



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA

# Introduzione a Javascript IV parte

**Fabio Vitali**

Corsi di laurea in Informatica e  
Informatica per il Management  
Alma Mater – Università di Bologna

# Oggi parleremo di...

- Javascript
  - Sintassi avanzata
  - Object orientedness e Javascript
  - Closure
  - Immediately Invoked Function Expressions





ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA

# Javascript avanzato

(in che modo è **diverso**  
dagli altri linguaggi)

# Peculiarità di Javascript

- Valori falsy e truthy
- *Funzioni come entità di prima classe*
- *Classi e prototipi*
- *Closure e IIFE*
- *Altre peculiarità di ECMA 2015*
- *Gestione dell'asincronicità*



# JS: Falsy e truthy (1/4)

- Javascript definisce come *falsy* quei valori che in caso di casting a Booleano diventano falsi:
  - *false*
  - *0*
  - *null*
  - *undefined*
  - *""*
  - *NaN*
- Ogni altro tipo di valore è *truthy* (ovvero cast a true), inclusi:
  - *"pippo", 3.14, Infinity*
  - *{}*
  - *[]*
  - *"0", "undefined", "null"*



# JS: Falsy e truthy (2/4)

Qualunque programmatore di derivazione C o Java, per vedere se un valore è falso o nullo o indefinito, scriverà una cosa tipo:

```
if (value != null && value.length >0) {  
    // ok agisci  
}
```

magari allungando pure la lista dei controlli.

In Javascript c'è il casting automatico, che permette una scrittura molto più semplice e veloce:

```
if (value) {  
    // ok agisci  
}
```



# JS: Falsy e truthy (3/4)

Se vi arrivano parametri da fuori, bisogna inizializzarli ad un valore decente se sono vuoti o falsi o non definiti. Supponiamo che param sia spesso un numero, ma non sempre:

```
if (param== null || param==0) {  
    misura = "12px" ;  
} else {  
    misura = param + 'px'  
}
```

o se ve ne ricordate, magari potreste usare l'operatore ternario:

```
misura = (param? param:'12')+ 'px' ;
```

Ma Javascript ha un modo ancora più semplice:

```
misura = (param || '12')+ 'px' ;
```



# JS: Falsy e truthy (4/4)

In pratica, moltissimi programmatori lo usano come verifica della presenza e istanziamento di una variabile o la disponibilità di una libreria o un servizio.

```
if (window.XMLHttpRequest) {  
  ...  
} else if (window.ActiveXObject) {  
  ...  
}
```

Oppure per verificare l'esistenza di un parametro opzionale, invece di:

```
function connect(hostname, port, method) {  
  if (hostname === undefined) hostname = "localhost";  
  if (port === undefined) port = 80;  
  if (method === undefined) method = "HTTP";  
  ...  
}
```

posso usare:

```
function connect(hostname, port, method) {  
  hostname = hostname || "localhost";  
  port = port || 80;  
  method = method || "HTTP" ;  
  ...  
}
```







ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA

# Funzioni in Javascript

# JS: funzioni come entità di I classe (1/6)

In Javascript, le funzioni sono oggetti quasi come tutti gli altri:

- Possono essere assegnate a variabili
- Possono essere passate come parametri di funzione
- Possono essere restituite da una funzione
- Possono essere elementi di un object
- Possono essere elementi di un array

in più:

- Possono essere invocate con l'operatore ()



# JS: funzioni come entità di I classe (2/6)

- Si può assegnarle ad una variabile:

```
var potenza = function(a,b) {  
    return Math.pow(a,b);  
}  
var c = potenza(5,3)
```

```
function potenza(a,b) {  
    return Math.pow(a,b);  
}  
var c = potenza(5,3)
```

- Si può anche avere una funzione come metodo di un oggetto:

```
e = {p:3,q:5,r:7}  
e.sum = function() {  
    return this.p+this.q+this.r;  
}
```

Function expression

Function statement

- Il nome semplice è una variabile, se si aggiungono le parentesi diventa un'invocazione e la funzione viene eseguita:

```
if (e.sum) { // controllo l'esistenza: variabile  
    var c = e.sum() // invoco ed eseguo: funzione  
}
```



# JS: funzioni come entità di I classe (3/6)

Si può assegnarle come proprietà di un oggetto o di un array:

```
var persona = {  
  nome: 'Giuseppe',  
  cognome: 'Rossi',  
  altezza: 180,  
  nascita: new Date(1995,3,12),  
  saluta: function(name, id) {  
    var saluto = "Ciao "+name  
    if (id) {  
      document.getElementById(id).innerHTML = saluto;  
    } else {  
      alert(saluto) ;  
    }  
  }  
}
```



# JS: funzioni come entità di I classe (4/6)

Si possono restituire funzioni come risultato di altre funzioni:

```
var expGenerator = function(e) {  
    return function(b) {  
        return Math.pow(b,e)  
    }  
}
```

