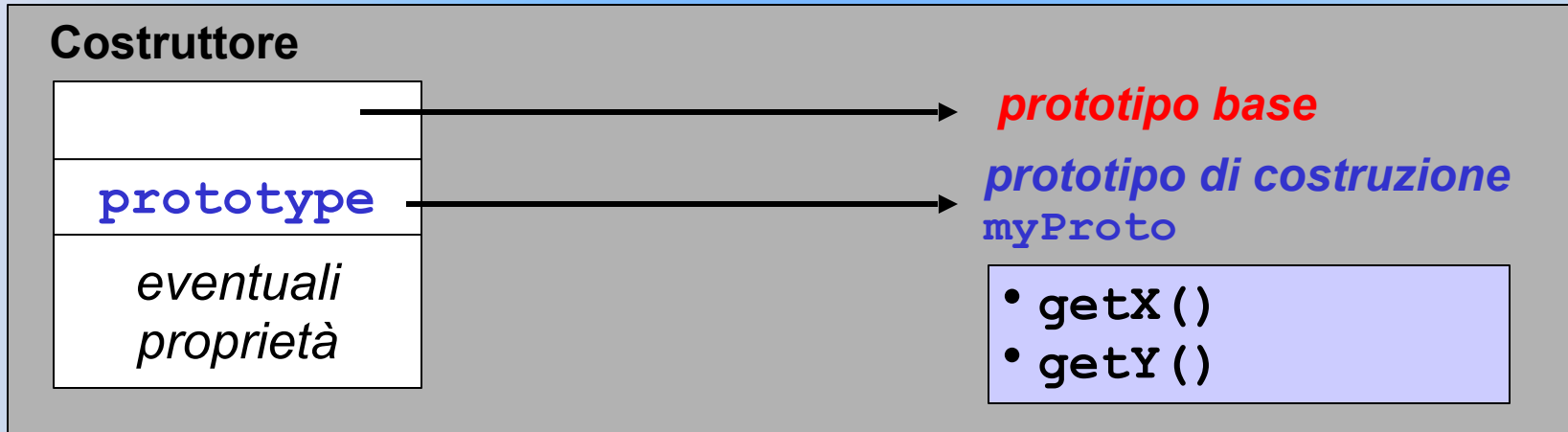


DOPO

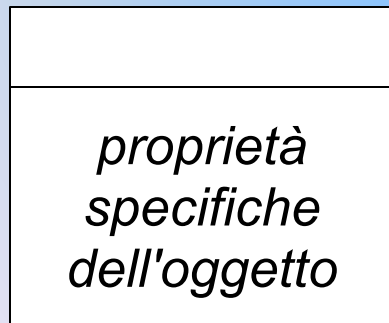


Ricerca delle proprietà

DOPO

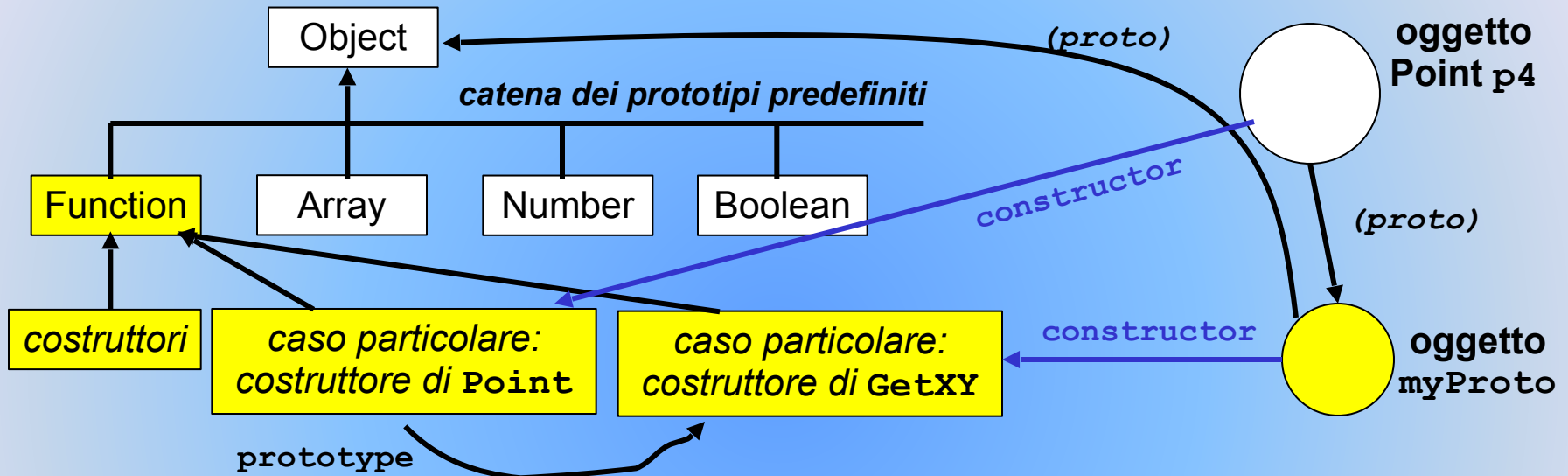


Oggetto



ordine di ricerca delle proprietà

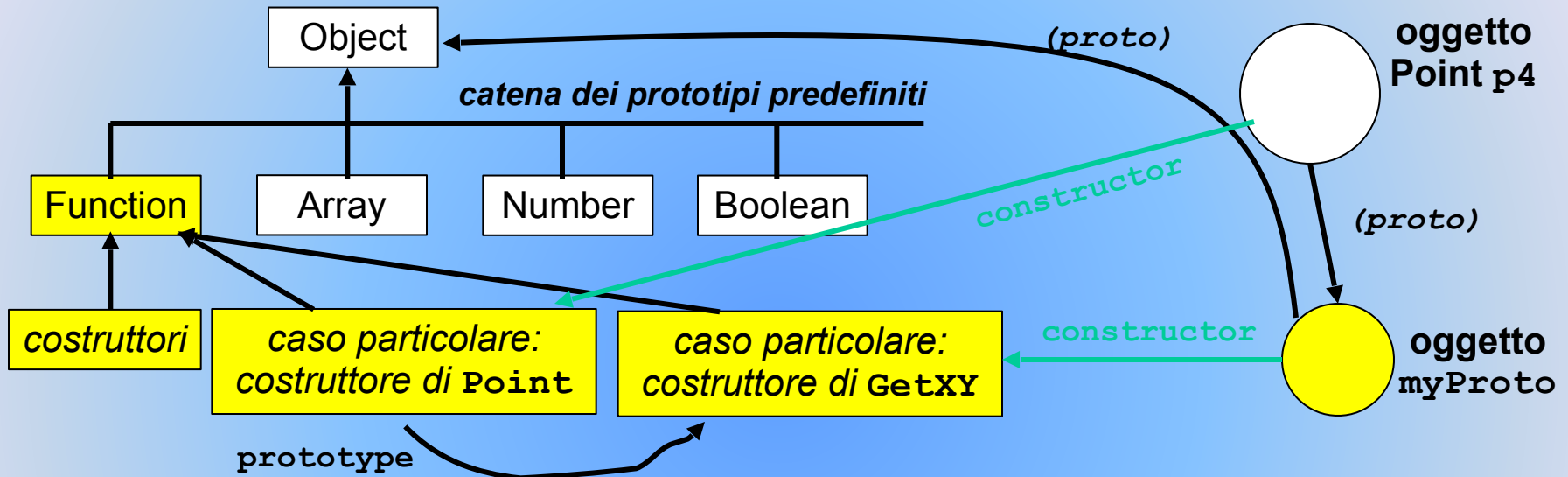
Nuovi Esperimenti (1)



Test n° 1:

<code>myProto.isPrototypeOf(p4)</code>	<code>true</code>
<code>GetXY.prototype.isPrototypeOf(p4)</code>	<code>true</code>
<code>Point.prototype.isPrototypeOf(p4)</code>	<code>true</code>
<code>Object.prototype.isPrototypeOf(p4)</code>	<code>true</code>
<code>Function.prototype.isPrototypeOf(p4)</code>	<code>false</code>

Nuovi Esperimenti (2)



Test n° 2:

<code>Point.prototype.isPrototypeOf(myProto)</code>	<code>true</code>
<code>Object.prototype.isPrototypeOf(myProto)</code>	<code>true</code>
<code>Function.prototype.isPrototypeOf(myProto)</code>	<code>false</code>
<code>Point.prototype.isPrototypeOf(GetXY)</code>	<code>false</code>
<code>Object.prototype.isPrototypeOf(GetXY)</code>	<code>true</code>
<code>Function.prototype.isPrototypeOf(GetXY)</code>	<code>true</code>

Prototipi di costruzione: un approccio alternativo

- Anziché associare a un costruttore un prototipo *"totalmente nuovo"*, si può scegliere di **aggiungere nuove proprietà al prototipo di costruzione esistente**
- Non più `Point.prototype = myProto`
- Ma `Point.prototype.getX = function() { ... }`
`Point.prototype.getY = function() { ... }`

I due approcci non sono equivalenti !

- Una modifica al prototipo esistente *influisce anche sugli oggetti già creati*
- Invece, come si è visto, la "messa in pista" di un nuovo prototipo riguarda *solo gli oggetti creati da quel momento in avanti*.

Esempio

Supponendo di avere:

```
Point = function(i,j) {  
    this.x = i;  
    this.y = j;  
}
```

si vuole **modificare il prototipo** in modo da includere `getX` e `getY`.

1) Creazione di un primo oggetto: `p1 = new Point(1,2)`

2) Verifica: `p1.getX()` e `p1.getY()` non sono supportati

