

Javascript

Ingegneria del software

Vincenzo Bonnici

Corso di Laurea in Informatica

Dipartimento di Scienze Matematiche, Fisiche e Informatiche

Università degli Studi di Parma

2025-2026

- E' un linguaggio di programmazione creato nel 1995 da Brendan Eich (Netscape)
 - Originariamente chiamato "Livescript"/ Mocha: nato come linguaggio "semplice" per i non sviluppatori (designers/scripters/amateurs)
- **Javascript != Java** (chiamato cosi' solo per marketing!)
- Standard nel 1996 da European Computer Manufacturer's Association (**ECMA**)
 - Chiamato **ECMAScript** - ISO/IEC 16262
 - JS è un implementazione (dialetto) di ECMAScript
- Versione attuale dello standard: ECMAScript 2020
(https://www.w3schools.com/js/js_versions.asp)

Ruolo di Javascript

- Web client-side scripting
 - praticamente monopolista!
- E non solo...
 - es. NodeJs



Cosa fa Javascript?

Instant search

Web Images Videos Maps News Shopping Gmail more ▾

Google

Everything Images Videos More

Show search tools

Weather

- weather
- walmart
- white pages
- wikipedia
- W

About 1,110,000,000 results (0.23 seconds)

Weather for New York, NY - Change location - Add to iGoogle

89°F | °C Wed Thu Fri Sat

Current: Partly Cloudy Wind: W at 16 mph Humidity: 21%

82°F | 60°F | 76°F | 58°F | 73°F | 58°F | 78°F | 63°F

Detailed forecast: The Weather Channel - Weather Underground - AccuWeather

National and Local Weather Forecast, Hurricane, Radar and Report ☆
The Weather Channel and weather.com provide a national and local weather forecast for

Live chat

Nuovo messaggio

A: Ingegneria di Internet ✖

Scrivi un messaggio come Lorenzo Bracciale...

Aggiungi file Aggiungi foto Invia

Aggiungere Javascript ad una pagina HTML

```
<script type="text/javascript">  
    // Scrivi qui il tuo codice javascript  
</script>
```

embedded

```
<script type="text/javascript" src="my_script.js">  
</script>
```

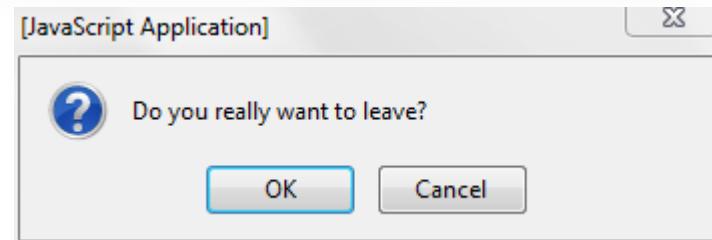
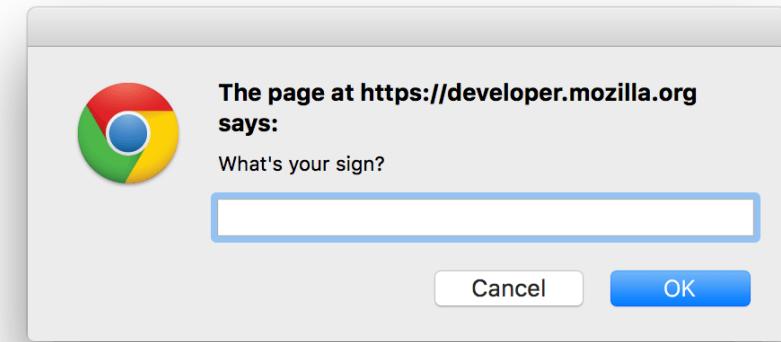
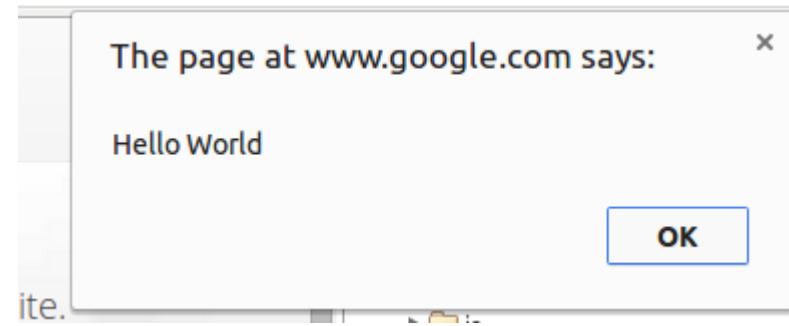
external

Solitamente inclusi in

- <head> : procedure JS indipendenti dalla specifico contenuto della pagina/istanza HTML
- o alla fine del <body> : procedure che operano su specifici elementi HTML con ID e che devono aspettare il caricamento di tutta la pagina per averli sicuramente a disposizione (in realtà c'è anche l'evento onLoad)

Statements e commenti

```
// visualizziamo una finestra popup con un messaggio  
alert('ciao');  
/* visualizziamo una finestra in cui chiediamo una domanda  
all'utente.  
La funzione ritorna il valore immesso */  
prompt('come ti chiami?');  
// mostra una finestra con "Ok"/ "Annulla".  
// Ritorna la scelta fatta  
confirm('sei uno studente di PW?'); //
```



; alla fine di uno statement è un a «best practice»

Variabili e concatenazione

```
// definisco una variabile  
var myName;  
// la assegno all'output della funzione "prompt"  
myName = prompt('come ti chiami?');  
// concateno due stringhe  
alert('ciao ' + myName );
```

Tipi di dato

Ogni dato appartiene ad un tipo:

- ma non serve specificarlo nella dichiarazione (dynamically typed)
- Ci sono tipo "primitivi" (questi) e tipi "complessi" (oggetti, array, funzioni)

<code>var test = 5;</code>	la variabile è un numero . (anche "float": 5.123) Operazioni: + - * / (es 5 + 3.1)
<code>var test = "ciao";</code>	la variabile è una stringa . (singoli o doppi apici) Concatenazione con + (es: "ciao" + ' a tutti')
<code>var test = true;</code>	variabile booleana Inverso con ! (es !test è false)
<code>var test;</code>	la variabile è undefined
<code>var test = null;</code>	la variabile è null

```
const prefix = '06';
```

// read only

```
// case matters
```

```
var programmazioneWeb;
```

```
var programmazioneweb;
```

- **Dynamic**: non è compilato, gira in una “macchina virtuale”
- **Loosely typed**: non bisogna dire che tipo ha una variabile
- **Case-sensitive**: aTTenZione all'E MaiUscole!

Esempio

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4
5  </head>
6  <body>
7
8
9      <script type="text/javascript">
10
11      var n1, n2, somma;
12
13      n1 = prompt("Numero 1:");
14      n2 = prompt("Numero 2:");
15
16      somma = parseFloat(n1) + parseFloat(n2);
17
18      somma = alert("La somma fa " + somma);
19
20
21      </script>
22
23  </body>
24  </html>
```

Description: The parseInt() function parses a string argument and returns an integer of the specified radix (the base in mathematical numeral systems).

Syntax: parseInt(string, radix);

Description: The parseFloat() function parses a string argument and returns a floating point number.

Syntax: parseFloat(string);

Oggetti

Un oggetto è una **lista di coppie “proprietà: valore”**, racchiuse in parentesi angolari {}

- Un valore può essere un tipo primitivo, un altro oggetto o una funzione

```
var studente = {  
    name: "Pierpaolo Loreti" stringa  
    age: 80, intero  
    scores: [1,2,3], array  
    classes: {pw: 30, fi: 18} altro oggetto  
};
```

da qui: JavaScript Object Notation (JSON)

Non esistono metodi/attributi privati

Oggetti

```
var studente = {} // oggetto vuoto  
studente = new Object() // stessa cosa
```

Creo oggetto

```
studente.voto = 30;
```

Aggiungo proprietà

```
console.log(studente.voto); // 30  
console.log(studente['voto']); // stessa cosa
```

Accedo

```
delete studente.voto;
```

Rimuovo proprietà

```
console.log(studente.voto); // undefined
```

Un “garbage collector” rimuove le variabili che non ci servono dalla memoria automaticamente

- Lo capisce quando non abbiamo più “riferimenti” ad un oggetto
- Possiamo dichiarare nuove variabili dinamicamente senza preoccuparci* di rimuoverle dalla memoria

* in realtà un po' dovremmo preoccuparci, anche qui possiamo fare dei “memory leak”...

more info: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management

Array

- **Contenitori** di variabili (anche con tipi diversi)
- Sono oggetti con **proprietà** numeriche e **metodi/attributi** per “maneggiarli”
- Ogni membro dell’array ha un **indice** (che parte da 0)

Definizione

```
var myFirstArray = [ 5, "ciao", false, undefined];
```

Accesso

```
alert(myFirstArray[1]); // alert("ciao")  
myFirstArray.length; // ritorna 4
```

Array

- **Creazione** di un array (equivalenti)
 - var arr = new Array(element0, element1, ..., elementN);
 - var arr = Array(element0, element1, ..., elementN);
 - var arr = [element0, element1, ..., elementN];
- **Modificare** un membro
 - myFirstArray[0] = “nuovo valore”;
- **Aggiungere** un membro
 - myFirstArray.push(“ciao”) //aggiunge alla fine
 - myFirstArray[10] = “ciao”; // aggiunge al decimo posto
(che succede a length?)
- **Rimuovere** un membro
 - myFirstArray.pop() // rimuove ultimo elemento ritornandolo
 - metodo splice // vedi dopo
 - delete myFirstArray[10] // rimuove l’elemento ma non sposta gli indici dell’array
- **Lunghezza** dell’array
 - myFirstArray.length
- **Svuotare** un array
 - myFirstArray = []
 - myFirstArray.length = 0
 - manualmente con cicli e splice (vedi dopo)

```
var colors = ['red', 'green', 'blue'];
colors.forEach(function(color) {
  console.log(color);
});
var list = myArray.join(" - "); // "red - green - blue"
```

```
var a = ['a', 'b', 'a', 'b', 'a'];
console.log(a.indexOf('b'));
```

Array: insidie

```
var a = ['a', 'b', 'c'];
a.length;                      ritorna 3
delete a[0];
a.length;                      [undefined, 'b', 'c']
                                ritorna sempre 3
a[10] = 'd';
a.length;                      ritorna 11
```

Stringhe

```
var s = "Ciao a tutti";
// una stringa è un istanza dell'oggetto String
var s = new String("Ciao a tutti"); //equivalente (auto boxing)

s.indexOf(' a '); // ritorna 4 (prima occorrenza)
s.slice(1); // ritorna "iao a tutti" (non modifica la stringa)
s.trim(); // leva whitespaces ad inizio e fine stringa

s = "ciao a \
tutti"; // andare a capo
```

Date

L'oggetto built-in “Date” ha metodi e costanti per le date
(js non ha il tipo primitivo “data”)

```
today = new Date() // data di oggi
var Xmas95 = new Date("December 25, 1995 13:30:00")
Xmas95 = new Date(1995, 11, 25)
var Xmas95 = new Date(1995, 11, 25, 9, 30, 0)

Xmas95.getMonth() // ritorna 11
Xmas95.getFullYear() //ritorna 1995.
getTime() // ritorna i millisecondi dal 1-1-1970
```

Typeof e instanceof

- `typeof true; // returns "boolean"`
 - `typeof 62; // returns "number"`
-
- `var theDay = new Date(1995, 12, 17);`
 - `theDay instanceof Date; // true`

Operatori

Di confronto

<code>== e !=</code>	Uguale e non uguale
<code>> e >= e < e <=</code>	Maggiore, maggiore o uguale, minore, minore o uguale
<code>==!= e !==</code>	Identico o non identico (stesso dato e tipo)

```
var a = "5"; // stringa contentente il numero 5
var b = 5; //numero 5
alert(a == b); // true
alert(a === b); // false
```

Matematici

<code>a++ o a--</code>	Aggiunge uno o rimuove uno alla variabile a
<code>a += 4</code>	Aggiunge 4 alla variabile a

Bit a bit

<code>a b</code>	OR bit a bit
<code>a & b</code>	AND bit a bit
<code>~a</code>	NOT

If, else e switch

Undefined, "", 0, false, null sono interpretati come "falsy"

```
if (a == 5) {  
    alert("a e' 5");  
} else if (a > 5) {  
    alert("a e' maggiori di 5");  
} else {  
    alert("a e' minore di 5 (o non e' un numero!) ");  
}
```

IF ternario

```
var status = (age >= 18) ? "adult" : "minor";
```

```
switch (expression) {  
    case label_1:  
        statements_1  
        [break;]  
    case label_2:  
        statements_2  
        [break;]  
        ...  
    default:  
        statements_def  
        [break;]  
}
```