Posso creare delle funzioni in serie chiamando il generatore:

```
var quadrato = expGenerator(2) ;  
var cubo = expGenerator(3) ;
```

E queste sono vere funzioni:

```
var c = quadrato(5) ;           // c vale 25  
var d = cubo(5) ;              // d vale 125
```

**Questa tecnica viene usata spessissimo!**



# JS: funzioni come entità di I classe (5/6)

Posso passare funzioni anonime come parametri di funzione:

```
var msg = document.getElementById('msg') ;  
msg.innerHTML = '<p>via!</p>' ;  
window.setTimeout(function() {  
    msg.innerHTML += '<p>1 secondo è passato</p>' ;  
}, 1000) ;  
window.setTimeout(function() {  
    msg.innerHTML += '<p>2 secondi sono passati</p>' ;  
}, 2000) ;  
window.setTimeout(function() {  
    msg.innerHTML += '<p>3 secondi sono passati</p>' ;  
}, 3000) ;
```

Nota: `setTimeout(f,n)` esegue la funzione `f` dopo `n` millisecondi dalla invocazione.



# JS: funzioni come entità di I classe (6/6)

La funzione `bind(obj, args)` permette di associare parametri a funzioni anonime o chiamate indirettamente:

```
var msg = document.getElementById('msg') ;
msg.innerHTML = '<p>via!</p>' ;
for (var i=1; i<=3; i++) {
    window.setTimeout(
        function(n) {
            this.innerHTML += '<p>'+n+' secondi sono passati';
        }.bind(msg, i),
        i*1000
    ) ;
}
```

Nella chiamata `bind(obj, args)`, `obj` rappresenta l'oggetto a cui verrà associata la funzione (cosa trovo dentro alla variabile predefinita `this`), mentre `args` sono gli argomenti che voglio passare alla funzione.





ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA

# Object orientedness in Javascript



# JavaScript: oggetti e classi

- JavaScript è un linguaggio object-oriented anche se non è tipato come Java.
- In un linguaggio object oriented tradizionale, la classe è un template sulla base del quale vengono istanziati gli oggetti del programma, specificando i **membri** (stati dell'oggetto, valori) e i **metodi** (comportamenti dell'oggetto, funzioni).
- JavaScript ha oggetti, ma non sono basati sul concetto di classe, ma quello di prototipo.
- Poiché le funzioni sono oggetti di primo livello, ogni oggetto può contenere al suo interno delle funzioni, senza ricorrere alla classe.
- E' possibile istanziare oggetti semplicemente dichiarandone il contenuto, oppure tramite un costruttore.



# Classi in Javascript

- Le classi non sono entità di primo livello. Al loro posto si usano degli oggetti che provengono dallo stesso costruttore.
- Un costruttore è banalmente una funzione che restituisce un oggetto. Si usa new per usarlo come costruttore dell'oggetto.

```
function Persona(nome,altezza,nascita){  // Costruttore
    this.nome = nome
    this.altezza = altezza
    this.nascita = nascita
    this.saluta = function() { return 'ciao!' }
}
```

```
var mario = new Persona("Mario", 175, new
Date(1993,6,14)); // Creazione di un oggetto
var alice = new Persona("Alice", 168); // manca un
parametro: non è un problema. alice.nascita è undefined
```



# Parentesi pedantina

I linguaggi object-oriented sono divisi in:

- Class-based (es. SmallTalk, C++, Java, C#, etc.),
  - la classe esiste come concetto esplicito e primario: le classi formano una gerarchia di tipi, l'ereditarietà avviene tra classi, gli oggetti sono istanze pure delle classi (non hanno metodi propri).
  - Il design delle interfacce precede ed è strumentale alla creazione degli algoritmi per la esecuzione dei compiti dell'applicazione. Questo facilita la compilazione e fornisce una base "contrattuale" tra creatori ed utenti degli oggetti per la garanzia del buon funzionamento del programma.
- Prototype-based (es. ECMAScript, Javascript, etc.),
  - non esiste il concetto di classe, ma quello di prototipo, una istanza primaria, astratta, sempre accessibile e modificabile, di cui le singole istanze clonano (e, se serve, modificano) sia membri sia metodi.
  - Il design delle interfacce è contemporaneo e indipendente dalla creazione degli algoritmi, e può essere modificato in qualunque momento, anche a run-time. Non c'è contratto, ma **massima flessibilità.**



# JavaScript: Prototype

- Ogni oggetto in Javascript è autonomo e si possono aggiungere tutti i metodi/proprietà che si vuole senza modificare gli altri.
- Per aggiungere proprietà/metodi condivisi da molti oggetti debbo usare l'oggetto prototype.
- Si usa per creare o riusare librerie di oggetti e metodi:
  - estendere le proprietà di un oggetto built-in nel linguaggio
  - estendere le proprietà di oggetti creati in precedenza
- Ogni oggetto javascript ha una proprietà .prototype a cui si può aggiungere un membro ed associare una funzione
- La modifica del prototipo può avvenire in qualunque momento nell'esecuzione del programma.