3) **Modifica diretta del prototipo esistente:**

```
Point.prototype.getX = function(){ return this.x }  
Point.prototype.getY = function(){ return this.y }
```

4) Verifica: `p1.getX()` e `p1.getY()` **ora funzionano!!**

5) Ovviamente funzionano anche sugli oggetti creati successivamente.

Ereditarietà tramite prototipi

- Le catene di prototipi sono il modo offerto da JavaScript per supportare una sorta di *"ereditarietà"*
- È una *"ereditarietà"* tra oggetti, non tra classi come nei linguaggi "object-oriented"
- Si parla di *"prototype-based inheritance"*

Il meccanismo:

- Quando si crea un nuovo oggetto con `new`, ad esso viene automaticamente associato dal sistema il *prototipo di costruzione* previsto per quel costruttore
- Ciò *vale anche per i costruttori stessi*, che per default hanno come prototipo `Function.prototype`

Esprimere l'ereditarietà

- Si supponga di aver già definito in precedenza il costruttore [di una classe] base, ad esempio **Persona**
- Per esprimere l'idea di una "sottoclasse" che "erediti" da essa, ad esempio **Studiante**, occorre:

- ① associare esplicitamente a **Studiante.prototype** un nuovo oggetto **Persona**

MA questo passo causa anche l'effetto indesiderato di alterare il costruttore predefinito di **Studiante**, perché **constructor** è una sotto-proprietà di **prototype** → occorre rimettere le cose a posto:

- ② cambiare esplicitamente la proprietà **constructor** di **Studiante.prototype** (che ora punterebbe a **Persona**) in modo che punti di nuovo al costruttore **Studiante**

Esempio (1/2)

Costruttore base

```
Persona = function(n,a) {  
    this.nome = n; this.anno = a;  
    this.toString = function(){  
        return this.nome + " è nata nel " + this.anno }  
}
```

Costruttore "derivato"

```
Studente = function(n,a,m) {  
    this.nome = n; this.anno = a; this.matr = m;  
    this.toString = function(){  
        return this.nome + " è nata nel " + this.anno  
        + " e ha matricola " + this.matr }  
}
```

