



Evoluzione del WEB

di Roberta Molinari

WEB 1.0

only read web

- ▶ **1990 - 2000**
 - ▶ Secondo la definizione dello stesso Berners sono stati gli anni dell’”**only read web**”.
 - ▶ **Pagine ipertestuali statiche**
 - ▶ Era l’internet dei contenuti, contrassegnata da **siti web statici**, realizzati in semplice **HTML**, con una frequenza di aggiornamento ridotta.
 - ▶ Solo i webmaster avevano le competenze tecniche necessarie e gli strumenti per poter aggiornare le pagine di un sito internet.
 - ▶ **L’utenza poteva così solo usufruire dei contenuti** senza creare interazione e le pagine offrivano la possibilità di essere semplicemente consultate
-



WEB 1.0

only read web

- ▶ **12 marzo 1989** - Lo scienziato informatico Tim Berners-Lee pubblica "Information management: a proposal". Nel paper viene delineata la sua visione di ciò che diventerà il World Wide Web.
- ▶ **6 agosto 1991** - Il primo sito web al mondo viene messo online all'indirizzo <http://info.cern.ch/hypertext/WWW/TheProject.html>

World Wide Web

The WorldWideWeb (W3) is a wide-area [hypermedia](#) information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an [executive summary](#) of the project, [Mailing lists](#) , [Policy](#) , November's [W3 news](#) , [Frequently Asked Questions](#) .

[What's out there?](#)

Pointers to the world's online information, [subjects](#) , [W3 servers](#), etc.

[Help](#)

on the browser you are using

- ▶ **23 gennaio 1993** - Nasce Mosaic, il primo browser utilizzato a livello commerciale grazie al National Center for Supercomputing Applications dell'Università dell'Illinois.

WEB 1.0

only read web

- ▶ **30 aprile 1993** - il CERN, su proposta di Tim Berners-Lee, decide di rendere pubblico il World Wide Web.

“Se la tecnologia fosse stata proprietaria e sotto il mio totale controllo, probabilmente non sarebbe decollata. La decisione di rendere il web un sistema aperto fu necessaria per renderlo universale. Non puoi proporre qualcosa come uno spazio universale e al tempo stesso mantenere il controllo su di esso”.

WEB 1.0

only read web

- ▶ **18 gennaio 1994** - Viene registrato il dominio di Yahoo!, primo sito che cerca di categorizzare il crescente mondo online
- ▶ **Ottobre 1994** - Berners-Lee fonda il World Wide Web Consortium (W3C), l'organizzazione che sviluppa i protocolli e le linee guida del web per assicurarne una costante espansione
- ▶ **15 agosto 1995** - Nasce la prima versione di Internet Explorer, il browser di Microsoft che diventerà rapidamente uno dei più popolari del web
- ▶ **15 settembre 1997** - Larry Page e Sergei Brin registrano il dominio di Google.com: il motore di ricerca che, grazie all'innovativo algoritmo PageRank, riuscirà a dominare questo nascente mercato



WEB 2.0

read-write web

- ▶ **2000 – 2006**
- ▶ **“read-write web”**
- ▶ **Pagine dinamiche interattive**
- ▶ Aumentano le funzionalità dei browser attraverso un'evoluzione del linguaggio HTML e la possibilità d'interpretazione di **linguaggi di scripting** (come il JavaScript)
- ▶ si migliora la qualità di elaborazione dei server attraverso una nuova generazione di **linguaggi integrati con il web server** (come JSP, PHP, ASP, etc.), trasformando pertanto i web server negli attuali application server.
- ▶ L'utenza non tecnica **interagisce** con i contenuti dei siti internet.
- ▶ Per la prima volta si è data grande importanza all'usabilità e al modo di condividere i contenuti.
- ▶ Partecipazione attiva degli utenti alla **costruzione dei contenuti**, alla loro classificazione e distribuzione; gli elementi principali sono i Blogs, Wiki, Social Network, Forum



WEB 2.0

read-write web

- ▶ **2000** - Gli utenti internet nel mondo superano i 400 milioni
 - ▶ **10 marzo 2000** - La bolla delle dot-com raggiunge il suo apice: il Nasdaq chiude a 5048 punti. Un anno dopo avrà perso oltre il 70%.
 - ▶ **15 gennaio 2001** - Jimmy Wales e Larry Sanger lanciano Wikipedia, l'enciclopedia online aperta e collaborativa che sfrutta i principi non commerciali alla base dell'open web
 - ▶ **4 febbraio 2004** - Nasce TheFacebook, prima incarnazione di quello che diventerà il social network da oltre due miliardi di utenti
-
- ▶ **2005** - Gli utenti di internet nel mondo superano il miliardo