```
/* ciclo for */
for(var i = 0; i <= 10; i++) {
    console.log(i);
}
```

```
/* ciclo while */
var i = -20;
while(i > 0) {
    console.log(i);
    i++;
}
```

- **for...in** itera per le proprietà di un oggetto
- **for...of** itera per gli elementi di un array/mappa/set (un "iterabile")

```
var arr = [3, 5, 7];
arr.foo = "hello";

for (var i in arr) {
    console.log(i); // logs "0", "1", "2", "foo"
}

for (var i of arr) {
    console.log(i); // logs "3", "5", "7"
}
```

- L'oggetto built-in “Math” ha metodi e costanti per operazioni matematiche

Funzioni/costanti	Descrizione
sin(), cos(), tan()	Funzioni trigonometriche
pow(), exp(), expm1(), log10(), log1p(), log2()	Funzioni esponenziali
min(), max()	Ritornano min o max di una lista
random()	Ritorna un numero casuale tra 0 e 1
PI	costante per pi greco
round(), fround(), trunc()	Arrotondamenti e troncamenti

Esempio:

```
Math.random() // 0.932132321
```

Eccezioni

- Indicano che qualcosa è andato storto...
- es. jnkdfsnjkfd (); // ReferenceError: jnkdfsnjkfd is not defined
- Un eccezione può essere qualunque tipo di dato (oggetto, stringa, numero...)
- Il frammento di codice “lancia” un’eccezione (throw), che puo’ essere gestita (catch)
 - 1. Interrompe la normale esecuzione
 - 2. cerca una routine in grado di risolvere il problema (catch)
 - 3. se “gestita”, il flusso continua da dopo il blocco “catch”

```
function getMonthName(monthId) {  
    if (monthId == 1) { return "Gennaio";}  
    else if (monthId == 2) { return "Febbraio";}  
    /* ... */  
    else if (monthId == 12) { return "Dicembre";}  
    else {  
        throw "Il mese non e' valido";      C'e' un problema, lancio l'eccezione  
    }  
}
```

```
function f(myMonth) {  
    try {  
        monthName = getMonthName(myMonth);  
    }  
    catch (e) {  
        monthName = "unknown";  
    }  
    finally {  
        // eseguita in ogni caso (ad es. chiudi un file)  
    }  
}
```

Gestisco l’eccezione

Error object

- Struttura dati “errore generico” per l’eccezione.
- Due proprietà:
 - 1. name: errore sintetico (“DOMException”)
 - 2. message: descrizione verbosa dell’errore

```
throw (new Error('The message'));
```

Esistono Errori più specifici (ReferenceError, URIError ...) lista completa su:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error#Error_types

Dichiarazione di funzioni

```
// dichiarazione di funzione
// argomenti separati da virgola (o nessun argomento!)

function somma(a, b) {
    return a + b;
}

// OPPURE

var somma = function(a,b) { return a + b; }

// invocazione (in entrambi i casi)
somma(5, 6); // ritorna 11
```

Che succede se passiamo più di 3 argomenti a questa funzione?
Che succede se passiamo meno di 3 argomenti a questa funzione?

- Sono trattate **come "oggetti"**
(caratteristica presa dal linguaggio Schema)

Possiamo **assegnarle** a variabili

```
var sum = function(x,y) {return x+y;}
```

Possiamo **passarle** a come argomento a funzioni

```
function sumOfSquare(x,y, sum) {  
    return sum (x*x, y*y);  
}
```

Possiamo **ritornare** da un'altra funzione

```
function differentSum() {  
    var sum = function(x,y) {return x+y;}  
    return sum;  
}  
// come invoco differentSum?
```

Scope

- Variabili locali definite dentro una funzione hanno lo scope relative al blocco della funzione stessa (**local scope**)
- Quando definiamo una variabile fuori da ogni funzione, è definita nel **global scope** e diventa visibile da ogni altro javascript che gira nella pagina
 - potenzialmente pericoloso! Interazioni non volute, spazio dei nomi ristretto!
 - **namespace pollution** (ovvero creazioni di variabili globali)
- Tutto quello definito nel global scope è una proprietà dell'oggetto “**window**”
 - `a = 4; //globale` `window.a = 4;` sono due espressioni equivalenti!
 - quindi per cancellare una variable globale basta `delete window.a;`

```
// global scope
var a = 5;
function x() {
    return a + 1;
}
x(); // ritorna 6
a; // 5
```

```
// local scope
function x() {
    var a = 5;
    return a + 1;
}
x(); // ritorna 6
a; // undefined
```

Dichiarazione implicita

- Le variabili globali diventano proprietà dell'oggetto “window”
- Questi due codici sono equivalenti

```
function foo() {  
    var x;  
    x = 5;  
    y = 6;  
    return x + y;  
}
```



```
function foo() {  
    var x;  
    x = 5;  
    window.y = 6;  
    return x + window.y;  
}
```

Var e let

- lo scope di var è il functional block più vicino
- lo scope di let è l'enclosing block più vicino

```
for(let i=0; i<2; i++) {  
    console.log(i);  
}  
console.log(i); // i is not defined
```

```
for(var i=0; i<2; i++) {  
    console.log(i);  
}  
console.log(i); // i = 2
```

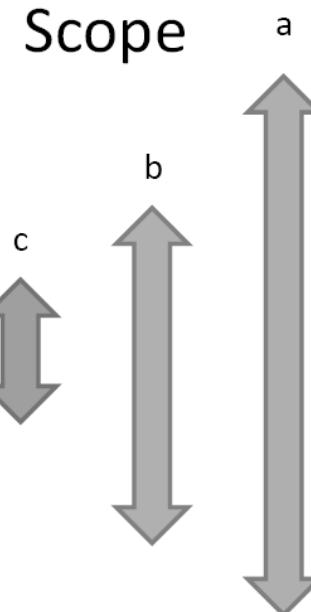
Funzioni e oggetti

- Possiamo definire funzioni dentro altre funzioni
- La funzione “nested” può accedere allo scope della funzione che la include (oltre che allo scope globale)

```
function molto_fuori() {  
    var a = 5;  
    function fuori() {  
        var b = 6;  
        function dentro() {  
            var c = 7;  
            console.log(a,b,c);  
        }  
        return dentro();  
    }  
    return fuori();  
}  
molto_fuori();
```

```
function x(p1) {  
    var a = p1; //outer scope di y  
    function y() {  
        return a*2;  
    }  
    return y();  
}
```

funzione “nested”
(inner function)



L'inner function può accedere allo scope delle outer functions
l'outer function non può accedere allo scope delle inner functions

Funzioni che ritornano funzioni

```
function multisum(p1, a, b) {  
    var x = p1;  
    function sum(a, b) {  
        return x * (a + b);  
    }  
    return sum(a,b);  
}
```

multisum(10, 1,2) ← torna 30

La funzione "multisum" ritorna l'output di "sum", ovvero ritorna un numero

```
function multisum(p1) {  
    var x = p1;  
    return function sum(a, b) {  
        return x * (a + b);  
    }  
}
```

multisum(10); ← torna una funzione

multisum(10)(1,2) ← torna 30

Closure

```
function multisum(p1) {  
    var x = p1;  
    return function sum(a, b) {  
        return x * (a + b);  
    }  
}
```

Ambiente (scope outer function)

Inner function (lo scope "si chiude" su quello del padre)

```
function salutatore(name) {  
    var text = 'Ciao' + name; // Local variable  
    var diCiao = function() { alert(text); }  
    return diCiao;  
}  
  
var s = salutatore('Lorenzo');  
s(); // alerts "Ciao Lorenzo"
```

"s" non memorizza solo il return della funzione "salutatore" (che è una funzione), ma anche il suo scope esterno (ad es la variabile "text")

Closure

```
function multisum(p1) {  
    var x = p1;  
    return function sum(a, b) {  
        return x * (a + b);  
    }  
}
```

Ambiente (scope outer function)

Inner function (lo scope "si chiude" su quello del padre)

```
function salutatore(name) {  
    var text = 'Ciao' + name; // Local variable  
    var diCiao = function() { alert(text); }  
    return diCiao;  
}  
  
var s = salutatore('Lorenzo');  
s(); // alerts "Ciao Lorenzo"
```

"s" non memorizza solo il return della funzione "salutatore" (che è una funzione), ma anche il suo scope esterno (ad es la variabile "text")

Closure: perché sono utili

```
function counter() {  
    var a = 0;  
    return {  
        inc: function() { ++a; },  
        dec: function() { --a; },  
        get: function() { return a; },  
        reset: function() { a = 0; }  
    }  
}
```

Definizione



```
var c = counter();  
c.inc();
```

Utilizzo

Ho quindi metodi o attributi privati (se faccio “c.a” ottengo un errore)

Simulo object oriented programming

Impostiamo un programma JS - Independently Invoked Functional Expression

mioprogramma.js

```
var a = 0;  
var b = 0;  
  
function pippo(x,y) {  
    // qui mettiamo del codice  
    return x*y;  
}  
  
// ... altro
```

Problema

Ho scritto a,b e pippo nel global scope!
sono l'unico a usare questi nomi?

Soluzione

Mettiamo tutto il codice in una funzione!
Invochiamo questa funzione all'avvio

mioprogramma.js

```
(function() {  
    var a = 0;  
    var b = 0;  
  
    function pippo(x,y) {  
        // codice di esempio  
        return x*y;  
    }  
})();
```

Funzione ANONIMA invocata immediatamente

- Un oggetto può avere come proprietà una funzione

- La parola chiave **this** usata dentro la **funzione** indica l'oggetto che la contiene

- Dipende dal contesto, la stessa funzione puo' indicare come "this« oggetti diversi

```
var `studente = {  
    name: "pippo",  
    getName : function() {  
        return this.name;  
    }  
}
```

!!! Attenzione al contesto !!!

```
var x = 9;  
var module = {  
    x: 81,  
    getX: function() { return this.x; }  
};
```

```
module.getX(); // 81
```

fuori da un oggetto (scope
globale)
this === window

```
var getX = module.getX;  
getX(); // quanto ritorna?
```

- Posso creare un oggetto normalmente ...

```
var jimmy = {name: "Jimmy", color: "pink", age: 0};  
var timmy = {name: "Timmy", color: "pink", age: 0};
```

- Oppure posso usare una funzione che imposta le proprietà dell'oggetto

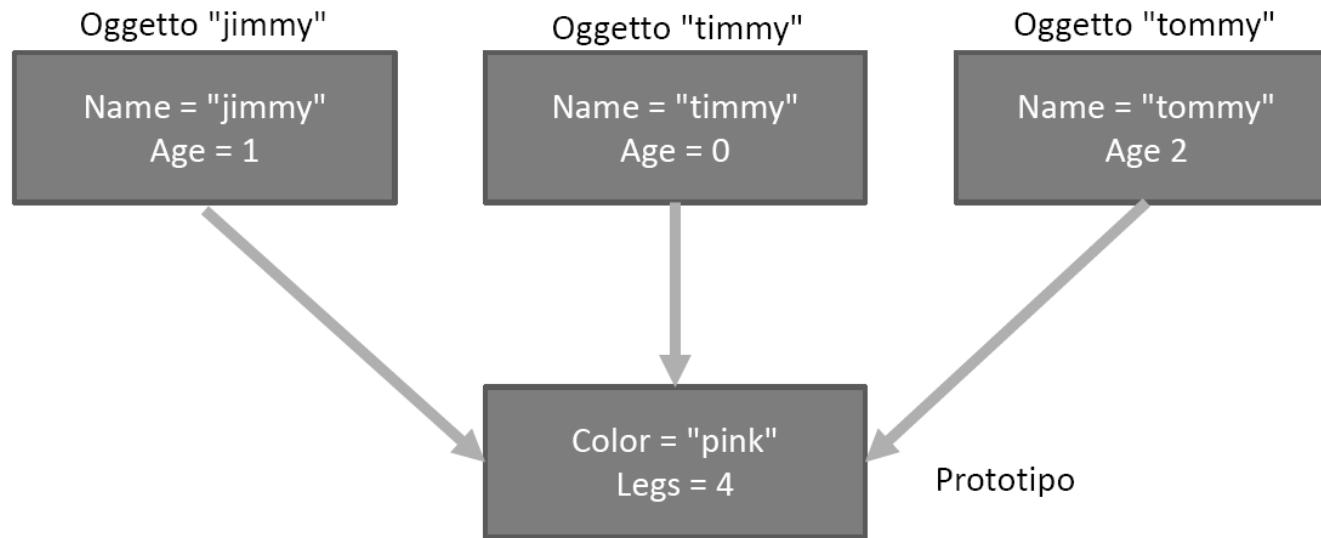
```
function Pig(name, color) {  
    this.name = name;  
    this.color = color;  
    this.age = 0;  
}  
var tommy = new Pig("Tommy", "pink");
```

- Questa funzione si chiama "costruttore"

- E' una funzione normalissima, ma la usiamo per "costruire" un oggetto

Prototipi di oggetto

- In JS gli oggetti hanno un prototipo, che è un altro oggetto da cui eredita tutte le proprietà



Possiamo vedere il prototipo di un oggetto scrivendo:

```
myObject.__proto__
```

Oggetti: Proprietà di base e ereditate

- Quando chiamiamo una proprietà di un oggetto:
 - prima si cerca tra le proprietà dell'oggetto
 - poi tra le proprietà del prototipo
 - poi tra le proprietà del prototipo del prototipo (etc)

[Prototype chain]

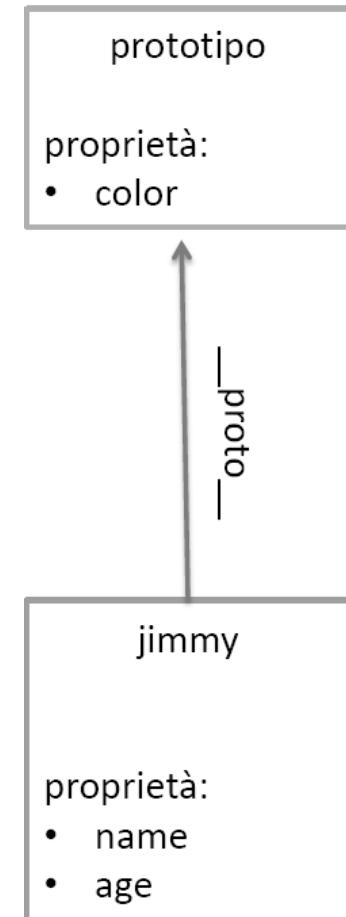
- Esempio

 jimmy.name -> trova la proprietà nell'oggetto
 jimmy.color -> trova la proprietà nel prototipo

- Possiamo vedere se la proprietà è dell'oggetto o del prototipo con:

```
  jimmy.hasOwnProperty('color'); // false  
  jimmy.__proto__.hasOwnProperty('color'); // true
```

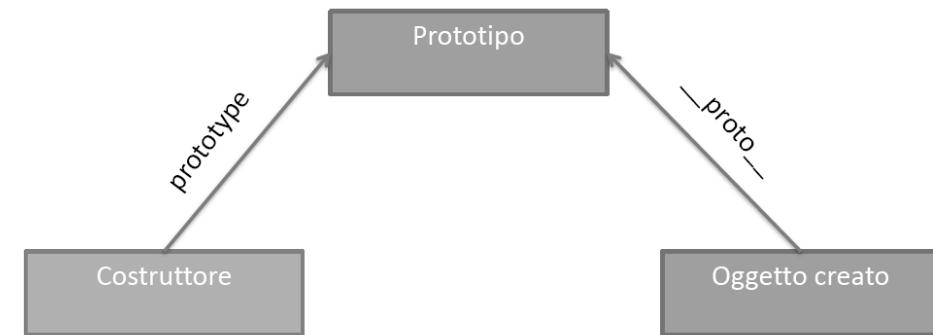
- Quindi possiamo ridefinire (override) alcune proprietà del prototipo



Prototipi di funzione

- Ogni funzione ha la proprietà “prototype” il cui valore è un oggetto
- Scrivendo Pig.prototype.color = "pink" assegniamo una proprietà a quell’oggetto
- Tutti gli oggetti creati con questo costruttore, avranno come prototipo il prototipo della funzione

```
function Pig(name, age) {  
    this.name = name;  
    this.age = age;  
}  
Pig.prototype.color = "pink";  
  
var tommy = new Pig("Tommy", 2);
```



1. Viene creato un nuovo oggetto vuoto
2. Viene passato al costruttore, in modo che ci possa riferire con “this”
3. Il costruttore setta le proprietà dell’oggetto
4. Il costruttore imposta:

prototipo dell’oggetto creato = prototipo della funzione
Pig.prototype -> tommy.__proto__

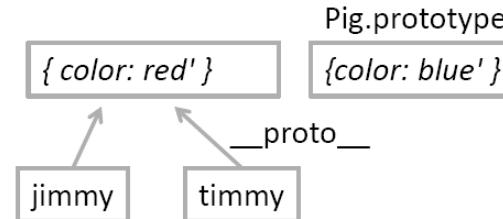
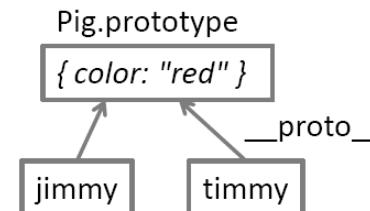
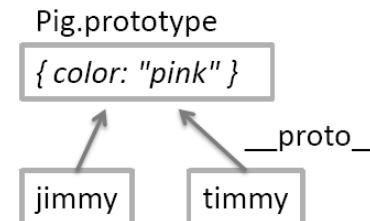
Prototipi

- **Prototipi di funzione:** è l'istanza di un oggetto che diventerà il prototipo per tutti gli oggetti creati usando la funzione come costruttore
- **Prototipi di oggetto:** è l'istanza dell'oggetto dal quale l'oggetto è ereditato

```
function Pig(name, age) {  
    this.name = name;  
    this.age = age;  
}  
Pig.prototype.color = "pink";  
  
var jimmy = new Pig("Jimmy", 1);  
var timmy = new Pig("Timmy", 2);  
  
jimmy.color;  
timmy.color;
```

```
Pig.prototype.color = "red";  
jimmy.color;  
timmy.color;
```

```
Pig.prototype = {color: "blue"};  
jimmy.color;  
timmy.color;
```



- Il metodo bind ci permette di definire chi è il “this“ per una funzione
 - metodo di una funzione -> la funzione è un oggetto! (Function object)

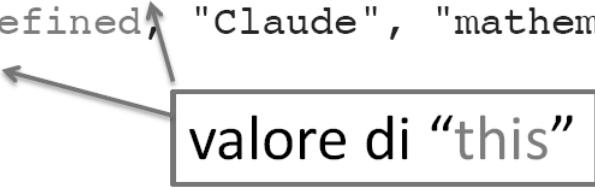
```
var a = {id: 10};  
var x = function() {return this.id}  
var w = x.bind(a);  
  
x(); // ritorna undefined  
w(); //ritorna 10
```

Apply e call

- apply per chiamare una funzione impostando un certo this e passando gli argomenti come array
- call come apply, ma gli argomenti sono passati esplicitamente

```
myFunction.apply(myObject, ["Susan", "teacher"]);
myFunction.call(undefined, "Claude", "mathematician");
```

valore di “this”



```
function Car(maker, model, year) {
  this.maker = maker;
  this.model = model;
  this.year = year;
}
var mycar = new Car("FIAT", "500", 1936);

// oppure ...
var new_car = new Object()
Car.apply(new_car, ["FIAT", "500", 1936]);
```

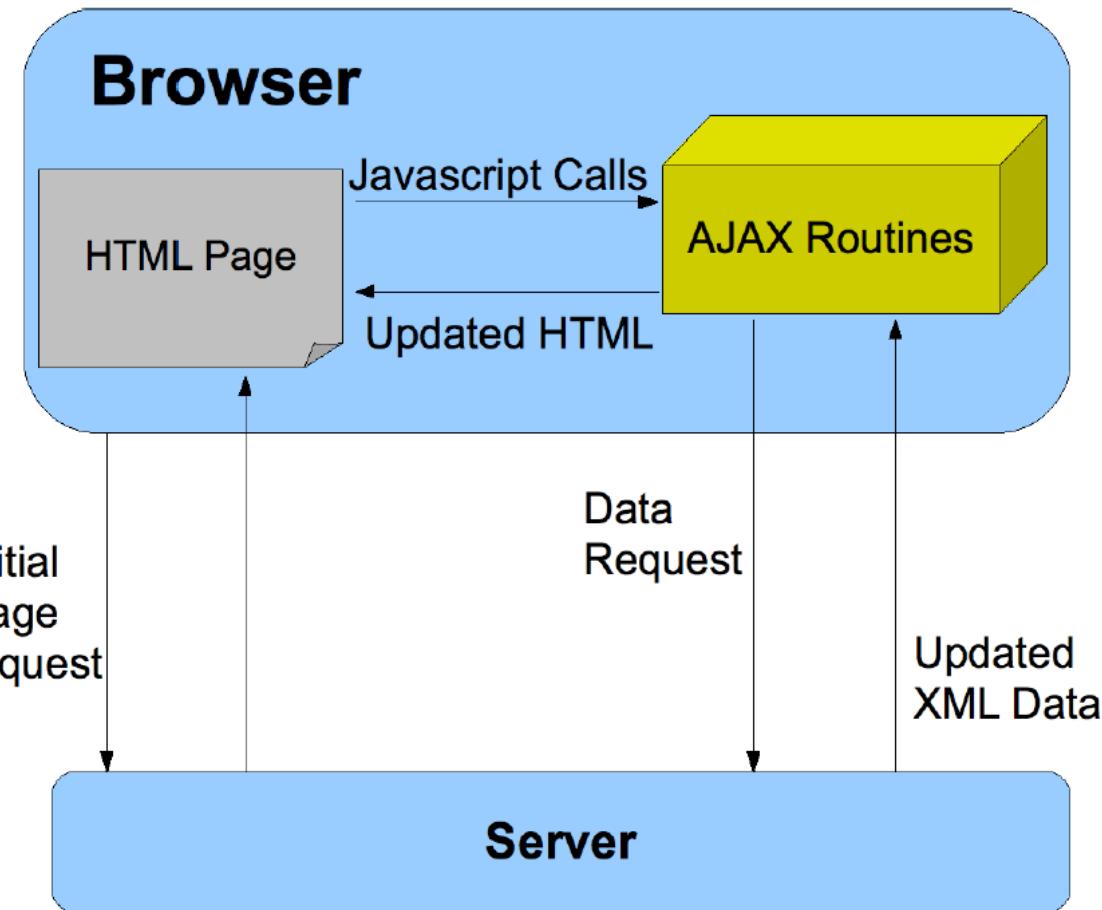
By reference e by value

```
var a = 10;  
(function(input) { input = 11; })(a);  
console.log(a); //ritorna 10, input è una copia di a
```

tipi primitivi