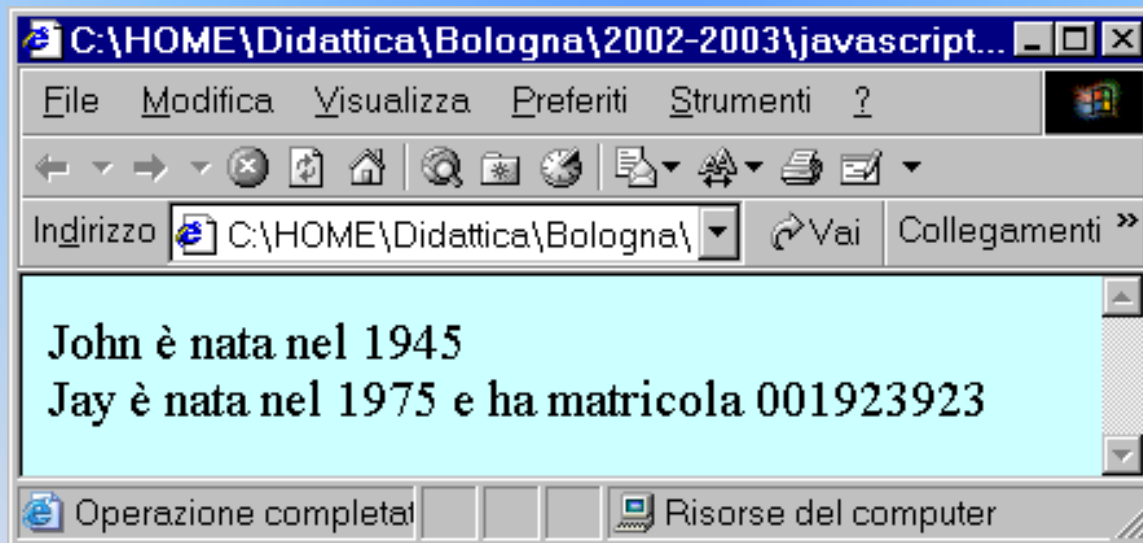
Impostazione catena dei prototipi:

```
Studente.prototype = new Persona()  
Studente.prototype.constructor = Studente
```

Esempio (2/2)

Test

```
function test() {  
    var p = new Persona("John", 1945)  
    var s = new Studente("Jay", 1975, "001923923")  
    document.write(p)  
    document.write(s)  
}
```



Ereditarietà: una alternativa (1/2)

Un approccio alternativo al precedente, *che non tocca i prototipi*, consiste nel *riutilizzare tramite call la funzione costruttore "base"*, in modo analogo a quanto si fa in Java tramite *super*

```
Rettangolo = function() {  
    var latoX, latoY;  
    this.setX = function(a) { latoX = a }  
    this.setY = function(a) { latoY = a }  
    this.getX = function() { return latoX }  
    this.getY = function() { return latoY }  
}
```

```
Quadrato = function() {  
    Rettangolo.call(this) ;  
}
```

Simula il "rimballo" del costruttore di default di **Quadrato** verso quello di **Rettangolo**, che in Java si esprimerebbe con *super ()*

Ereditarietà: una alternativa (2/2)

Test:

```
q1 = new Quadrato() ; q1.setX(4) ; q2.setY(5) ;
```

funziona, quindi Quadrato ha "ereditato" da Rettangolo

```
q2 = new Quadrato() ; q1.setX(9) ; q2.setY(8) ;
```

funziona, quindi Quadrato ha "ereditato" da Rettangolo

```
res1 = r.getX() + ", " + r.getY()      dà "4, 5"
```

```
res2 = s.getX() + ", " + s.getY()      dà "9, 8"
```

Conclusione:

l'ereditarietà funziona correttamente anche per quanto riguarda le variabili, come dimostrato dal fatto che q1 e q2 mantengono ciascuno il proprio stato.

Ereditarietà: "super" nei costruttori

Costruttore base

```
Persona = function(n,a) {  
  this.nome = n; this.anno = a;  
  this.toString = function() {  
    return this.nome + " è nata nel " + this.anno }  
}
```

Costruttore "derivato"

```
Studiante = function(n,a,m) {  
  Persona.call(this,n,a);  
  this.matr = m;  
  this.toString = function() {  
    return this.nome + " è nata nel " + this.anno  
    + " e ha matricola " + this.matr }  
}
```

Simula il "rimballo" del costruttore di Studiante verso quello di Persona, che in Java si esprimerebbe con `super(n,a)`

Ereditarietà: "super" nei metodi

- Per richiamare metodi definiti nel costruttore base, emulando il comportamento del costrutto *super* di Java anche in questo, **nel caso della manipolazione esplicita della catena dei prototipi** si può agire così:

```
Studente = function(n,a,m) {  
  Persona.call(this,n,a); this.matr = m;  
  this.toString = function() {  
    return Studente.prototype.toString.call(this)  
      + " e ha matricola " + this.matr }  
}
```

Studente.prototype è, nel nostro caso, **un oggetto Persona**

quindi **call** invoca la **toString** di **Persona**

Alternativa: "super" nei metodi

- Invece, **nel caso non si vogliano toccare i prototipi**, occorre usare esplicitamente un oggetto del tipo che funge da prototipo, appoggiandosi ad esso per invocare il metodo desiderato:

```
Studiante = function(n,a,m) {  
  Persona.call(this,n,a); this.matr = m;  
  this.toString = function() {  
    return p.toString.call(this) +  
      " e ha matricola " + this.matr }  
}
```

p è un oggetto **Persona** (che deve esistere quando questa funzione viene chiamata), quindi **call** invoca la **toString** di **Persona**

Ereditarietà: esperimenti (1/2)

```
Persona = function(n,a){  
  this.nome = n; this.anno = a; this.toString = function(){ ...}  
}  
Studente = function(n,a,m){  
  this.nome = n; this.anno = a; this.matr = m; this.toString = ...  
}
```

Catena dei prototipi impostata esplicitamente:

```
protoPersona = new Persona();  
Studente.prototype = protoPersona  
Studente.prototype.constructor = Studente
```

ALCUNI ESPERIMENTI:

<code>p.isPrototypeOf(s)</code>	<i>false</i>
<code>Persona.isPrototypeOf(s)</code>	<i>false</i>
<code>Object.isPrototypeOf(s)</code>	<i>false</i>
<code>Object.prototype.isPrototypeOf(s)</code>	<i>true</i>
<code>Persona.isPrototypeOf(Studente)</code>	<i>false</i>
<code>protoPersona==Studente.prototype</code>	<i>true</i>
<code>protoPersona.isPrototypeOf(Studente)</code>	<i>false</i>
<code>protoPersona.isPrototypeOf(Studente.prototype)</code>	<i>true</i>
<code>protoPersona.isPrototypeOf(s)</code>	<i>true</i>

Ereditarietà: esperimenti (2/2)

```
Persona = function(n,a){  
  this.nome = n; this.anno = a; this.toString = function(){ ...}  
}  
Studiante = function(n,a,m){  
  this.nome = n; this.anno = a; this.matr = m; this.toString = ...  
}
```

NESSUNA CATENA DI PROTOTIPI IMPOSTATA ESPLICITAMENTE!

