



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA

# Introduzione a Javascript III parte

**Fabio Vitali**

Corso di laurea in Informatica

Alma Mater – Università di Bologna

# Oggi parleremo di...

## Javascript

- Sintassi base (parte I)
- Javascript client-side (parte II)
- ***Sintassi avanzata (parte III)***
  - ***Sintassi avanzata***
  - ***Object orientedness***
  - ***Closure***
  - ***Immediately Invoked Function Expressions (IIFE)***

## Temi trasversali

- *Navigazione sul DOM*
- *AJAX*
- *Programmazione asincrona*
- *Modularizzazione del codice*
- *Interpolazione*
- *Routing*
- *Binding mono e bi-direzionale*





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:
  - *"any non-empty string", 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 istanziazione 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 = "GET";  
  ...  
}
```

posso usare:

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





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:

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

- Si può assegnare una funzione come metodo di un oggetto:

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

- 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  
    let c = e.sum() // invoco ed eseguo: funzione  
}
```



# JS: funzioni come entità di 1 classe (2/6)

Function expression

Function statement

- Si può assegnare ad una variabile:

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

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

- Si può assegnare una funzione come metodo di un oggetto:

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

- 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
  let 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:

```
let persona = {
  nome:      'Giuseppe',
  cognome:   'Rossi',
  altezza:   180,
  nascita:   new Date(1995, 3, 12),
  saluta:    function(name, id) {
    let 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:

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

Posso creare delle funzioni in serie chiamando il generatore:

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

E queste sono vere funzioni:

```
let c = quadrato(5) ;           // c vale 25  
let 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:

```
let 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:

```
let msg = document.getElementById('msg') ;
msg.innerHTML = '<p>via!</p>' ;
for (let 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.



# Funzioni filtro su array

Gli array di Javascript hanno tantissimi metodi che accettano una funzione come parametro. Permettono di fare specifiche operazioni sugli elementi dell'array in maniera veloce e sistematica. Ad esempio:

```
let salespeople = [  
  { name: 'Alice', cognome: 'Bruni' , sales: 78500 },  
  { name: 'Bruno', cognome: 'Verdi' , sales: 135000 },  
  { name: 'Carla', cognome: 'Rossi' , sales: 251200 },  
  { name: 'Dario', cognome: 'Bianchi', sales: 7500 }  
]
```

```
let byCognome = function (i,j) {return i.cognome > j.cognome ? 1 : -1 }  
let largerthan100 = function(i) { return i.sales >= 100000 }  
let best = function(i) { i.best = true }
```

```
let sorted = salespeople.sort(byCognome) ;
```

sorted contiene gli elementi di salespeople ordinati per cognome

```
salespeople.filter(largerthan100).forEach(best)
```

Ho selezionato solo gli elementi di salespeople con vendite >= 100000, poi ho assegnato loro il campo best a true. Ora salespeople è:

```
[ { name: 'Alice', cognome: 'Bruni' , sales: 78500 },  
  { name: 'Bruno', cognome: 'Verdi' , sales: 135000, best: true },  
  { name: 'Carla', cognome: 'Rossi' , sales: 251200, best: true },  
  { name: 'Dario', cognome: 'Bianchi', sales: 7500 }  
]
```

# Funzioni filtro su array (2)

- `array.sort(f)`
  - restituisce un array ordinato sulla base della funzione `f`, che deve avere due parametri e restituire un valore 1 (stesso ordine) o -1 (inverti l'ordine)
- `array.filter(f)`
  - restituisce un secondo array che contiene solo gli elementi che soddisfano la funzione booleana `f`.
- `array.some(f)`, `array.every(f)`
  - restituisce vero se almeno un elemento (`some()`) o tutti gli elementi (`every()`) soddisfano la funzione booleana `f`.
- `array.find(f)`
  - restituisce il primo elemento che soddisfa la funzione booleana `f`
- `array.forEach(f)`
  - esegue sull'array la funzione `f` permettendo di modificare l'array direttamente.
- `array.map(f)`
  - crea un nuovo array in cui ogni elemento viene modificato dalla funzione `f`
- `array.reduce(f)`
  - esegue su ogni elemento dell'array la funzione `f` passando il risultato dell'esecuzione precedente. Ottimo per fare totali.