```
var a = {id: 10};  
(function(input) { input.id = 11; })(a);  
console.log(a.id);  
// ritorna 11, l'oggetto è passato per riferimento
```

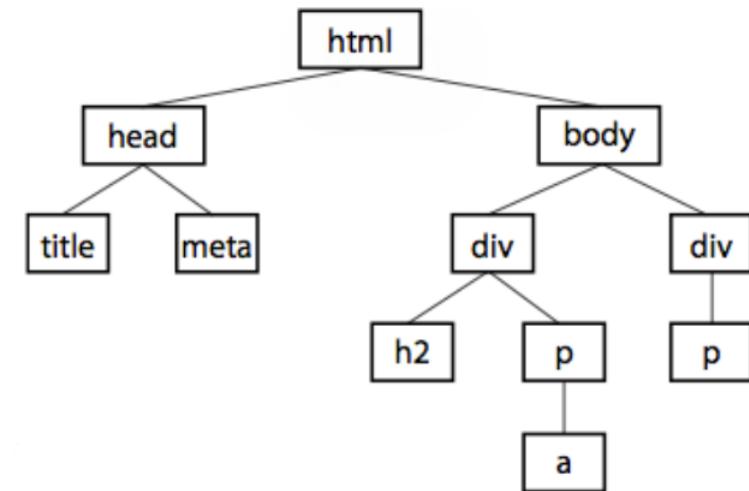
oggetti



- **Cambiare elementi e stili** in una pagina
 - Ad es: aggiungere classi css in risposta a eventi generati dall'utente (click, scroll ecc) -> DOM manipulation (more later)
- **Comunicazione asincrona**
 - Invio dati senza ricaricare la pagina o senza interazione con l'utente
 - Il paradigma è spesso chiamato AJAX
- **Altro**
 - Test browser capabilities ed adattamento (polyfills)
 - Concorrente APP mobile?
 - Cordova, Phonegap?

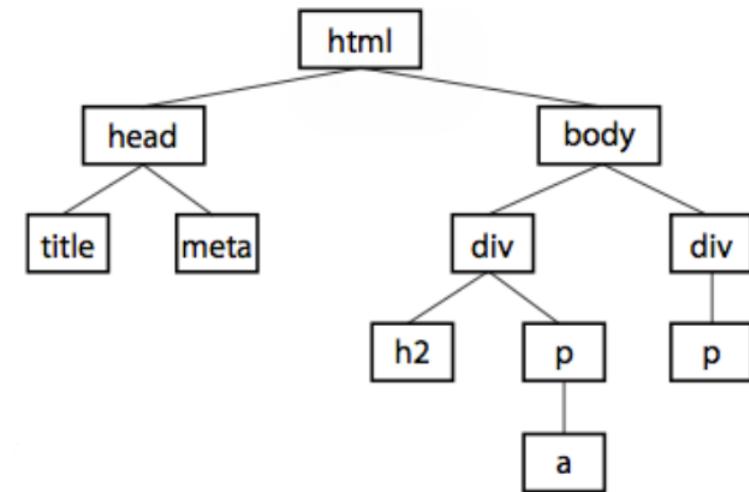
Document Object Model

- E' un'interfaccia di programmazione per HTML (e XML)
 - standard W3C
- Fornisce una mappa strutturata del nostro documento ed i metodi per interfacciarsi con gli elementi
 - Come fa un programma JS (ma anche PHP, Ruby, Java ecc) ad aggiungere una riga ad una tabella?
 - Non certo editando il testo html ...
 - Sarebbe comodo fare “tabella1.inserisciRiga(‘<td>Mia riga</td>’)”
- Ogni elemento della pagina è un nodo
- L'elemento radice è “document”
- “document” ha una serie di proprietà standard



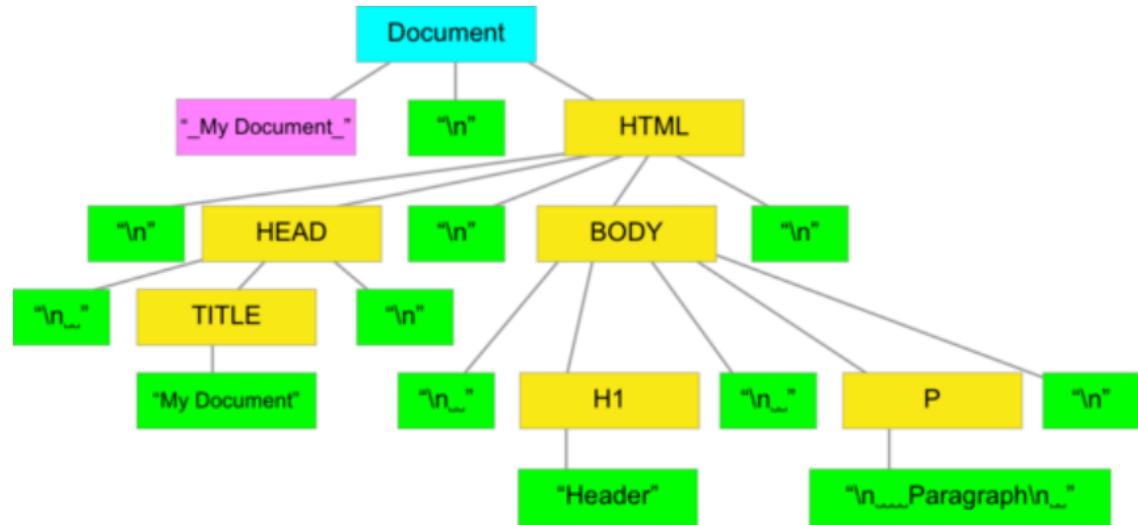
Document Object Model

- E' un'interfaccia di programmazione per HTML (e XML)
 - standard W3C
- Fornisce una mappa strutturata del nostro documento ed i metodi per interfacciarsi con gli elementi
 - Come fa un programma JS (ma anche PHP, Ruby, Java ecc) ad aggiungere una riga ad una tabella?
 - Non certo editando il testo html ...
 - Sarebbe comodo fare “tabella1.inserisciRiga(‘<td>Mia riga</td>’)”
- Ogni elemento della pagina è un nodo
- L'elemento radice è “document”
- “document” ha una serie di proprietà standard



Document Object Model

```
<!-- My document -->
<HTML>
<HEAD>
  <TITLE>My Document</TITLE>
</HEAD>
<BODY>
  <H1>Header</H1>
  <P>
    Paragraph
  </P>
  <P>
    Paragraph
  </P>
</BODY>
</HTML>
```



- **Node**: quanto serve per la rappresentazione dell'albero
- **Document**: deriva da node , ma alcune proprietà non sono applicabili (es: un Document non ha attributi).
- Browser:
 - **window** : finestra (tab) del browser contenente il documento
 - Attenzione: alcuni metodi sono sempre applicati alla finestra (es: `window.resizeTo`)
 - **navigator**: informazioni sul browser
 - **history**: scorrere in avanti/indietro la history
 - ...

Le specifiche DOM elaborate da W3C sono suddivise in livelli, ciascuno dei quali contiene moduli obbligatori o opzionali. Per sostenere di appartenere ad un certo 'livello', un'applicazione deve soddisfare tutti i requisiti di tale livello e dei livelli inferiori. La specifica attuale di DOM è al Livello 2, tuttavia alcune delle specifiche del Livello 3 ora sono già raccomandazioni del W3C.

- Livello 0
 - include tutto quello che viene fornito a DOM per la creazione del Livello 1, per esempio: `document.images`, `document.forms`, `document.layers`, e `document.all`. Nota, questa non è una specifica convenzionale pubblicata dal W3C ma piuttosto dà un riferimento a che cosa esisteva prima del processo di standardizzazione.
- Livello 1
 - navigazione di un documento DOM e manipolazione del contenuto.
- Livello 2
 - supporto al Namespace XML, viste filtrate e Eventi DOM.
- Livello 3
 - consiste in 6 specifiche differenti:
 - il nucleo del Livello 3;
 - caricamento e salvataggio del Livello 3;
 - XPath del Livello 3;
 - viste e formattazione del Livello 3;
 - requisiti del Livello 3;
 - validazione del Livello 3, che potenzia ulteriormente DOM.

Selezionare un elemento

- **document.getElementById("miodiv")**

- Ritorna il node associato al div con id “miodiv”

```
<!doctype HTML>
```

```
<html>
```

```
<head>
```

```
</head>
```

```
document.getElementById("prova").innerHTML = "ciao";
```

```
<body>
```

```
<div id='prova'></div>
```

```
</body>
```

```
</html>
```

- **document.getElementsByTagName("p")**

- Ritorna una NodeList degli elementi “p”

- **document.getElementsByClassName("myclass")**

- Ritorna una NodeList degli elementi “p”

- **document.querySelectorAll("p .warning")**

- Permette di usare selettori css e ritorna una NodeList

Una **NodeList** è simile a un array di nodi dell’albero DOM.

Esempio:

```
var paragraphs = document.getElementsByTagName("p");
paragraphs[0]; paragraphs.length
```

Esempio

- scriviamo CIAO dentro un div quando premo un pulsante

```
<!doctype HTML>
<html>
  <head>
  </head>
  <body>
    <div id="prova"></div>
    <button onclick="document.getElementById('prova').innerHTML = 'CIAO'; ">
      Premi qua
    </button>
  </body>
</html>
```

Attributo: "onclick"
Valore: codice javascript

Esempio

- Spesso ci è utile raggruppare più righe di codice in una funzione

```
<!doctype HTML>
<html>
  <head>
  </head>
  <body>
    <div id="prova"></div>
    <button onclick="scriviCiao()">
      Premi qua
    </button>
    <script type="text/javascript">
      function scriviCiao() {
        document.getElementById('prova').innerHTML = 'CIAO';
      }
    </script >
  </body>
</html>
```

```
<!doctype HTML>
<html>
  <head>
  </head>
  <body>
    <div id="prova"></div>
    <button onclick="scriviCiao()">
      Premi qua
    </button>
    <script type="text/javascript">
      function scriviCiao() {
        var provadiv = document.getElementById('prova');
        provadiv.innerHTML = 'CIAO';
      }
    </script >
  </body>
</html>
```

Eventi

onblur/onfocus	Un elemento prende/perde il focus
onchange	Il contenuto di un form cambia
onclick	Mouse click
onerror	Quando c'e' un errore nel caricamento di immagini o del documento
onload	La pagina ha finito di caricarsi
onkeydown/onkeypress/onkeyup	Un tasto viene premuto/tenuto premuto/rilasciato
onmousedown/onmouseup	un bottone del mouse è premuto/rilasciato
onmousemove/onmouseout/onmouseover	Il mouse si è spostato/spostato fuori da un elemento/spostato sopra un elemento
onsubmit	E' stato premuto il pulsante submit di un form

Eventi javascript.....

Abort
Blur
Change
Click
DblClick
DragDrop
Error
Focus
KeyDown
KeyPress
KeyUp
Load
MouseDown
MouseMove
MouseOver
MouseOut
MouseUp
Move
Reset
Resize
Select
Submit
Unload

1. con un **attributo HTML**

```
<body onclick="myFunction();">
```

2. con un **metodo DOM**

```
window.onclick = myFunction;  
document.getElementById("miodiv").onclick = ...
```

3. con **addEventListener**

```
window.addEventListener("click", myFunction);
```

Eventi-attributi del tag:

abort-onAbort; blur-onBlur; change-onChange; click-onClick; error-onError; focus-onFocus; load-onLoad; mouseout-onMouseOut; mouseover-onMouseOver; reset-onReset; resize-onResize; select-onSelect; submit-onSubmit; unload-onUnload

Manipolare un nodo DOM

```
var myImage = document.getElementById("someimage");
```

```
var oldSrc = myImage.getAttribute('src');
```

Leggere attributi
di un nodo

```
myImage.setAttribute('src', 'otherimage.jpg');
```

Scrivere attributi
di un nodo

```
var myP = document.getElementById("someparagraph");
```

```
myP.innerHTML = "<p>New text</p>";
```

Leggere/Modificare HTML

```
myP.style.color = '#fff';  
myP.style.backgroundColor = '#fff'
```

Leggere/Modificare CSS

in JS le proprietà CSS si scrivono aGobbaDiCammello

Creare un nodo DOM

```
// crea un nodo ma non lo visualizza (non gli  
abbiamo detto dove metterlo)  
var newDiv = document.createElement("div");  
  
// come il precedente ma crea un nodo  
// contenente del testo  
var ourText = document.createTextNode("Ciao!");  
  
// mettiamoli dentro un elemento della pagina  
var ourDiv = document.getElementById("mydiv");  
newDiv.appendChild(ourText);  
ourDiv.appendChild(newDiv)
```

ourDiv.insertBefore(newHeading, para);

Inserisci il nodo “newHeading” prima del nodo “para”. Entrambi sono contenuti nel nodo genitore “ourDiv”.

ourDiv.replaceChild(newImg, oldImg);

Sostituisce newImg al posto di oldImg

parentDiv.removeChild(removeMe);

Rimuove il nodo removeMe dal suo genitore che è il nodo parentDiv

Un input di tipo text può acquisire il focus in tre modi:

- quando l'utente passa il mouse sopra e preme il pulsante sinistro
- oppure, tramite il tasto di tabulazione
- o eseguendo il metodo focus.

Il gestore di eventi può essere agganciato in due modi,

- assegnando al tag gli attributi:
`onClick="alert('Mouse cliccked');"`...
`onClick="myHandler();"`...
- Oppure tramite notazione DOT:
`document.myForm.elements[0].onClick=myHandler;`

Da notare che nel secondo non possono essere passati parametri formali, anche se in questo modo il gestore di eventi può essere cambiato a runtime.

Per i radio button, l'oggetto risulta cliccato ad ogni pressione del mouse.

Per capire quale radio button è checked, non possiamo usare il suo nome perché è uguale a tutti i nomi degli altri radio button del suo gruppo, ma dobbiamo navigare l'array di elementi:

```
var radioElements =document.myForm.elements;  
for(var i=0; i<radioElements;i++){  
    if(radioElement[i].checked){  
        element=radioElement[i].value;  
        break;  
    }  
}
```

Non include le caratteristiche di DOM 0, anche se sono ancora supportate.

Propagazione dell'evento in 3 fasi

Si definisce il nodo dell'albero del documento da cui proviene l'evento, chiamato anche target node.

La prima fase è chiamare il **capturing** phase, ovvero, l'evento inizia dalla root e si muove fino al target node. Se ci sono altri gestori di eventi nel percorso tra la root ed il nodo target, saranno eseguiti.

La seconda fase riguarda il **target node**, per cui se è registrato un **gestore** questo sarà eseguito.

La terza fase è la **bubbling** phase, ovvero, l'evento ritorna alla root (si propaga a ritroso nell'albero DOM) senza rieseguire i gestori.

Non tutti gli eventi ritornano, un gestore può fermare la propagazione chiamando il metodo **stopPropagation()** dell'oggetto Event.

DOM 2 usa il metodo dell'oggetto Event, **preventDefault()** per fermare le operazioni di default, come ad esempio il submit di un form, per consentire di controllare l'output prima di inviarlo.

La registrazione di un gestore di eventi, anche temporaneo, è fatta attraverso il metodo:

```
addEventListener(“nomeDellevento”, funzioneDiGestione, booleano);
```

dove il valore boolean specifica se l'evento è abilitato durante la fase di cattura.

La proprietà currentTarget specifica il nodo che sta eseguendo il gestore di eventi.

MouseEvent (un sottooggetto di Event), ha due proprietà, clientX e clientY per specificare le coordinate del mouse relativa all'angolo sinistro in alto della finestra.

Esempio: validare un campo di un FROM

```
1  <!DOCTYPE html PUBLIC "-//w3c//DTD XHTML 1.0 Strict//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3
4  <!-- validator2.html
5   An example of input validation using the change and submit
6   events, using the DOM 2 event model
7   Note: This document does not work with IE6
8  -->
9  <html>
10 <head>
11 <title> Illustrate form input validation with DOM 2</title>
12 <script type = "text/javascript">
13 <!--
14 // ****
15 // -----
16 // -----
17 // -----
18 // -----
19 // -----
20 // -----
21 // -----
22 // -----
23 // -----
24 // -----
25 // -----
26 // -----
27 // -----
28 // -----
29 // -----
30 // -----
31 // -----
32 // -----
33 // -----
34 // -----
35 // -----
36 // -----
37 // -----
38 // -----
39 // -----
40 // -----
41 // -----
42 // -----
43 // -----
44 // -----
45 // -----
46 // -----
47 // -----
48 // -----
49 // -----
50 // -----
51 // -----
52 // -----
53 // -----
54 // -----
55 // -----
56 // -----
57 // -----
58 // -----
59 // -----
60 // -----
61 // -----
62 // -----
63 // -----
64 // -----
65   </script>
66   </head>
67
68 <body>
69
70 <h3> Customer Information </h3>
71 <form action = "">
72   <p>
73     <input type = "text" id = "custName" />
74     Name (last name, first name, middle initial)
75     <br /><br />
76
77     <input type = "text" id = "phone" />
78     Phone number (ddd-ddd-dddd)
79     <br /><br />
80
81     <input type = "reset" />
82
83     <input type = "submit" />
84   </p>
85 </form>
86 <script type = "text/javascript">
87 <!--
88
89 /* Get the DOM addresses of the elements and register
90   the event handlers */
91
92 var customerNode = document.getElementById("custName");
93 var phoneNode = document.getElementById("phone");
94 customerNode.addEventListener("change", chkName, false);
95 phoneNode.addEventListener("change", chkPhone, false);
96
97 // -->
98 </script>
99 </body>
100 </html>
```

Esempio: validare un campo di un FROM

```
18 //function chkName(event) {
19 // Get the target node of the event
20
21 var myName = event.currentTarget;
22
23 // Test the format of the input name
24 // Allow the spaces after the commas to be optional
25 // Allow the period after the initial to be optional
26
27
28 var pos = myName.value.search(/\w+, ?\w+, ?\w.?/);
29
30 if (pos != 0) {
31     alert("The name you entered (" + myName.value +
32           ") is not in the correct form. \n" +
33           "The correct form is: " +
34           "last-name, first-name, middle-initial \n" +
35           "Please go back and fix your name");
36     myName.focus();
37     myName.select();
38 }
39
40
41 // ****
42 // The event handler function for the phone number text box
43
44 function chkPhone(event) {
45
46     // Get the target node of the event
47
48     var myPhone = event.currentTarget;
49
50     // Test the format of the input phone number
51
52     var pos = myPhone.value.search(/^\d{3}-\d{3}-\d{4}$/);
53
54 if (pos != 0) {
55     alert("The phone number you entered (" + myPhone.value +
56           ") is not in the correct form. \n" +
57           "The correct form is: ddd-ddd-dddd \n" +
58           "Please go back and fix your phone number");
59     myPhone.focus();
60     myPhone.select();
61 }
62 }
```

Esempio: validare un campo di un FROM (2 – in pratica un solo listener)

```
11 <script type = "text/javascript">
12 <!--
13 // ****
14 // The event handler function for the name text box
15
16 function chkName() {
17     var myName = document.getElementById("custName");
18
19 // Test the format of the input name
20 // Allow the spaces after the commas to be optional
21 // Allow the period after the initial to be optional
22
23     var pos = myName.value.search(/\w+, ?\w+, ?\w\.\w/);
24
25     if (pos != 0) {
26         alert("The name you entered (" + myName.value +
27               ") is not in the correct form. \n" +
28               "The correct form is: " +
29               "last-name, first-name, middle-initial \n" +
30               "Please go back and fix your name");
31         myName.focus();
32         myName.select();
33         return false;
34     } else
35         return true;
36 }
37
38 // ****
39 // The event handler function for the phone number text box
40
41 function chkPhone() {
42     var myPhone = document.getElementById("phone");
43
44 // Test the format of the input phone number
45
46     var pos = myPhone.value.search(/^\d{3}-\d{3}-\d{4}$/);
47
48     if (pos != 0) {
49         alert("The phone number you entered (" + myPhone.value +
50               ") is not in the correct form. \n" +
51               "The correct form is: ddd-ddd-dddd \n" +
52               "Please go back and fix your phone number");
53         myPhone.focus();
54         myPhone.select();
55         return false;
56     } else
57         return true;
58 }
```

```
81 </form>
82 <script type = "text/javascript">
83 <!--
84 // Set form element object properties to their
85 // corresponding event handler functions
86
87 document.getElementById("custName").onchange = chkName;
88 document.getElementById("phone").onchange = chkPhone;
89 // -->
90 </script>
91 </body>
92 </html>
```

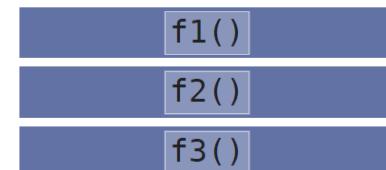
Javascript asincrono

JavaScript is a **single-threaded, non-blocking, asynchronous, concurrent** programming language with lots of flexibility.

JavaScript mantiene uno stack in memoria chiamato **function execution stack** che tiene traccia della funzione in esecuzione

- Quando una funzione viene invocata, viene aggiunta al function execution stack
- Se la funzione invoca un'altra funzione, quest'ultima viene aggiunta a stack ed eseguita
- Quando una funzione termina viene rimossa dallo stack e il controllo passa alla funzione precedente
- Si continua così finchè non ci sono più funzioni nello stack

```
1 function f1() {  
2   ...  
3 }  
4 function f2() {  
5   f1();  
6 }  
7 function f3() {  
8   f2();  
9 }  
10 f3();
```



Function Execution
Stack

Alcune funzioni possono terminare "più tardi": posso avere una funzione con un ritardo impostato, oppure il risultato dipende da qualcun altro (dati da un server, query a database, ...).

In queste circostanze non voglio bloccare l'esecuzione del codice, ma continuare con le altre funzioni.

Ci sono due tipi principali di operazioni che vengono eseguite in modo asincrono (non bloccanti):

- Browser API/Web API: **eventi sul DOM** come click, scroll e simili e metodi come setTimeout()
- **Promise**: oggetti che permettono di eseguire operazioni asincrone

Javascript asincrono: browser PAI/web API

```
function stampami() { /* callback */
  console.log('stampa me');
}

function test() {
  console.log('test');
}

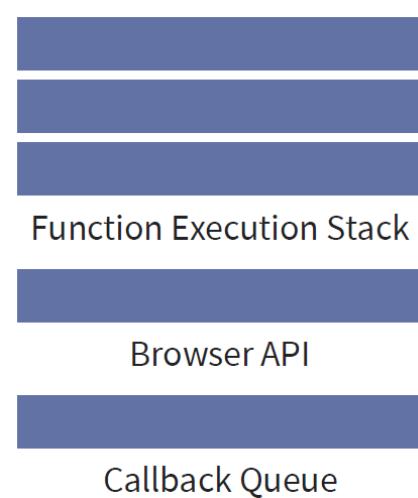
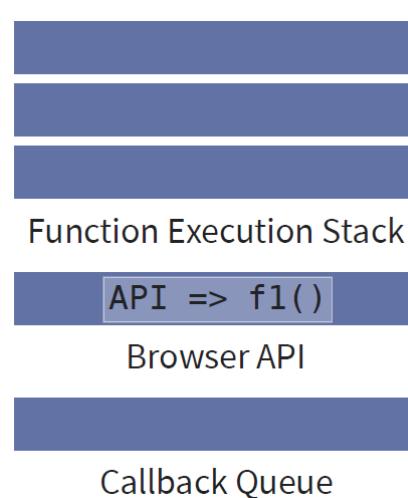
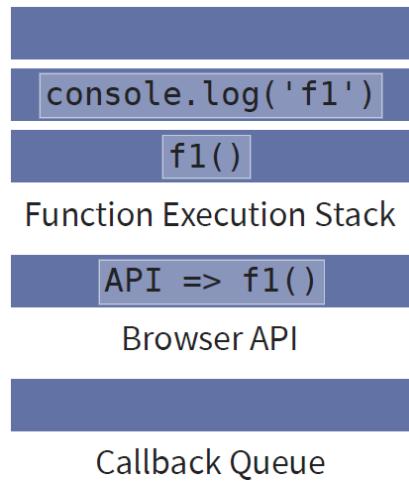
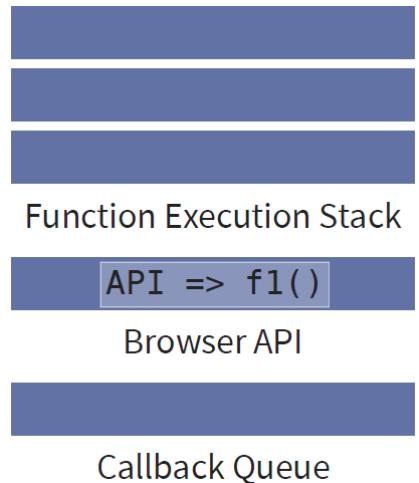
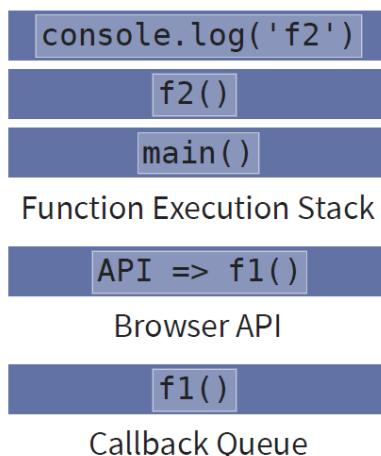
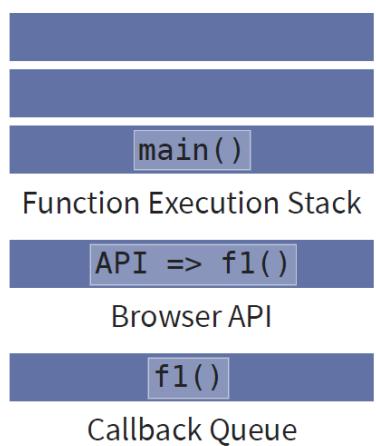
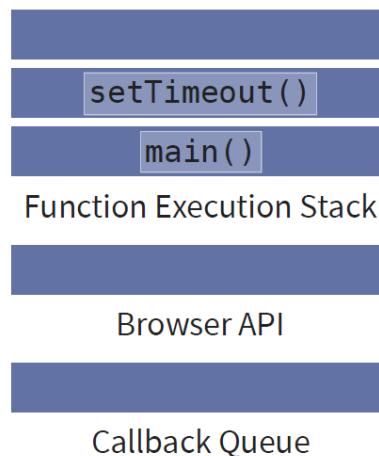
setTimeout(stampami, 2000);
test();
```

In che ordine vengono stampate le due frasi?

- Vengono attesi 2 secondi e poi esegue stampami() e quindi test() ?
- Oppure viene eseguita test() e dopo 2 secondi stampami() ?
- JavaScript ha una coda dedicata per le **callback** (Callback Queue)
- Viene creato un ciclo che periodicamente controlla la coda e sposta le callback nello stack (Event Loop):
 - Le funzioni nello stack vengono eseguite normalmente
 - Se viene invocata una API del browser, si aggiunge una callback nella coda (macrotask)
 - Se lo stack è vuoto, viene spostata la callback dalla coda allo stack
 - Ricomincia da capo

Javascript asincrono: browser PAI/web API