GLI STESSI ESPERIMENTI:

<code>p.isPrototypeOf(s)</code>	<i>false</i>
<code>Persona.isPrototypeOf(s)</code>	<i>false</i>
<code>Object.isPrototypeOf(s)</code>	<i>false</i>
<code>Object.prototype.isPrototypeOf(s)</code>	<i>true</i>
<code>Persona.isPrototypeOf(Studiante)</code>	<i>false</i>
<code>protoPersona==Studiante.prototype</code>	<i>false</i>
<code>protoPersona.isPrototypeOf(Studiante)</code>	<i>false</i>
<code>protoPersona.isPrototypeOf(Studiante.prototype)</code>	<i>false</i>
<code>protoPersona.isPrototypeOf(s)</code>	<i>false</i>

Array (1/2)

- Un **array** Javascript è una entità a metà strada fra l'array e il Vector di Java
- Come in Java, gli elementi sono numerati da 0; l'attributo **length** dà la lunghezza dinamica dell'array
- È costruito sulla base del costruttore **Array** i cui argomenti sono **il contenuto iniziale dell'array**:
`colori = new Array("rosso", "verde", "blu")`
- I singoli elementi sono referenziati con l'usuale notazione *a parentesi quadre*: ad esempio, `colori[2]`
- Le celle di un array Javascript non hanno il vincolo di omogeneità in tipo: ogni cella può contenere indistintamente numeri, stringhe, oggetti, altri array, ...

Array (2/2)

- E' anche possibile definire un **array vuoto** e **aggiungere elementi successivamente** mediante **assegnamenti**:

```
colori = new Array()  
colori[0] = "rosso"  
...
```

- È quindi possibile **aggiungere elementi dinamicamente**, man mano che ne sorge la necessità (*come in un Vector*)

```
colori = new Array("rosso", "verde", "blu")
```

```
...
```

```
colori[3] = "giallo";
```

```
for (i=0; i<colori.length; i++)  
    document.write(colori[i] + " ")
```

array di dimensione 3

aggiunta del 4°
elemento

Array - Costruzione Alternativa

- A partire da Javascript 1.2, anche per gli array esiste un ***modo alternativo di costruzione***: basta elencare la ***sequenza, racchiusa fra parentesi quadre, di valori iniziali separati da virgole***:

```
vett = [ 120, -0.5, "paperino" ]
```

Ma la cosa davvero interessante è
la visione degli *oggetti* come *array* ! → → →

Oggetti come Array (1/2)

- In Javascript, ogni oggetto è definito dall'insieme delle sue proprietà, che è *dinamicamente estendibile*
- Internamente, ciò è ottenuto *rappresentando ogni oggetto tramite un array.*
- Tale mapping *oggetti* → *array* rende possibile una notazione "array-like" per accedere agli oggetti usando il nome della proprietà come selettore
- se *p* è un oggetto e *s* è una stringa contenente il nome di una sua proprietà *x*,
- la notazione *p[s]* dà accesso alla proprietà di nome *x*, *in modo analogo alla "dot notation" p.x*

***Ma se è analogo alla dot notation,
dov'è il vantaggio??***

Oggetti come Array (2/2)

Il vantaggio è che

- usare la *"dot notation"* $p.x$ implica che *il nome della proprietà x sia noto a priori*, quando il programma viene scritto.
- al contrario, la notazione $p[s]$ permette di *accedere anche a una proprietà x il cui nome NON sia noto a priori*, in quanto nel programma compare solo la variabile stringa s che conterrà in futuro il nome della proprietà desiderata.

Introspezione

- La possibilità di *aggiungere dinamicamente proprietà* a un oggetto pone il problema di *scoprire quali proprietà esso abbia*, ossia di procedere alla sua introspezione
- A questo fine è previsto il costrutto:

for (variabile in oggetto) ...

che itera per ogni proprietà visibile dell'oggetto

Non sono visibili le proprietà con l'attributo DontEnum settato

- Ad esempio, per elencare i nomi di tutte le proprietà:

```
function show(ogg) {  
  for (var pName in ogg)  
    document.write("proprietà: " + pName + "<BR>")  
}
```

Da Introspezione a Intercessione

- Con il costrutto introspettivo `for(..in..)` è possibile scoprire le proprietà (visibili) di un oggetto:

```
function show(ogg) {  
    for (var pName in ogg)  
        document.write("proprietà: " + pName + "<BR>")  
}
```

- Per accedere a tali proprietà occorre però ottenere un riferimento ad esse a partire dalla stringa che ne contiene il nome: a ciò provvede la notazione *ogg[pName]*

```
function show(ogg) {  
    for (var pName in ogg) {  
        document.write("proprietà: " + pName +  
            ", tipo " + typeof(ogg[pName]) + "<BR>")  
    }  
}
```

Notazione array-like: Esempio

- Ad esempio, l'invocazione `show(p1) ; show(p2)` sui due oggetti `Point` definiti in precedenza produce

proprietà: `x`, tipo *number*
proprietà: `y`, tipo *number*
proprietà: `z`, tipo *number*
proprietà: `getX`, tipo *function*
proprietà: `getY`, tipo *function*

} p1

proprietà: `x`, tipo *number*
proprietà: `y`, tipo *number*
proprietà: `getX`, tipo *function*
proprietà: `getY`, tipo *function*

} p2

L'Oggetto Globale

- PROBLEMA: come può Javascript distinguere fra *metodi di oggetti* e *funzioni "globali"* (come le precedenti) ?
- Non distingue: le *funzioni "globali"* non sono altro che *metodi di un "oggetto globale"* definito dal sistema.