WEB 3.0

read-write-execute web

- ▶ **2006 –**
- ▶ **read-write-execute web**
- ▶ **dati e semantica**
- ▶ **Internet come un enorme database** da utilizzare in diverse applicazioni per recuperare dati da fornire all'utenza, si parla di **Data Web**.
- ▶ **web semantico**, si concentra sull'organizzazione e la strutturazione dei dati in modo che le informazioni siano comprensibili non solo alle persone, ma anche alle macchine. Ciò significa che i contenuti web sono etichettati con metadati strutturati che consentono ai motori di ricerca e ad altri strumenti di riconoscere, interpretare e utilizzare le informazioni in modo più efficiente.;
- ▶ **L'intelligenza artificiale**, permette ai motori di ricerca interrogazioni attraverso il linguaggio naturale e anche di reperire informazioni individuando necessità e gusti degli utenti in base al loro comportamento in rete;



WEB 3.0

read-write-execute web

- ▶ **web potenziato** capace di modificare la società: grazie ai Social Network
- ▶ **web in 3D:** le nuove tecnologie e l'elevato accesso alla rete internet hanno permesso di replicare la realtà in formato digitale. Possiamo quindi accedere alla rete e effettuare buona parte delle interazioni che compongono la nostra vita reale. Il primo esempio è stato second life

WEB 4.0

- ▶ Adesso
 - ▶ “spazio” e “big data”.
1. la **realtà aumentata**: ci permette di interagire in tempo reale con il web sovrapponendo il mondo che ci circonda con la rete.
 2. L'**alter ego digitale**: si crea pian piano che i nostri documenti si aggiornano e collegano fra loro, man mano che popoliamo la rete con i nostri contenuti personali.
 3. Le nuove **interfacce domotiche**, dei nostri elettrodomestici e nelle nuove automobili intelligenti ci permetterà di scambiare i dati relativi al mondo reale con il nostro alter ego digitale. Grazie anche all'**IOT**
- ▶ Il maggiore controllo dell'informazione permetterà di modificare la realtà circostante.