```
function f1() {  
    console.log('f1');  
}  
  
function f2() {  
    console.log('f2');  
}  
  
function main() {  
    console.log('main');  
    setTimeout(f1, 0);  
    f2();  
}  
main();
```



Promise

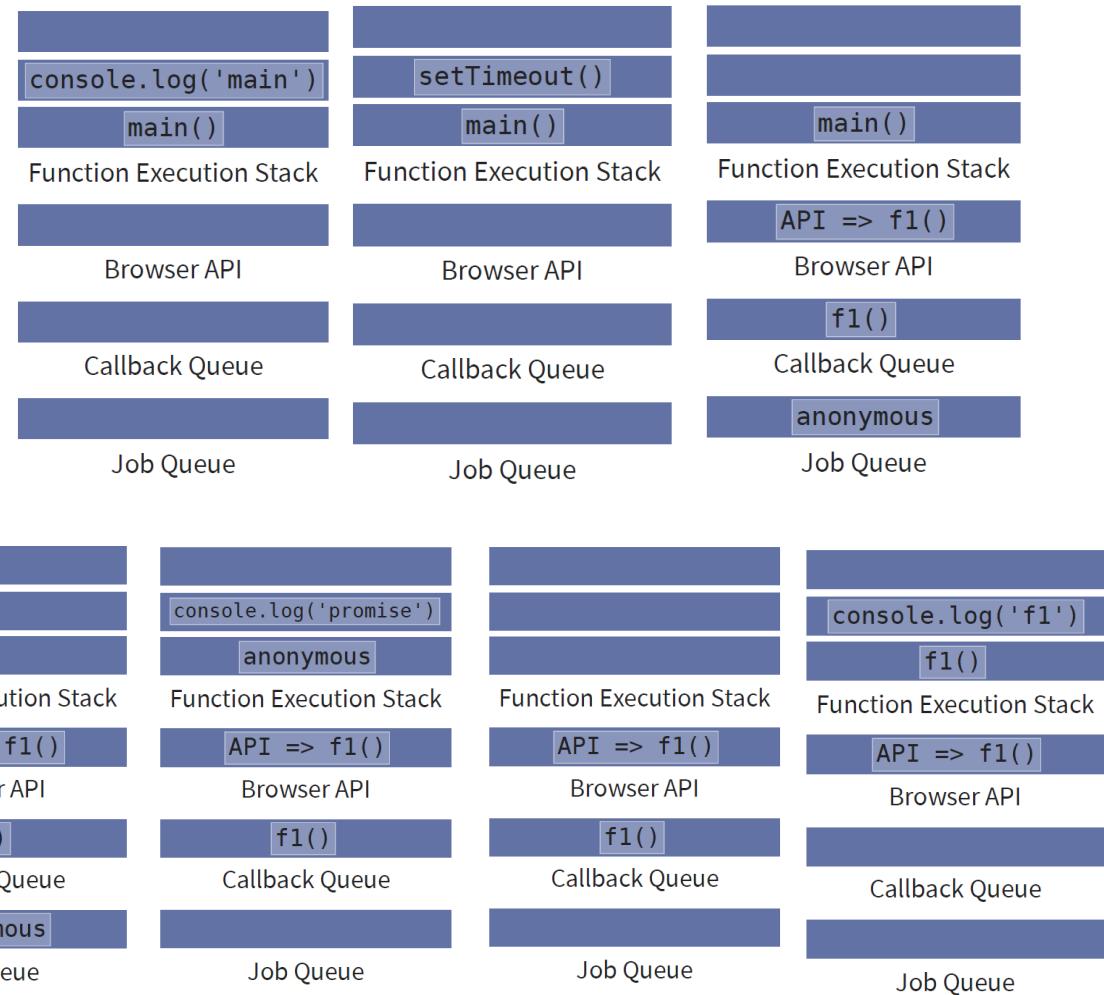
- Le Promise sono oggetti che permettono di eseguire **codice asincrono**
- Per gestire le Promise viene usata una coda separata (**Job Queue**)
- Una funzione nella Job Queue si chiama **microtask**
- Se ci sono elementi sia nella Callback Queue che nella Job Queue, l'Event Loop esegue prima quelli nella Job Queue (microtask)

```
function f1() {
  console.log('f1');
}

function f2() {
  console.log('f2');
}

function main() {
  console.log('main');
  setTimeout(f1, 0);
  new Promise((resolve, reject) =>
    resolve('promise')
  ).then(resolve => console.log(resolve))
  f2();
}

main();
```



Funzioni async

- Con ECMAScript 2017 è stato aggiunto il supporto per scrivere funzioni asincrone, un altro modo per gestire le Promise.
- Una funzione asincrona è una funzione che:
 - È stata dichiarata con la parola chiave **async**
 - Permette l'uso della parola chiave **await** al suo interno
 - Restituisce una **Promise**

```
1 async function hello() { return "Ciao!" };  
2 hello(); /* ritorna una Promise */
```

Si può scrivere in forma più elegante

```
let hello = async function() { return "Ciao!" };
```

Oppure con una arrow function

```
let hello = async () => "Ciao!";
```

Per consumare la Promise posso usare `then()`

```
hello().then((value) => console.log(value));
```

Oppure la forma abbreviata

```
hello().then(console.log);
```

Funzioni async: await

- I vantaggi delle funzione asincrone si vedono quando combinate con la parola chiave await :
 - può essere messa davanti ad una Promise
 - può essere messa davanti ad una funzione asincrona
 - può essere usata soltanto dentro una funzione asincrona
 - mette in pausa il codice finchè la Promise non viene valorizzata,
 - quindi restituisce il suo valore di ritorno

```
async function hello() {
  return await Promise.resolve("Hello");
}

hello().then(console.log);
```

```
function dopo2sec() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('Fatto!');
    }, 2000);
  });
}

async function asincrona() {
  console.log('Eseguo e aspetto');
  const result = await dopo2sec();
  console.log(result);
}
asincrona();
```

Con le promise:

```
1 setTimeoutPromise(-10).then(msg => {
2   console.log(msg);
3 }).catch(error => {
4   console.error(error)
5 }).finally(() => {
6   console.log("Esegui comunque!");
7 }) ;
```

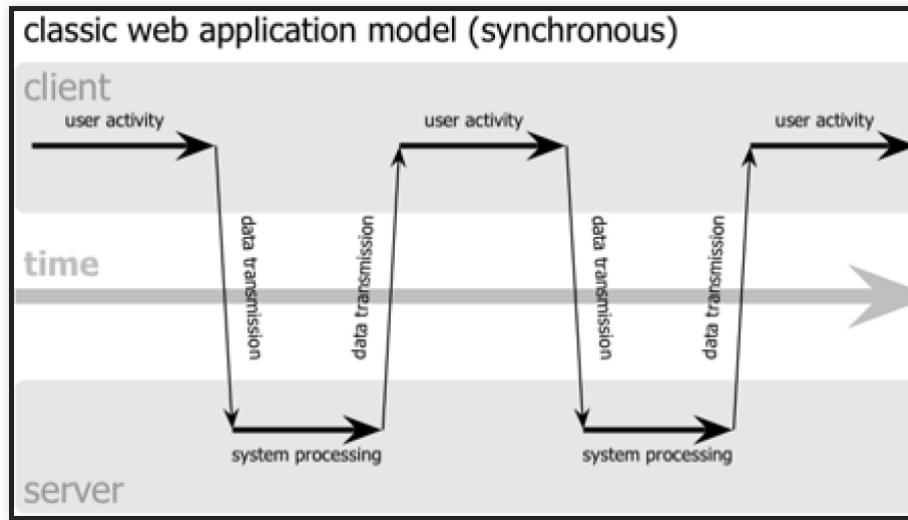
Con codice asincrono:

```
1 asincrona();
2 async function asincrona() {
3   try {
4     const msg = await setTimeoutPromise(-10);
5     console.log(msg);
6   } catch (error) {
7     console.error(error);
8   } finally {
9     console.log("Esegui comunque!");
10  }
11 }
```

Web classico

Quando si inserisce un URL nel browser:

- il computer risolve l'indirizzo IP usando il DNS
- il browser si connette a quell'IP e richiede il file specificato
- il web server (es: Apache) accede al file system e restituisce il contenuto



- Cambiare i dati === ricaricare la pagina
 - ogni richiesta blocca l'interfaccia grafica fino al termine
 - devo ritrasmettere sempre tutto
 - user experience scomoda: perdo il riferimento visivo
 - validazione spesso solo all'invio (es: username già esistente)
- Ma noi vogliamo fare Web Applications, cioè un sito internet che sia molto simile nell'uso ad un'applicazione desktop.

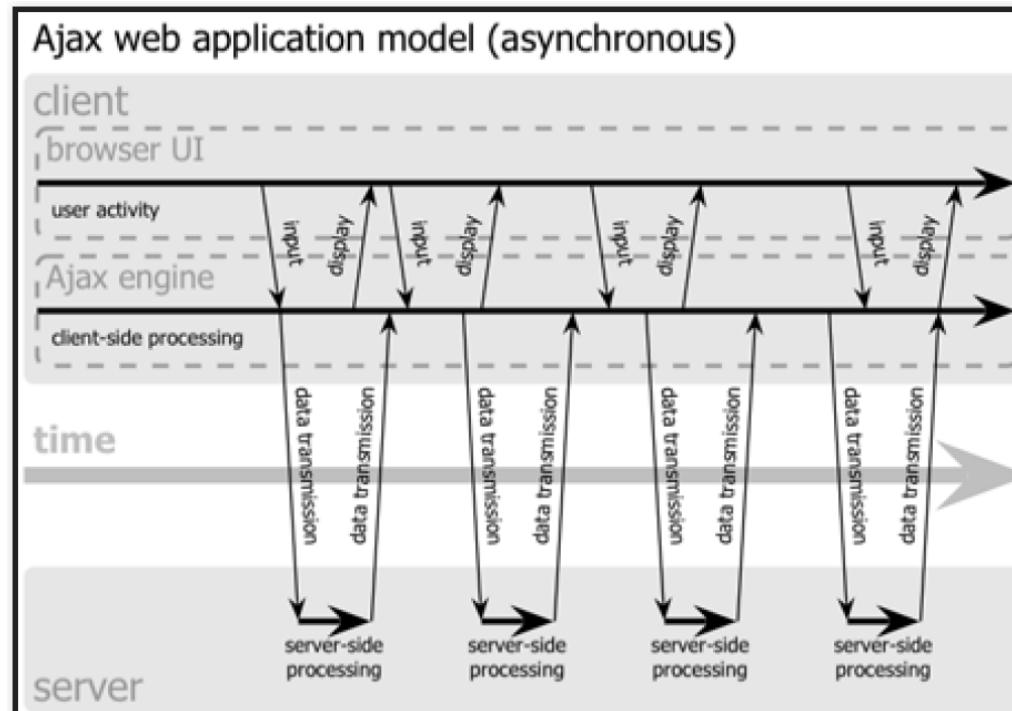
Web interattivo (asincrono)

Asynchronous JavaScript + XML: un modo di utilizzare JavaScript per arricchire un documento HTML con nuove informazioni (senza riscaricarlo)

Ajax: è la chiave dell'interattività delle applicazioni Web!

- Asynchronous
- JavaScript
- XML → le nuove informazioni hanno questo formato

In pratica: Ajax = utilizzare l'oggetto JavaScript **XMLHttpRequest**



XMLHttpRequest

The XMLHttpRequest specification defines an API that provides scripted client functionality for transferring data between a client and a server.

Non è un linguaggio di programmazione, ma un modo di usare JavaScript.

Specifica disponibile su, <http://www.w3.org/TR/XMLHttpRequest>
documentazione su w3schools

Come usarlo

1. preparare l'oggetto
2. inviare la richiesta
3. quando arriva la risposta, utilizzarla

"Utilizzarla" ≈ leggerla e aggiornare il DOM con le nuove informazioni

XMLHttpRequest

```
1 var xhr = new XMLHttpRequest();
2 xhr.onreadystatechange = myFunction;
3 xhr.open("GET", url, true);
4 xhr.send(null);
```

Bisogna dire all'oggetto XMLHttpRequest :

- url → chi contattare
 - myFunction → cosa fare quando riceve la risposta
 - il parametro booleano nell' open specifica che la richiesta è asincrona
-
- **SAME ORIGIN POLICY:** Restrizione che impedisce ad uno script di comunicare con server diversi da quello da cui origina il documento che lo ospita
 - Lo script su www.trieste.it/index.html non può fare xhr.open("GET", "http://udine.it", true)

Quando arriva la risposta:

- xhr.onreadystatechange → handler, funzione invocata ad ogni cambio di stato
- xhr.readyState → stato dell'interazione (tra oggetto e server):
 - 0 → oggetto appena costruito (dopo new ...)
 - 1 → pronto (dopo open())
 - 2 → headers risposta ricevuti (dopo send())
 - 3 → loading (si sta scaricando il payload della risposta)
 - 4 → risposta ricevuta (o errore)
 - Di solito ci interessa solo il 4.

XMLHttpRequest

- Quando è disponibile (`readyState == 4`) e non ci sono stati errori, la risposta si trova in:
 - `xhr.responseText` → corpo della risposta, come stringa
 - `xhr.responseXML` → corpo della risposta, come oggetto di tipo Document, solo se:
 - il corpo della risposta HTTP era un documento XML valido
 - il MIME type ricevuto è null o di tipo XML (`text/xml`, `application/xml`, ...)