L' *oggetto "globale"* ha

- come *metodi*, le *funzioni non attribuite a uno specifico oggetto nonché quelle predefinite*
- come *dati*, le *variabili globali*
- come *funzioni*, le *funzioni predefinite*

Funzioni globali predefinite

Funzioni globali:

- eval valuta il programma Javascript passato come stringa (*riflessione, intercessione*)
- **escape** converte una stringa nel formato portabile: i caratteri non consentiti sono sostituiti da "sequenze di escape" (es. %20 per ' ')
- **unescape** riporta una stringa da formato portabile a formato originale
- **isFinite**, **isNaN**, **parseFloat**, **parseInt**
- ...

[Costruttori di] Oggetti predefiniti

Oggetti di uso generale:

- **Array, Boolean, Function, Number, Object, String**
- l'oggetto **Math** contiene la libreria matematica: costanti (E, PI, LN10, LN2, LOG10E, LOG2E, SQRT1_2, SQRT2) e funzioni di ogni tipo
Non va istanziato ma usato come componente "statico".
- l'oggetto **Date** definisce i concetti per esprimere date e orari e lavorare su essi. *Va istanziato* nei modi opportuni.
- l'oggetto **RegExp** fornisce il supporto per le *espressioni regolari*.

Oggetti di uso grafico:

- **Anchor, Applet, Area, Button, Checkbox, Document, Event, FileUpload, Form, Frame, Hidden, History, Image, Layer, Link, Location, Navigator, Option, Password, Radio, Reset, Screen, Select, Submit, Text, Textarea, Window**

Date: costruzione

Costruttori:

- `Date()`, `Date(millisecondi)`, `Date(stringa)`,
`Date(anno, mese, giorno[, hh, mm, ss, msec])`

Note:

- `Date()`: viene creato un oggetto corrispondente alla data e all'ora correnti, come risultano sul sistema in uso
- `Date(millisecondi)`: i millisecondi sono calcolati dalle ore 00:00:00 del 1° gennaio 1970 usando il giorno standard UTC di 86.400.000 ms
- range: da -100.000.000 a +100.000.000 giorni rispetto all' 1/1/1970
- sono supportati sia UTC sia GMT
- `Date(string)`: `string` è nel formato riconosciuto da `Date.parse`
- `Date(anno, mese, giorno,...)`: anno, mese e giorno *devono* essere forniti, gli altri sono opzionali (quelli non forniti sono posti a zero).

Date: metodi & esempi

Metodi

- `getDay`: restituisce il giorno della settimana, da 0 (dom) a 6 (sab)
- `getDate`: restituisce il numero del giorno, da 1 a 31
- `getMonth`: restituisce il mese, da 0 (gennaio) a 11 (dicembre)
- `getFullYear`: restituisce l'anno (su quattro cifre)
- `getHours`: restituisce l'ora, da 0 a 23
- `getMinutes`: restituisce l'ora, da 0 a 59
- `getSeconds`: restituisce l'ora, da 0 a 59
- ...

Esempio:

```
d = new Date() ; millennium = new Date(3000, 00, 01)
st = new String((millennium-d)/86400000)
days = st.substring(0, st.indexOf(".")) // parte intera
document.write("Mancano " + days + " giorni al 3000")
```

Output:

Mancano 364358 giorni al 3000.

L'Oggetto globale: chi è

- L' **oggetto "globale"** è UNICO e viene sempre creato dall'interprete prima di eseguire alcunché
- Però **non esiste un identificatore "global": in ogni situazione c'è un dato oggetto usato come globale**
- **In un browser Web, l'oggetto globale solitamente coincide con l'oggetto window**
*Ma non è sempre così: a lato server, per esempio, sarà probabilmente l'oggetto **response** a svolgere quel ruolo!*
- Quindi, in un browser, per scoprire tutte le proprietà dell'oggetto globale, basta invocare **show(window)**
- **DUBBIO: non sapere chi sia a coprire il ruolo di oggetto globale può essere un problema? Dipende...**

L'Oggetto Globale: cautele

- Variabili e funzioni non assegnate a uno specifico oggetto sono **assegnate all' oggetto "globale"...**
- **... a meno che tali definizioni compaiano dentro lo scope di una funzione**, nel qual caso sono **locali**.

E allora? Che problemi ci sono?

- nessuno, se si usano semplicemente le proprietà globali, senza far "emergere" l'oggetto sottostante (*cioè nella maggioranza dei casi...*)
- parecchi, se si usa **eval** o un'altra funzione riflessiva, perché **eval("var f")** è diverso da **var f**

Infatti, la prima definizione avviene in uno scope **che non è quello globale!** Test: **f==window.f** nei due casi.

Oggetto Globale e Funzioni come dati (1/5)

È noto che Javascript permette di definire *variabili che si riferiscono a funzioni*:

```
var f = function (z) { return z*z; }
```

e di *passare funzioni come parametro* ad altre funzioni:

```
function calc(f, x) {return f(x); }
```

Tuttavia:

- la variabile `f` deve fare riferimento a un **oggetto del programma costruito dal costrutto *function***
- non può essere una stringa contenente il nome della funzione che si vuole eseguire!

`calc("Math.sin", .8)` dà errore

Oggetto Globale e Funzioni come dati (2/5)

Oltre all'approccio basato sul costruttore `Function`,

si può sfruttare l'oggetto globale per **ottenere un riferimento all'oggetto funzione corrispondente a un dato nome di funzione** purché la funzione richiesta sia già definita nel sistema.

Il punto chiave

- in Javascript, se `p` è un riferimento a un oggetto, e `s` è il nome di una sua proprietà `x`,
- la notazione "array-like" `p[s]` fornisce un **riferimento all'oggetto (proprietà) `x`**

Oggetto Globale e Funzioni come dati (3/5)

Ad esempio, la notazione:

```
var math = Math;  var nome = math["sin"]
```

pone nella variabile **nome** un referimento all'oggetto
funzione **Math.sin**

(Nota: l'assegnamento `math = Math` è necessario perché la notazione array-like è ammessa solo su variabili, e `Math` non lo è)

A seguito di ciò, definita la funzione:

```
function calc(f,x) { return f(x) }
```

è ora possibile effettuare l'invocazione:

```
calc(nome, .8)      che dà  0.7173560908995228
```

perché il nome "sin" viene traslato in un riferimento all'oggetto `Math.sin`, utilizzabile per la chiamata.