# Funzioni freccia

*(arrow functions)*

Una nuova sintassi per definire funzioni:

Sintassi tradizionale

```
let square = function(x) {  
  return x * x;  
}  
let c = square(5) ;
```

Sintassi freccia

```
let square = (x) => {  
  return x * x;  
}  
let c = square(5,3);
```

Sintassi freccia senza graffe e return (può contenere una sola istruzione)

```
let square = (x) => x * x;  
let c = square(5);
```

Funzione freccia come IIFE

```
let c = (x => x * x)(5);
```



# Funzioni freccia

*(arrow functions)*

Utile per definire semplici funzioni callback, ad esempio nelle funzioni filtro degli array o negli eventi:

Sintassi tradizionale

```
let arr = [1, 2, 3];  
let squares = arr.map(function (x) { return x * x });
```

Funzione freccia su un array

```
let arr = [1, 2, 3];  
let squares = arr.map(x => x * x);
```

Funzione freccia sugli eventi

```
let showHelp = document.getElementById('showHelp') ;  
let helpDiv = document.getElementById('helpDiv') ;  
showHelp.onclick = () => helpDiv.classList.toggle('d-none')
```



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!' }
}
```

```
let mario = new Persona("Mario", 185, new Date(2002, 3, 14));
```

Creazione di un oggetto

```
let alice = new Persona("Alice", 168);
```

manca un parametro, ma non è un problema.  
**alice.nascita** è undefined



# this in Javascript

**this** è una keyword che si riferisce sempre ad un oggetto.

- All'interno di una classe, **this** si riferisce sempre all'istanza

```
function Persona(n) { // Costruttore
  this.nome = n
}
```

- All'interno di una callback di un evento, **this** si riferisce sempre all'oggetto che ha ricevuto l'evento

```
myButton.onclick = function(e) {
  this.innerHTML = "clicked!"
}
```

- se faccio **bind()** esplicito, **this** è l'oggetto a cui ho fatto **bind()**

```
function f() { return this.a; }

let g = f.bind({ a: "ciao" });
console.log( g() );
```

ciao

- altrimenti **this** è il *global this*, ovvero **window**.



# 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

Modificare il prototipo cambia non solo le istanze successive, ma anche quelle già generate in precedenza.

```
function Persona(nome, altezza) { // Costruttore
  this.nome = nome
  this.altezza = altezza
  this.nascita = nascita
}

let mario = new Persona("Mario", 185);

Persona.prototype.welcome = function() {
  alert("Benvenuto, "+this.name+"!");
}

mario.welcome(); //
```

**welcome ()** esiste anche  
per le istanze già create



# 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:

```
let a = prompt("Enter filename", "");
let b = '.html';
let c = '.pdf';

if (a.substr(-1*b.length) == b) {
    // gestisci file HTML
}
if (a.substr(-1*c.length) == c) {
    // gestisci file PDF
}
```



# JS: Usare prototype (2/3)

Ovviamente possiamo definire funzioni globali:

```
function endsWith(string,value) {
    return string.substr(-1*value.length)==value
}

let a = prompt("Enter filename","");

if (endsWith(a, '.html')) {
    // gestisci file HTML
}

if (endsWith(a, '.pdf')) {
    // gestisci file PDF
}
```

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

Questo prende il nome di *namespace pollution*



# 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  
}  
  
let a = prompt("Enter filename", "");  
  
if (a.endsWith('.html')) {  
    // gestisci file HTML  
}  
  
if (a.endsWith('.pdf')) {  
    // gestisci file PDF  
}
```



# E' corretto modificare il prototipo di una classe esistente?

- Le opinioni divergono da almeno 15 anni.
  - e.g.: <https://stackoverflow.com/questions/6223449/>
    - Q: Why is it frowned upon to modify JavaScript object's prototypes?**
    - A: The problem is that prototype can be modified in several places. For example one library will add the `map()` method to Array's prototype and your own code will add the same but with another purpose. So one implementation will be broken.
- Risposta "Controllo io le librerie del mio codice!"
  - "Se io aggiungo un metodo `map()` al prototipo di Array, so già che nessuna altra libreria che sto usando lo fa"
- Risposta "lo faccio apposta per questo!"
  - "Se io aggiungo un metodo `map()` al prototipo di Array, forse lo faccio proprio per cambiare il comportamento dell'altra libreria."
- Risposta "Basta non correre il rischio della collisione dei nomi"
  - "Invece di usare `map()`, adotto un prefisso tipo "`fv_`", e quindi definisco `fv_map()` o qualcosa del genere, e non corro rischi."





ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA

# Scope delle variabili, closure e IIFE

# Scope delle variabili in Javascript

Javascript ha tre modi per definire le variabili e quattro tipi di scope delle variabili

- variabili globali
- variabili di modulo
- variabili di funzione
- variabili di blocco



# Scope delle variabili in Javascript

## *Variabili globali*

- Ogni variabile definita esternamente alle funzioni è globale, indipendentemente dall'operatore usato (anche niente):

```
var a = 5;  
let b = 7;  
c = 9;  
  
function foobar() {  
    return a + b  
}  
console.log(foobar() - c)
```

3

- **N.B.:** Per lunga tradizione, su browser (*e solo su browser*), le variabili globali sono considerate membri dell'oggetto predefinito `window`

```
var a = 5;  
console.log(a === window.a)
```

true

Server-side è l'oggetto `module.exports`



# Scope delle variabili in Javascript

## *Variabili di modulo*

- Parleremo diffusamente di moduli più avanti
- Ogni variabile definita esternamente alle funzioni ma in un file etichettato come modulo è globale alle funzioni del modulo, ma è locale al modulo:

```
// Saved as file "imported.js"  
var a = "Io sono a" ;  
  
export f: function() {  
    return a;  
}
```

```
import {f} from ('../imported.js');  
  
function g() {  
    var b = f();  
    console.log(b) ;  
    console.log(a) ;  
}
```

*io sono a  
undefined*



# Scope delle variabili in Javascript

## *Variabili locali*

- Ogni variabile definita con `var` internamente ad una funzione è locale alla funzione:
- Lo scope locale sovrascrive lo scope globale:

```
var a = 5;
let b = 7;
c = 9;

function foobar() {
  var a = 10 ;
  var d = 3 ;
  return a + d ;
}
console.log(foobar() - c)
```

4



# Scope delle variabili in Javascript

## *Variabili di blocco*

- Ogni variabile definita con `let` o `const` internamente ad un blocco parentetico è locale al blocco ed è definita da quel momento in poi:
- Lo scope di blocco sovrascrive gli altri scope:

```
var a = 5;

function foobar(b) {
  if (b > 3) {
    let a = 10 ;
    return a + b;
  }
  return a + b
}

console.log(foobar(2))
console.log(foobar(5))
```

```
7
15
```



# 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 `incrementa()` e `decrementa()`:

```
Counter = function() {
  this.state = 0; // variabile che vorrei fosse privata
  this.incrementa = function() {
    return ++this.state
  };
  this.decrementa = function() {
    return --this.state
  };
}
var c = new Counter() ;
c.incrementa() ;
var m = c.incrementa() ;
c.state = 7 ;
var n = c.incrementa() ;
```

m vale 2, corretto.

**Possibile!!! n vale 8!**

Questo è molto grave: significa che l'interfaccia limitata a `incrementa()` e `decrementa()` è solo apparente, non posso impedire l'accesso alle variabili private.



# Closure (2/2)

- Abbiamo detto che in Javascript ci sono solo quattro scope: quello globale, modulo, funzione, blocco. Non è vero.
- C'è un quinto scope, detto *closure*, che è lo scope della funzione all'interno della quale viene definita un'altra 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)

Ottengo allora delle variabili interne veramente private:

```
Counter = function() {  
  var state = 0; // privata  
  return {  
    incrementa: function() { return ++state },  
    decrementa: function() { return --state }  
  }  
}  
  
var c = new Counter() ;  
c.incrementa() ;  
var m = c.incrementa() ;  
c.state = 7 ;  
var n = c.incrementa() ;
```

m vale 2, corretto.

n vale 3

**c.state=7 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

# Altri aspetti di Javascript

# Definizioni di classe

Zucchero sintattico: un modo compatto per creare oggetti in maniera retro-compatibile con Java e C++:

## *Sintassi tradizionale*

```
let Shape = function(id,x,y) {  
  this.id = id;  
  this.x = x ;  
  this.y = y ;  
};  
Shape.prototype.move = (x,y) =>  
{  
  this.x = x;  
  this.y = y;  
};
```

## Sintassi con classi

```
class Shape {  
  constructor (id,x,y) {  
    this.id = id  
    this.x = x ;  
    this.y = y;  
  }  
  move (x,y) {  
    this.x = x ;  
    this.y = y ;  
  }  
}
```

Gli oggetti sono ancora basati sui prototipi, ma le definizioni sono più semplici.