```
1 xhr.onreadystatechange = function() {  
2     var text;  
3     if (xhr.readyState == 4) {  
4         if (xhr.status == 200) {  
5             text = xhr.responseText;  
6             /* usare text nel DOM */  
7         } else {  
8             /* gestire l'errore; */  
9         }  
10    }  
11};
```

- il server restituisce solo testo, o un frammento di HTML:

```
document.getElementById("toRewrite").innerHTML = xhr.responseText;
```

- il server restituisce testo formattato (es: `uno`, `due`, `tre`):

```
1 var listEl = document.getElementById("list");  
2 var pieces = xhr.responseText.split(",");  
3 var i, newLi;  
4 for (i = 0; i < pieces.length; i++) {  
5     newLi = document.createElement("li");  
6     newLi.innerHTML = pieces[i];  
7     listEl.appendChild(newLi);  
8 }
```

XMLHttpRequest e risposte in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<animals>
  <animal type="dog">
    <name>Simba</name>
    <color>white</color>
  </animal>
  <animal type="dog">
    <name>Gass</name>
    <color>brown</color>
  </animal>
</animals>
```

```
1 var xmlDoc = xhr.responseXML;
2 var animals = xmlDoc.getElementsByTagName("animal");
3 var i, name, nameEl;
4 for (var i = 0; i < animals.length; i++) {
5   var nameEl = animals[i].getElementsByTagName("name") [0];
6   var name = nameEl.childNodes[0].nodeValue;
7   ...
8 }
```

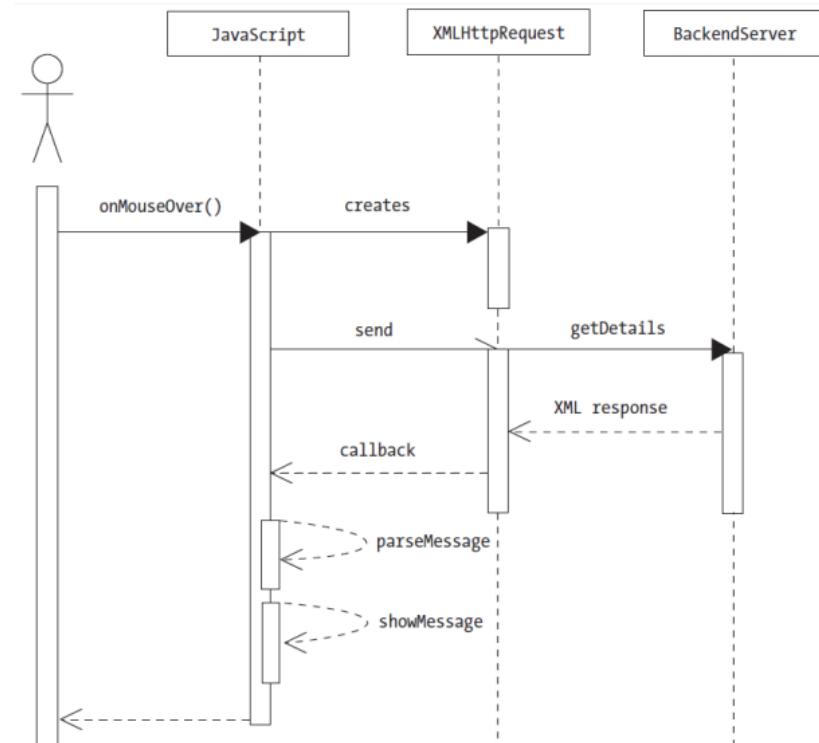
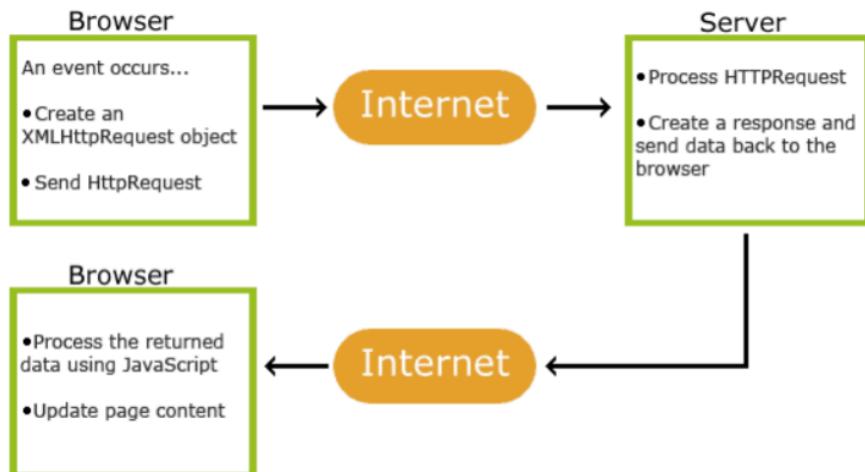
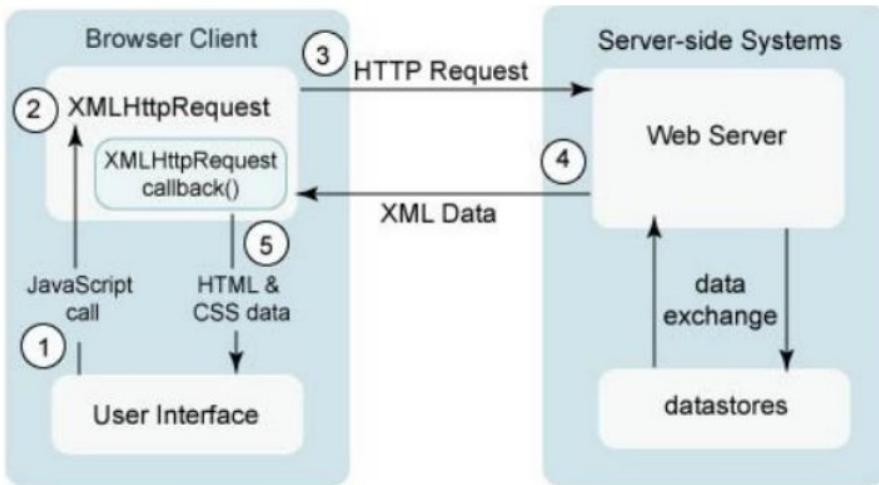
Nota: è possibile [convertire l'XML in un oggetto](#).

XMLHttpRequest e risposte in JSON

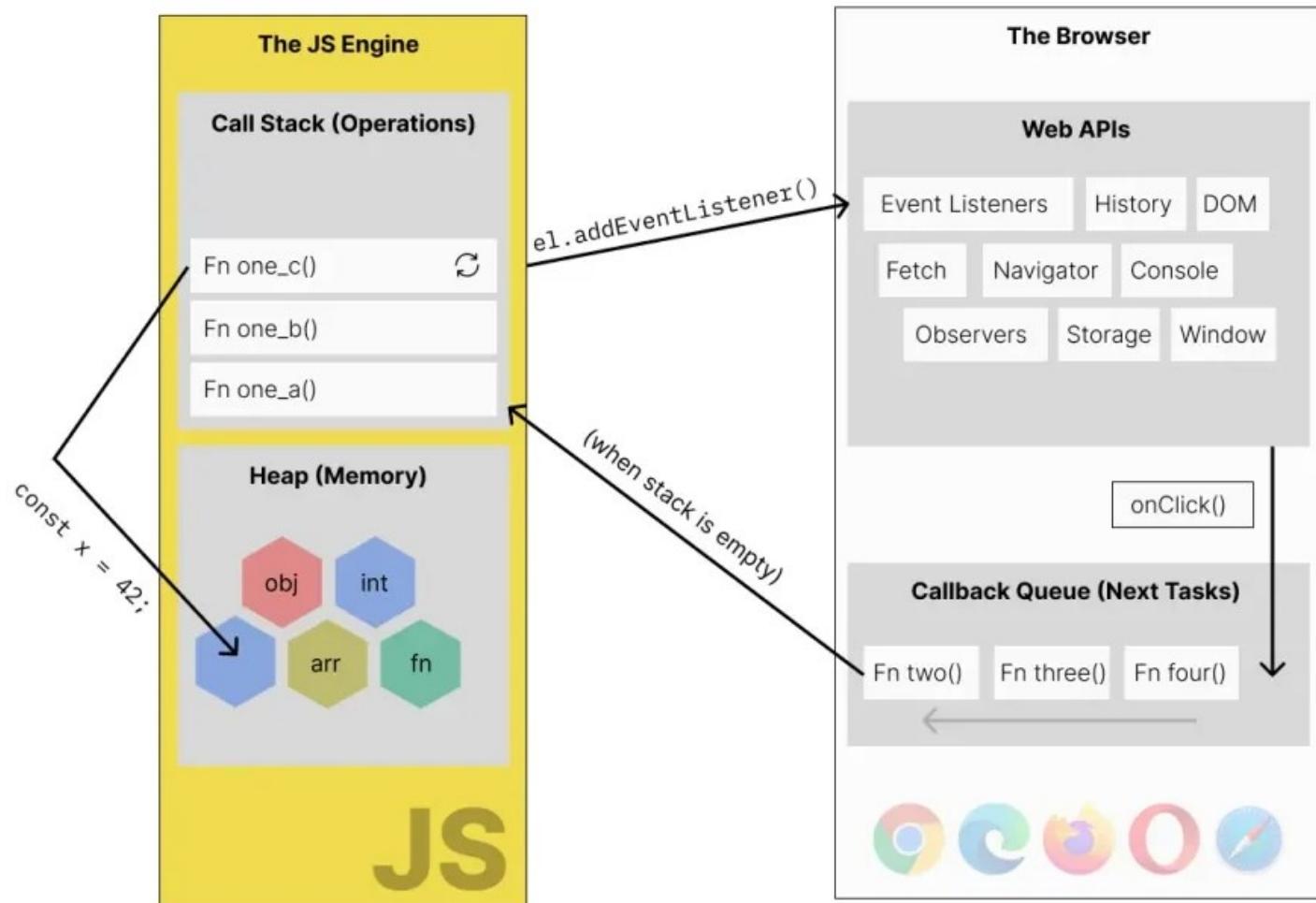
```
{ "employees": [
    {"firstName":"Mick","lastName":"Jagger" },
    {"firstName":"Ronnie","lastName":"Wood" },
    {"firstName":"Keith","lastName":"Richards" },
    {"firstName":"Charlie","lastName":"Watts" } ] }
```

```
1 xhr.onreadystatechange = function() {
2   var obj;
3   if (xhr.readyState == 4) {
4     if (xhr.status == 200) {
5       obj = JSON.parse(xhr.responseText);
6       /*usare obj nel DOM*/
7     } else {
8       /*gestire l'errore;*/
9     }
10   }
11 };
```

XMLHttpRequest e come descriverlo



Gestione degli eventi e stack



The Javascript event loop

XMLHttpRequest e IE

Per gestire tutti i browser bisogna controllare se il browser supporta l'oggetto XMLHttpRequest :

```
var xmlhttp;
if (window.XMLHttpRequest)
    {// IE7+, Firefox, Chrome, Opera, Safari
        xmlhttp = new XMLHttpRequest();
    }
else
    {// IE6, IE5
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
```

XMLHttpRequest e fetch()

L'uso di XMLHttpRequest è scomodo:

- pensato nel 2006, vecchie versioni di JavaScript
 - è codice asincrono: potrebbe e usare Promise o async
 - → funzione fetch introdotta di recente
-
- Ispirato alla funzione ajax() di JQuery, ma:
 - la promise non è rigettata se c'è uno status HTTP di errore
 - di default non manda cookie ad altri url
 - usa gli oggetti Response e Request

```
1  fetch('url')
2    .then(response => response.json())
3    .then(data => console.log(data));
```

```
1 async function postData(url = '', data = {}) {
2   const response = await fetch(url, {
3     method: 'POST',
4     headers: {
5       'Content-Type': 'application/json'
6     },
7     body: JSON.stringify(data)
8   });
9   return response.json();
10 }
11
12 postData('https://esempio', { dati: 'xxx' })
13   .then(data => {
14     console.log(data);
15   });

```

jQuery (<http://jquery.com/>) è una libreria di classi e funzioni Javascript che permette al programmatore di compiere in modo facile quanto segue:

- Navigazione nel documento HTML,
- gestione degli eventi,
- animazioni,
- funzionalità AJAX.