Oggetto Globale e Funzioni come dati (4/5)

Da qui a generalizzare, il passo è breve:

```
var math = Math;  
var funz = prompt("Nome funzione?")  
var f = math[funz]
```

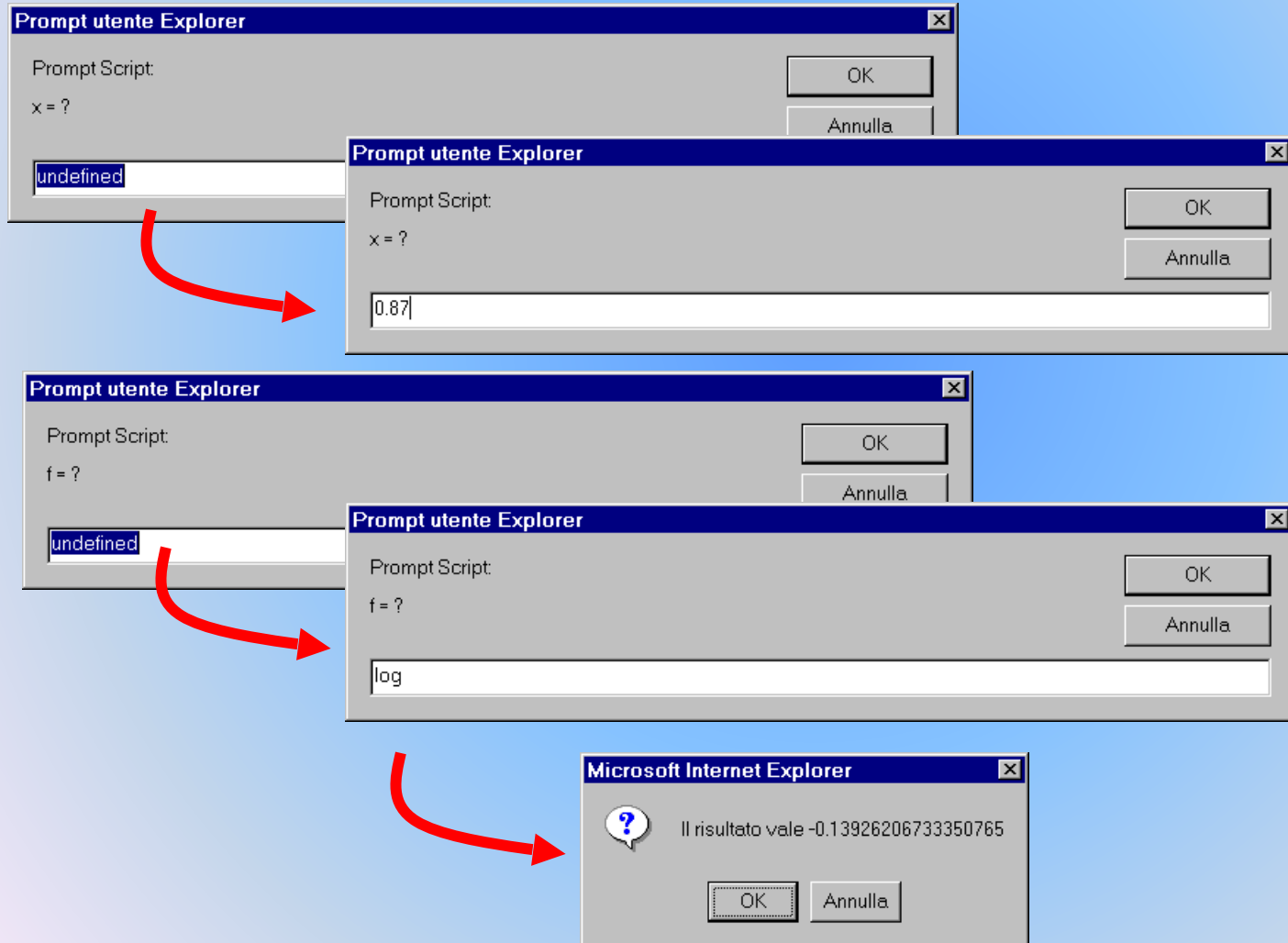
Ora, *l'utente può digitare nella finestra di dialogo il nome della funzione desiderata*, provocandone (la ricerca e) l'invocazione, attraverso un *meccanismo riflessivo*.

Il risultato può essere mostrato con un'altra finestra:

```
confirm("Risultato: " + calc(f,x))
```

Attenzione, però: così si *cercano le funzioni solo entro l'oggetto Math*, a differenza dell'approccio basato su `Function`.

Oggetto Globale e Funzioni come dati (5/5)

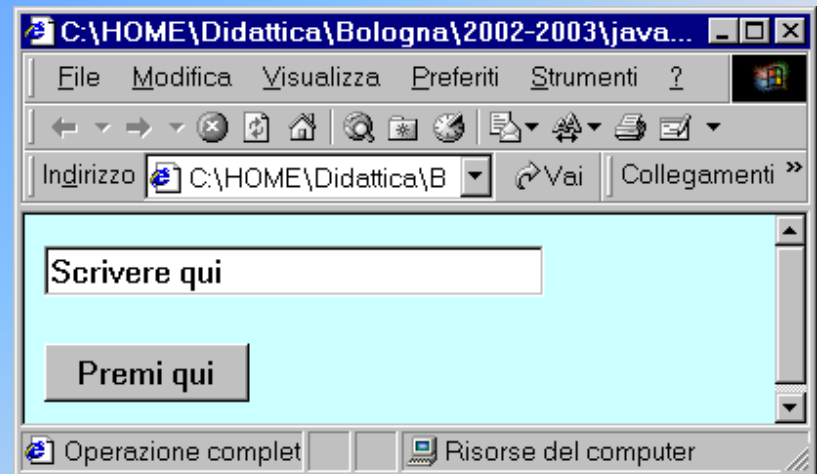


FORM e loro gestione

- Javascript è spesso usato nell'ambito di form HTML
- Un form contiene solitamente ***campi di testo*** e ***bottoni***

```
<FORM name="myform">  
  <INPUT type="text" name="campoDiTesto"  
    size=30 maxlength=30 value="Scrivere qui">  
  <P>  
  <INPUT type="button" name="bottone"  
    value="Premi qui">  
</FORM>
```