# Esempio di prototype

```
Persona.prototype.welcome = function(){  
    alert("Benvenuto, "+this.name+"!");  
}    // Aggiunta di una funzione al prototipo Persona  
  
mario.welcome();    // utilizzo della funzione
```



# JS: Usare prototype (1/3)

Supponiamo che facciate spesso gli stessi controlli, anche molto semplici, ad esempio che una certa stringa finisca con una certa sottostringa o un elemento abbia l'attributo id:

```
if (a.substr(-1*b.length) == b)) {  
    // ok agisci  
}  
if (c.substr(-1*d.length) == d)) {  
    // ok agisci  
}  
if (n.attributes && n.attributes['id']) {  
    // ok, agisci  
}  
if (m.attributes && m.attributes['id']) {  
    // ok, agisci  
}
```



# JS: Usare prototype (2/3)

Ovviamente possiamo definire funzioni globali:

```
function endsWith(string,value) {  
    return string.substr(-1*value.length)==value  
}  
function checkId(node) {  
    return (node.attributes && node.attributes['id'])  
}  
if (endsWith(a,b)) {  
    // ok agisci  
}  
if (checkId(n)) {  
    // ok agisci  
}
```

Ma questo viene considerato discutibile perché riempie lo spazio dei nomi globali di funzioni molto specifiche



# JS: Usare prototype (2/3)

Un approccio molto comune è modificare il prototype della classe builtin relativa:

```
String.prototype.endsWith = function(value) {  
    return this.substr(-1*value.length)==value  
}
```

```
Node.prototype.checkId = function(node) {  
    if (this.attributes && this.attributes['id'])  
        return true  
    return false  
}
```

```
if (a.endsWith(b)) {  
    // ok agisci  
}  
if (b.checkId()) {  
    // ok agisci  
}
```







ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA

# Closure e IIFE

# La closure (1/2)

Javascript non ha protezione dei membri privati di un oggetto, ma sono tutti accessibili e manipolabili.

Ad esempio, definiamo una classe Counter con uno stato privato accessibile attraverso l'interfaccia data dalle funzioni inc() e dec():

```
Counter = function() {  
    this.state = 0;                // privata  
    this.inc = function() { return this.state++; };  
    this.dec = function() { return this.state--; };  
}  
var c = new Counter();  
c.inc(); var d = c.inc() // d vale 2, corretto.  
c.state = 7; var e = c.inc() // possibile!!! Inoltre  
e vale 8!
```

Questo è molto grave: significa che l'interfaccia di inc() e dec() è solo apparente, non posso impedire l'accesso alle variabili private.



# Closure (2/2)

- Abbiamo detto che in Javascript ci sono solo due scope: quello globale e quello della funzione. Non è vero.
- C'è un terzo scope, detto *closure*, che è lo scope della funzione all'interno della quale viene definita la funzione.
- Ad esempio, una funzione che restituisce una funzione ha uno scope che è sempre accessibile alla funzione interna, ma non dal mondo esterno.



# Closure (2/2)

Otengo allora delle variabili interne veramente private:

```
Counter = function() {  
    var state = 0;                                // privata  
    return {  
        inc: function() { return ++state },  
        dec: function() { return --state }  
    }  
}  
  
var c = new Counter() ;  
c.inc() ; var d = c.inc()  // d vale 2, corretto.  
c.state = 7 ; var e = c.inc()  // e vale 3, c.state  
è lecita ma inutile.
```



# IIFE

## Immediately Invoked Function Expression

- Una function expression immediatamente invocata (IIFE) è una funzione anonima creata ed immediatamente invocata.
- Serve sostanzialmente per fare singleton (oggetti non ripetibili) dotati di closure (e quindi di stato interno privato).
- L'oggetto JQuery è il risultato di un IIFE, e così **MOLTISSIME** librerie Javascript usano IIFE:



# IIFE

## Immediately Invoked Function Expression

```
var people = (function() {  
    var persone = [] ;  
    return {  
        add:    function(p) { persone.push(p)},  
        lista: function(){ return persone.join(', ') }  
    }  
})();
```

- L'oggetto people, in questo caso, è un singleton con un array come stato interno e due metodi per accedere e modificare i valori.
- Per merito della closure, la variabile persone è accessibile da add e lista, ma NON dall'esterno.
- La coppia di parentesi alla fine invoca la funzione immediatamente e senza side effect.





ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA

**Fabio Vitali**

Dipartimento di Informatica – Scienze e Ingegneria  
Alma mater – Università di Bologna

Fabio.vitali@unibo.it

[www.unibo.it](http://www.unibo.it)