- Consente di scrivere codice in modo più compatto e ad alto livello.
- Funge da “normalizzatore”, fornendo dei costrutti conformi agli standard del W3C, indipendentemente dal browser su cui gira.
- Scaricare la libreria da
http://docs.jquery.com/Downloading_jQuery
- Inclusione remota:
`<script type="text/javascript" src="http://.../jquery.min.js"></script>`
- Inclusione locale (supponendo di aver scaricato il file in una sottodirectory **js**):
`<script type="text/javascript" src="js/.../xxx.js"></script>`

jQuery (<http://jquery.com/>) è una libreria di classi e funzioni Javascript che permette al programmatore di compiere in modo facile quanto segue:

- Navigazione nel documento HTML,
- gestione degli eventi,
- animazioni,
- funzionalità AJAX.

- Consente di scrivere codice in modo più compatto e ad alto livello.
- Funge da “normalizzatore”, fornendo dei costrutti conformi agli standard del W3C, indipendentemente dal browser su cui gira.
- Scaricare la libreria da
http://docs.jquery.com/Downloading_jQuery
- Inclusione remota:
`<script type="text/javascript" src="http://.../jquery.min.js"></script>`
- Inclusione locale (supponendo di aver scaricato il file in una sottodirectory **js**):
`<script type="text/javascript" src="js/.../xxx.js"></script>`

L'oggetto principale JQuery

- In Javascript una variabile è identificata tramite una sequenza di lettere (a-z,A-Z), cifre (0-9), il carattere \$ ed il carattere _ e deve iniziare con una lettera, \$ o _.
- Quindi \$ è un identificatore lecito per una variabile in Javascript.
- jQuery usa proprio il \$ per identificare il suo oggetto principale.
- In particolare abbiamo:
`window.jQuery = window.$ = jQuery = function(selector, context);`

Selezionare gli elementi del DOM

- In jQuery quasi tutto ruota attorno ad un potente motore di selezione degli elementi del DOM, che fa uso della stessa sintassi dei CSS.
- In particolare abbiamo a disposizione:
 - Selettori di base,
 - Selettori gerarchici,
 - Filtri:
 - Filtri di contenuto,
 - Filtri di visibilità,
 - Filtri di attributi,
 - Filtri di nodi figli,
 - Filtri di form.

Selettori di base

- Selezione degli elementi mediante **tipo del tag**:

```
$(“p”); // seleziona tutti i paragrafi <p>
```

- Selezione mediante il **nome della classe**:

```
$(“.foo”); /* seleziona tutti gli elementi con classe foo */
```

- Selezione mediante **id**:

```
$(“#bar”); /* seleziona tutti gli elementi con id bar */
```

- Selezione **combinata**:

```
$(“p.foo”); /* seleziona solo i paragrafi con classe foo */
```

```
$(“p.foo,#bar”); /* seleziona gli elementi che soddisfano almeno uno dei due selettori */
```

Selettori gerarchici

- Per selezionare gli elementi **discendenti** di un tag è sufficiente far seguire a quest'ultimo il nome del tag discendente dopo uno spazio:

```
 $("body span"); /* seleziona tutti i tag span contenuti all'interno del tag body. */
```

- Selezione dei **figli**:

```
 $("body>span"); /* seleziona soltanto i tag span figli (i.e., discendenti di primo  
livello) del tag body. */
```

- Selezione del **prossimo elemento**:

```
 $(".foo+p"); /* seleziona il prossimo paragrafo dopo il tag di classe foo. */
```

- Selezione di **elementi “fratelli”**:

```
 $(".foo~p"); /* seleziona tutti i prossimi paragrafi allo stesso livello dopo il tag di  
classe foo. */
```

- I filtri consentono di selezionare elementi del DOM in base alla loro **posizione**, al loro **stato** o in base ad altre variabili.
- La sintassi di base prevede l'uso dei due punti (:) seguita dal nome del filtro (**:filtro**).
- Alcuni filtri possono ricevere un parametro (**:filtro(parametro)**)

- **Primo o ultimo elemento:**

```
$("p:first"); // oppure $("p:last");
```

- Elementi che **non corrispondono** ad un selettore:

```
$("p:not(.foo)");
```

- Elementi **pari o dispari**:

```
$("p:odd"); // oppure $("p:even");
```

- Elementi **selezionati tramite indici**:

```
$("p:eq(2)");
```

- Elementi che **contengono uno specifico testo**:

```
$(“p:contains(testo)”);
```

- Elementi che **contengono un determinato elemento**:

```
$(“p:has(span)”);
```

- Elementi **vuoti**:

```
$(“p:empty”);
```

- Elementi **con figli**:

```
$(“p:parent”);
```

- Filtri di selezione degli **elementi visibili** (:visible) e **nascosti** (:hidden), rispettivamente:

```
$(“p:visible”);
```

```
$(“p:hidden”);
```

- Selezionare elementi che hanno un **attributo impostato** ad uno specifico **valore** ([attributo=valore]):
 `$("[class=foo]");`
- Selezionare elementi che **non** hanno un **attributo impostato** ad uno specifico **valore** ([attributo!=valore]):
 `$("[class!=foo]");`
- **Ennesimo figlio:**
 `$("p:nth-child(2)"); /* il conteggio parte da 1 */`
- **Primo o ultimo figlio:**
 `$("p span:first-child");`
 `$("p span:last-child");`

- Selezione tramite il tipo dell'elemento di input:

```
$(“input:radio”);
```

Altri filtri: :button, :checkbox, :file, :image, :password, :submit, :text

- Selezione degli elementi abilitati/disabilitati:

```
$(“:enabled”);
```

```
$(“:disabled”);
```

- Selezione degli elementi selezionati:

```
$(“:checked”);
```

```
$(“:selected”);
```

Comportamento degli script di JQuery

- Il vantaggio di jQuery è che quasi tutti i suoi metodi sono applicabili in cascata.
- Ciò è possibile in quanto ogni metodo restituisce l'oggetto jQuery modificato.
- Ne risulta una semplificazione nei seguenti compiti:
 - navigazione e modifica del DOM;
 - gestione con una sintassi semplice degli eventi del browser;
 - accesso e modifica di tutti gli attributi degli elementi;
 - animazioni ed altri effetti visivi;
 - accesso ad AJAX facilitato.

Creare nuovi elementi del DOM

- Creare un nuovo paragrafo (notare la presenza delle parentesi angolate):

```
$( "<p>" );
```

- A questo punto possiamo aggiungere attributo e testo al nuovo paragrafo:

```
$('<p class="bat">Questo &egrave; un nuovo paragrafo!</p>');
```

- Sintassi alternativa:

```
$( "<p>", {  
    "class": "bat",  
    "text": "Questo &egrave; un nuovo paragrafo!"  
});
```

Si noti che il nuovo elemento non è ancora visibile perché non è stato ancora inserito nel DOM.

Inserire elementi nel DOM

- **append()** e **prepend()** aggiungono, in coda ed in testa rispettivamente, gli elementi passati come argomenti internamente all'elemento a cui vengono concatenati.

```
var paragrafo = $("<p>", {  
    "css": {"background": "yellow"},  
    "text": "Questo è un nuovo paragrafo!"  
});  
$("body").prepend(paragrafo);
```

- **appendTo()** e **prependTo()**:

```
$("<p>", {  
    "css": {"background": "yellow"},  
    "text": "Questo è un nuovo paragrafo!"  
}).prependTo("body");
```

- **after()** e **before()**: come append() e prepend(), ma aggiungono il contenuto fuori dall'elemento a cui vengono concatenti.

- **insertAfter()** e **insertBefore()** stanno a after() e before() come appendTo() e prependTo() stanno a append() e prepend().

- **wrap()** serve a racchiudere elementi esistenti con nuovi elementi:

```
 $("span").wrap("<strong />");
```

- **unwrap()**: rimuove i tag che racchiudono l'elemento a cui viene concatenato.

- **wrapAll()**: svolge la stessa azione di wrap(), ma applicata a più elementi contemporaneamente.

- **wrapInner()**: racchiude il contenuto di un elemento, ma non i suoi tag.

Rimuovere elementi dal DOM

- Il metodo `remove()` consente di rimuovere gli elementi selezionati dal DOM:
 `$("p").remove(".foo");`

Il codice precedente rimuove dal documento tutti i paragrafi con classe foo.

- Per **nascondere** il paragrafo con id bar:

```
$("#bar").hide();
```

- L'elemento **rimane** tuttavia presente **nel DOM** e per tornare a visualizzarlo è sufficiente eseguire:

```
$("#bar").show();
```

- Esempio più complesso:

```
$("#bar")
  .css({ "background": "yellow",
          "border": "1px solid black"
        })
  .hide(2000, function() { console.log("Animazione completata!"); });
```

- si possono usare anche i metodi **slideUp** e **slideDown** o **slideToggle**

- Fade in, fade out e fadeTo consentono di sfruttare e gestire il livello di trasparenza degli elementi:

```
$("#form")
  .fadeOut(1000, function(){ console.log("Faded out!"); })
  .fadeIn(1000, function(){ console.log("Faded in!"); });
```

- Fade in, fade out e fadeTo consentono di sfruttare e gestire il livello di trasparenza degli elementi:

```
$( "form" )
.fadeOut(1000, function(){ console.log("Faded out!"); })
.fadeIn(1000, function(){ console.log("Faded in!"); });
```

La funzione principale per inviare richieste AJAX è il metodo statico **\$.ajax()**.

Dati i molti aspetti della chiamata che possono essere personalizzati, la funzione accetta un unico oggetto JavaScript con i parametri di base ed altri necessari per sovrascrivere i valori di default.

JQuery Ajax: parametri di base

- **url**: l'indirizzo al quale inviare la chiamata
- **success**: funzione da lanciare se la richiesta ha successo. Accetta come argomenti i dati restituiti dal server (interpretati di default come html o xml) e lo stato della chiamata
- **error**: funzione lanciata in caso di errore. Accetta un riferimento alla chiamata XMLHttpRequest, il suo stato ed eventuali errori notificati

Con questi tre parametri è possibile impostare una prima semplice chiamata AJAX di esempio:

```
$.ajax({  
    url : "mioserver.html",  
    success : function (data,stato) {  
        $("#risultati").html(data);  
        $("#statoChiamata").text(stato);  
    },  
    error : function (richiesta,stato,errori) {  
        alert("E' evvenuto un errore. Il stato della chiamata: "+stato);  
    }  
});
```

Nell'esempio, se la chiamata ha successo i dati verranno inseriti all'interno di specifici elementi DOM, altrimenti verrà mostrato un messaggio di errore

JQuery Ajax: parametri opzionali

https://www.w3schools.com/jquery/ajax_ajax.asp

Name	Value/Description
async	A Boolean value indicating whether the request should be handled asynchronous or not. Default is true
beforeSend(xhr)	A function to run before the request is sent
cache	A Boolean value indicating whether the browser should cache the requested pages. Default is true
complete(xhr,status)	A function to run when the request is finished (after success and error functions)
contentType	The content type used when sending data to the server. Default is: "application/x-www-form-urlencoded"
context	Specifies the "this" value for all AJAX related callback functions
data	Specifies data to be sent to the server
dataFilter(data,type)	A function used to handle the raw response data of the XMLHttpRequest
dataType	The data type expected of the server response.
error(xhr,status,error)	A function to run if the request fails.
global	A Boolean value specifying whether or not to trigger global AJAX event handles for the request. Default is true
ifModified	A Boolean value specifying whether a request is only successful if the response has changed since the last request. Default is: false.
jsonp	A string overriding the callback function in a jsonp request
jsonpCallback	Specifies a name for the callback function in a jsonp request
password	Specifies a password to be used in an HTTP access authentication request.
processData	A Boolean value specifying whether or not data sent with the request should be transformed into a query string. Default is true

JQuery Ajax: parametri opzionali

https://www.w3schools.com/jquery/ajax_ajax.asp

scriptCharset	Specifies the charset for the request
success(<i>result,status,xhr</i>)	A function to be run when the request succeeds
timeout	The local timeout (in milliseconds) for the request
traditional	A Boolean value specifying whether or not to use the traditional style of param serialization
type	Specifies the type of request. (GET or POST)
url	Specifies the URL to send the request to. Default is the current page
username	Specifies a username to be used in an HTTP access authentication request
xhr	A function used for creating the XMLHttpRequest object

Conoscendo questi parametri sarà quindi possibile richiedere dei dati in formato JSON da inserire in una tabella HTML (esempio):

```
$.ajax({  
    url : 'dati.php',  
    data : 'primariga=0&ultimariga=10', //le prime 10 righe  
    dataType : 'json', //restituisce un oggetto JSON  
    complete: function (righe,stato) {  
        for (i=0; i < righe.length; i++) {  
            var riga = righe[i];  
            $("<tr/>")  
                .append("<td>" + riga.colonna1 + "</td><td>" + riga.colonna2 + "</td>")  
                .appendTo("#tabella");  
        }  
    }  
});
```