Quando il bottone viene premuto è possibile invocare una funzione Javascript



FORM e loro gestione (cont'd)

- Quando si preme il bottone, *l'evento bottone premuto* può essere intercettato mediante *l'attributo onClick*

```
<FORM name="myform">  
  <INPUT type="button" name="bottone"  
    value="Premi qui"  
    onClick = "alert('Mi hai premuto') " >  
</FORM>
```



Ricorda: virgolette e
apici vanno *alternati*

FORM e loro gestione (cont'd)

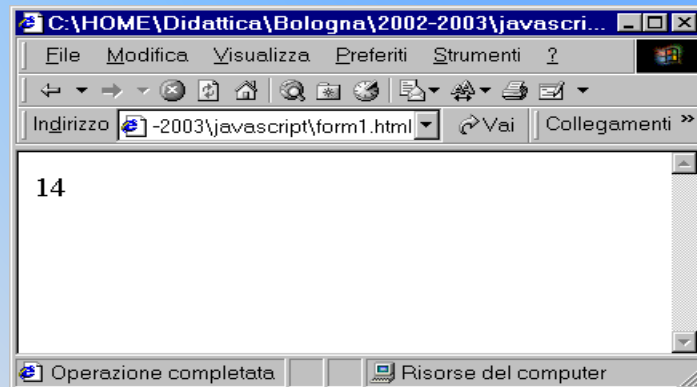
- **ALTERNATIVA:** quando si preme il bottone, far scrivere il risultato di una nostra funzione

```
<FORM name="myform">
```

```
  <INPUT type="button" name="bottone"  
    value="Premi qui"
```

```
    onClick = "document.write(sum(1,13))" >
```

```
</FORM>
```



La funzione `sum()` si
suppone definita
come in precedenza

FORM: QUALI EVENTI ?

- Gli eventi intercettabili su un link:
`onClick`, `onMouseOver`, `onMouseOut`

Gestore dentro al
tag corrispondente

- Gli eventi intercettabili su una finestra:
`onLoad`, `onUnload`, `onBlur`

- Esempio:

Gestore nel tag
<BODY... >

```
<BODY onLoad = "alert('caricato')" >
```

```
<FORM name="myform">
```

```
  <INPUT type="button" name="bottone"  
    value="Premi qui"
```

```
    onClick = "document.write(sum(1,13))" >
```

```
</FORM>
```

```
</BODY>
```

Ma anche `alert`, `confirm`, `prompt`, ...

FORM: GESTIONE degli EVENTI

- Per sfruttare il valore restituito da **confirm**, **prompt**, o *qualsiasi altra funzione Javascript* occorre inserire come valore dell'attributo **onClick** un *programma Javascript* (una sequenza o una chiamata di funzione)

Esempi:

```
onClick = "x = prompt('Cognome e Nome:') ;  
          document.write(x) "
```

```
onClick = "ok = confirm('Va bene così?') ;  
          if(!ok) alert('ATTENTO...') "
```

FORM: usare i CAMPI DI TESTO

- I campi di testo sono *oggetti dotati di nome* posti all'interno di un *oggetto form* pure esso *dotato di nome*
- Come tali sono referenziabili con la "dot notation":
document.nomeform.nomeTextField
- Il campo di testo è caratterizzato dalla proprietà *value*

Esempio:

```
<FORM name="myform">
```

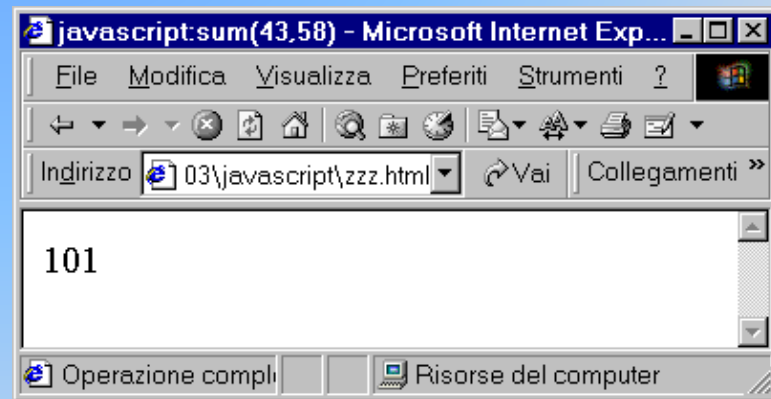
```
  <INPUT type="text" name="cognome" size=20>
```

```
  <INPUT type="button" name="bottone" value="Show"  
    onClick="alert(document.myform.cognome.value)">
```

```
</FORM>
```

FUNZIONI come LINK

- Una funzione Javascript *costituisce un valido link* utilizzabile nel tag HTML `<a href= ...> </a>`
- L'effetto del click su tale link è l'*esecuzione delle funzione* e l'apparizione del risultato in una nuova pagina HTML all'interno però della stessa finestra
- Esempio:
`<a href="Javascript:sum(43,58)" >`
Questo dovrebbe essere 101 `</a>`



Un caso particolare: Javascript e [MS]XML

Microsoft XML Parser (MSXML)

- **MSXML** è il **parser XML Microsoft**, usato da Internet Explorer e da molti altri strumenti in ambiente Windows.
- **IE 5.5** è uscito con una versione di MSXML obsoleta (**MSXML 2.5**), *non aderente agli attuali standard XML*
 - non supportava XSLT
 - però, la successiva versione MSXML 3 poteva comunque essere installata sopra la 2.6, in modalità "replace"
- **IE 6** è uscito già con la versione **MSXML 3**
- **Gli ultimi standard W3C** (in primis, XML Schema) richiedono **l'ultima versione, MSXML 4**, che tuttavia, per scelta progettuale Microsoft, ***non sostituisce mai la precedente, le si affianca soltanto ("side by side")***.

MSXML 4: problematiche

- **CONSEGUENZA:** Internet Explorer *continua a usare internamente MSXML 3 anche se sul sistema è stato installato MSXML 4 ("Microsoft XML core services")*
- **RISULTATO:** documenti corretti secondo gli attuali standard *possono non essere validati* a causa dell'uso di un parser obsoleto.