# Template literal

ES 2015 fornisce una terza sintassi per definire stringhe multi-linea con interpolazione di variabili (*ne riparleremo*).

```
let firstName = 'Jane';  
let x = `Hello ${firstName}!  
How are you  
today?`;
```

```
"Hello Jane!  
How are you  
today?"
```

Tre elementi fondamentali: :

- Backticks come delimitatori: `
- I new line fanno parte della stringa
- Interpolatori: `${varName}`

Molto utile per sbarazzarsi dell'incubo delle virgolette annidate.



# Template literal

Esempio frequentissimo:

*Sintassi tradizionale*

```
let items = ["first", "second", "third"] ;
let list = document.getElementById('linkList');
for (var i=0; i< item.length; i++) {
  let link =
    "<li><button onclick='goTo(\""+items[i]+"\") '>"+items[i]+"</button></li>";
  list.appendChild(link) ;
}
```

*Con template literal*

```
let items = ["first", "second", "third"] ;
let list = document.getElementById('linkList');
for (var i=0; i< items.length; i++) {
list.appendChild(
  `- <button onclick='goTo("${items[i]}") '>${items[i]}</button></li>` ) ;
}

```

# Optional chaining (ES 2020)

Sia dato un array contenente due oggetti come al solito:

```
let persone = [{
  nome:      'Giuseppe',
  cognome:   'Rossi',
  altezza:   180,
  nascita:   new Date(1995,3,12),
  indirizzo: {
    via: { strada: 'Via Indipendenza', numero: '15' },
    citta: 'Bologna',
    nazione: 'Italia'
  }
}, {
  nome:      'Andrea',
  cognome:   'Verdi',
  altezza:   175,
  nascita:   new Date(1994,7,9),
  email:     'andrea.verdi@gmail.com'
}] ;
```

il primo oggetto contiene il membro `indirizzo`, il secondo no.



# Optional chaining (ES 2020)

Se provo a fare un ciclo sull'array che accede ai campi degli oggetti, può succedere una cosa così:

```
for (let i in persone) {  
    console.log( persone[i].indirizzo.via.numero )  
}
```

Ottingo un errore:

```
Uncaught TypeError: Cannot read property 'via' of undefined
```

Per evitare l'errore dovrei controllare sistematicamente la catena dei campi dell'oggetto:

```
for (let i in persone) {  
    if (persone[i])  
        if (persone[i].indirizzo)  
            if (persone[i].indirizzo.via)  
                console.log( persone[i].indirizzo.via.numero )  
}
```

# Optional chaining (ES 2020)

Optional chaining è un'introduzione sintattica di ES 2020 per evitare questi controlli sistematici.

Posso inserire l'operatore di sequenza opzionale '?.' invece che '!'.

Se uno degli elementi della chain è `undefined`, la sequenza restituisce `undefined` invece che un runtime error.

```
for (let i in persone) {  
  console.log( persone[i]?.indirizzo?.via?.numero )  
}
```



# L'operatore spread ...

L'operatore spread (...) permette di spalmare i singoli elementi di un elemento strutturato (un array, un oggetto, un *iterable*) dove ci si aspetterebbe di trovare i singoli elementi ad uno ad uno.

Alcune applicazioni:

- concatenare array

```
let a1 = [1, 2, 3];  
let a2 = [4, 5, 6];  
let a3 = [...a1, ...a2];
```

a3 contiene [1, 2, 3, 4, 5, 6]

- unire o clonare oggetti

```
let fv = { nome: "Fabio", cognome: "Vitali" };  
let fv2 = { ...fv, professione: "docente" };  
let clonedfv = { ...fv };
```

- spalmare un *iterable* tra gli argomenti di una funzione

```
let fvAsArray = ["Fabio", "Vitali"];  
let fullName = (name, surname) => name + ' ' + surname;  
let myFullName = fullName(...fvAsArray);
```



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)