ESEMPIO

In un file XML, la processing instruction:

```
<?xml-stylesheet type="text/xsl"
  href="test.xsl"?>
```

invoca comunque le funzionalità di MSXML 3, anche se MSXML 4 è presente e installato sul sistema.

MSXML 4: schema di soluzione

- Per forzare l'uso di MSXML 4 una possibile soluzione consiste nell' **agire tramite uno script Javascript**, che
 - istanzi un oggetto parser MSXML 4.0
 - carichi tramite esso il documento XML e l'XSLT corrispondente
 - invochi tramite esso la validazione / trasformazione.

```
<script>
```

```
function loadAndTransform() {
```

```
    var xsrc = new ActiveXObject("MSXML2.DOMDocument.4.0");
```

```
    xsrc.async = false;
```

```
    xsrc.load("test.xml");
```

```
    var xslt = new ActiveXObject("MSXML2.DOMDocument.4.0");
```

```
    xsrc.async = false;
```

```
    xslt.load("test.xslt");
```

```
    document.body.innerHTML = xsrc.transformNode(xslt);
```

```
}
```

```
</script>
```

Soluzione: Esempio n.1

```
<html>
  <head>
    <title>Esempio Completo n. 1</title>
  </head>
  <body>
    <div id="result">Document is loading...</div>
    <script>
      var xsrc = new ActiveXObject("MSXML2.DOMDocument.4.0");
      xsrc.async = false;
      xsrc.load("Exp1.xml");
      var xslt = new ActiveXObject("MSXML2.DOMDocument.4.0");
      xslt.async = false;
      xslt.load("Exp1.xslt");
      result.innerHTML = xsrc.transformNode(xslt);
    </script>
  </body>
</html>
```

Funziona perché lo script compare dopo l'elemento HTML <div> che dallo script stesso viene modificato

Soluzione: Esempio n.2 (più pulito)

```
<html>
  <head>
    <title>Esempio completo n.2</title>
    <script>
      function transform() {
        var xsrc = new ActiveXObject("MSXML2.DOMDocument.4.0");
        xsrc.async = false;
        xsrc.load("Exp1.xml");
        var xslt = new ActiveXObject("MSXML2.DOMDocument.4.0");
        xslt.async = false;
        xslt.load("Exp1.xslt");
        document.body.innerHTML = xsrc.transformNode(xslt);
      }
    </script>
  </head>
  <body onload="transform()" >
  </body>
</html>
```

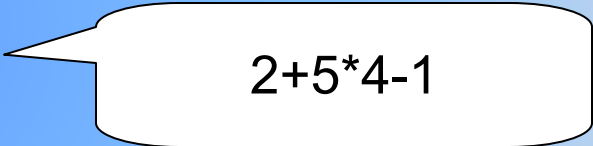
Stavolta lo script è una funzione, collocata nella head del documento e richiamata al caricamento del body.

Javascript per l'elaborazione di documenti XML: le espressioni aritmetiche

Espressioni aritmetiche in XML

- Espressioni rappresentate secondo la tassonomia già discussa più volte, sotto forma di albero XML:

```
<root>
  <!-- ===== -->
  <minusexp>
    <plusexp>
      <num>2</num>
      <timesexp>
        <num>5</num>
        <num>4</num>
      </timesexp>
    </plusexp>
    <num>1</num>
  </minusexp>
  <!-- ===== -->
  ...
</root>
```



2+5*4-1

Esplorazione XML via Javascript (1)

```
var xmlDoc = new ActiveXObject("Msxml2.DOMDocument.4.0");
xmlDoc.async = false;
xmlDoc.load("Expressions.xml");
var root = xmlDoc.documentElement;
var result;
document.write("<B><PRE>");

for (var i=0; i<root.childNodes.length; i++) {
    exp = root.childNodes[i];
    if (exp.nodeTypeString!="comment") {
        document.write("Espressione XML: <BR>");
        document.write("<font color='red'>");
        document.write("<xmp>" + exp.xml + "</xmp>");
        document.write("</font><font color='blue'>");
        document.write("Valutazione = " + evalExpr(exp) + "<HR>");
        document.write("</font>");
    }
}
document.write("</PRE></B>");
```

solo Internet Explorer

Funzione di valutazione
(Javascript)

Esplorazione XML via Javascript (2)

```
function evalExpr(exp) {  
    if (exp.tagName=="num") return exp.text;  
    else { // opexp  
        var arg1 = exp.firstChild;  
        var arg2 = exp.lastChild;  
        if (exp.tagName=="plusexp")  
            return evalExpr(arg1) + evalExpr(arg2);  
        if (exp.tagName=="minusexp")  
            return evalExpr(arg1) - evalExpr(arg2);  
        if (exp.tagName=="timesexp")  
            return evalExpr(arg1) * evalExpr(arg2);  
        if (exp.tagName=="divexp")  
            return evalExpr(arg1) / evalExpr(arg2);  
    }  
}
```

ATTENZIONE! ERRORE!

PROBLEMA: funzionano sottrazioni, moltiplicazioni e divisioni... ma la somma dà risultati assurdi! COME MAI?

Esplorazione XML via Javascript (3)

```
function evalExpr(exp) {  
    if (exp.tagName=="num") return parseInt(exp.text) ;  
    else { // opexp  
        var arg1 = exp.firstChild;  
        var arg2 = exp.lastChild;  
        if (exp.tagName=="plusexp")  
            return evalExpr(arg1) + evalExpr(arg2) ;  
        if (exp.tagName=="minusexp")  
            return evalExpr(arg1) - evalExpr(arg2) ;  
        if (exp.tagName=="timesexp")  
            return evalExpr(arg1) * evalExpr(arg2) ;  
        if (exp.tagName=="divexp")  
            return evalExpr(arg1) / evalExpr(arg2) ;  
    }  
}
```

Versione corretta

Nel caso precedente, numExp restituisce del testo
→ la conversione automatica stringa/numero scatta per sottrazioni, moltiplicazioni e divisioni... ma non per la somma perché l'operatore + fra stringhe ESISTE!
Quando trova +, la versione precedente CONCATENA!