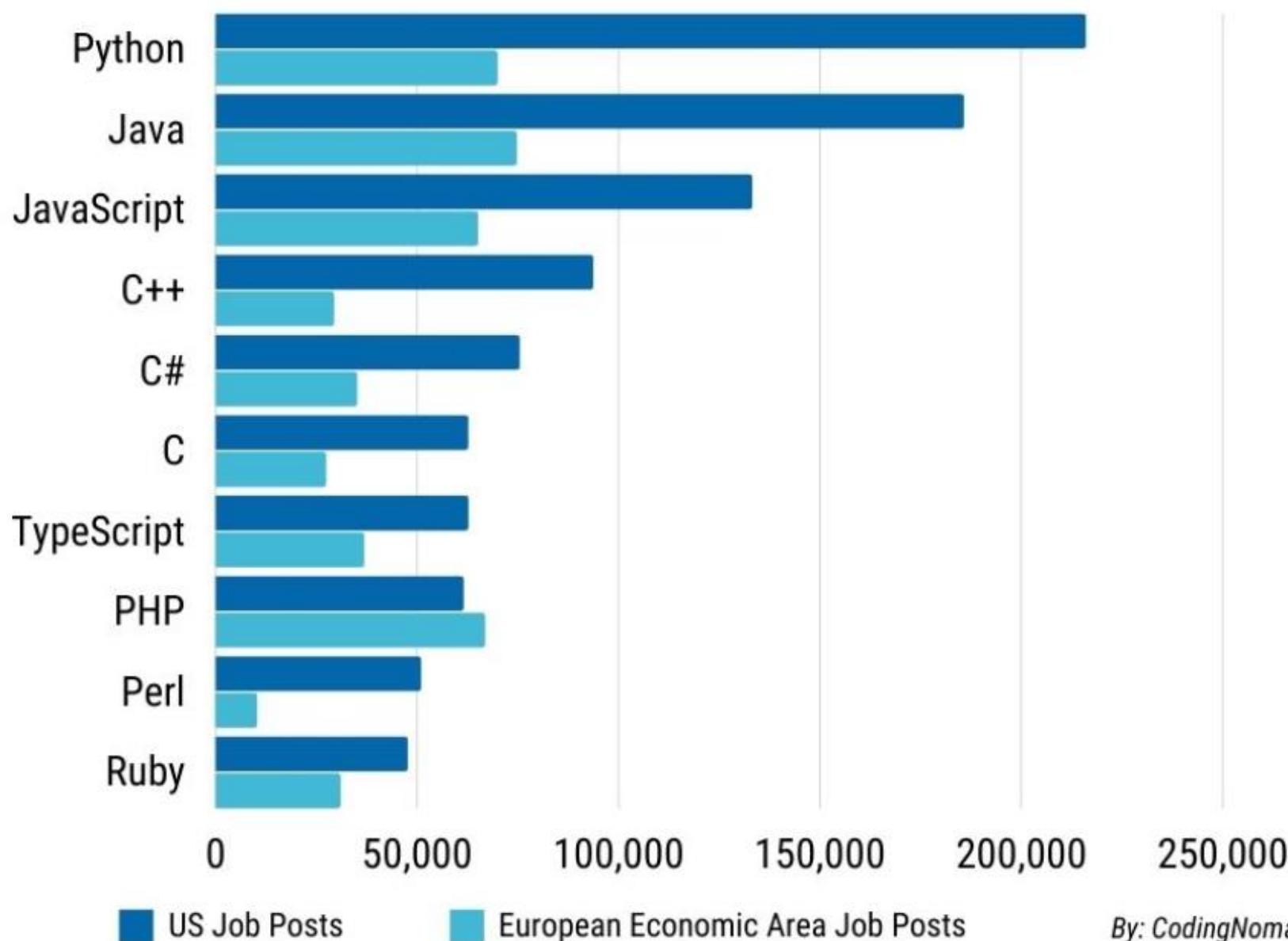


JavaScript

di Roberta Molinari

Most in-demand programming languages of 2022

Based on LinkedIn job postings in the USA & Europe



US Job Posts

European Economic Area Job Posts

By: CodingNomads

JavaScript

- ▶ È stato creato nel 1995 da Brendan Eich in soli 10 giorni, come parte del browser *Netscape Navigator 2.0*.
- ▶ Il linguaggio di programmazione inizialmente è stato prima chiamato *Mocha*, poi *LiveScript* e infine *JavaScript* in onore dell'azienda con cui collabora per la sua realizzazione (la Sun Microsystems, dal 2010 acquistata dalla Oracle, che stava creando Java).
- ▶ La sua prima versione standardizzata è l'*ECMA Script I (ESI)* del 1997.

JavaScript

JavaScript è il primo linguaggio di scripting del web. Le sue caratteristiche:

- ▶ **interpretato,**
- ▶ **object based basato su prototipi,**
- ▶ **single-thread, non bloccante**
- ▶ **multi-paradigma, (orientato agli oggetti, imperativo e dichiarativo-funzionale)**
- ▶ **guidato dagli eventi.**

Linguaggio di scripting: linguaggio interpretato destinato in genere a compiti di automazione del sistema operativo (*batch*) o delle applicazioni (*macro*), o a essere usato nelle pagine web

JavaScript utilizzi

-
- 1. Sviluppo web.** È utilizzato principalmente per la creazione di interattività e dinamicità all'interno delle pagine web: animazioni, effetti grafici, menu a discesa, form di validazione ...
 - 2. Applicazioni web.** Con l'aiuto di framework JavaScript come Angular, React, Vue.js e molti altri, si possono creare applicazioni web altamente dinamiche e interattive.
 - 3. Applicazioni desktop.** Grazie a tecnologie come Electron, può essere utilizzato anche per la creazione di applicazioni desktop.
 - 4. Applicazioni mobile.** Con tecnologie come React Native e Ionic, gli sviluppatori possono creare applicazioni mobile cross-platform utilizzando JavaScript ibride e native.
 - 5. Server-side:** utilizzando Node.js si possono creare applicazioni server side altamente scalabili e performanti utilizzando un unico linguaggio di programmazione.

JavaScript storia

- ▶ 1995-2000: Inizialmente molto semplice e limitato, nasce nel 1995 nel browser Navigator 2.0 della Netscape come "LiveScript". Ribattezzato JavaScript in onore dell'azienda con cui si collabora per la sua realizzazione (la Sun Microsystems, dal 2010 acquistata dalla Oracle, che stava creando Java). Microsoft realizza un linguaggio simile nel 1996 chiamandolo Jscript: oggi abbandonato. Netscape e Sun standardizzarono il linguaggio nel (1997 ESI)
- ▶ 2000-2005: JavaScript è diventato sempre più popolare e ha iniziato a essere utilizzato per creare applicazioni web più complesse. Sono state introdotte nuove librerie e framework, come jQuery e Prototype, che hanno semplificato la creazione di interattività.

JavaScript storia

- ▶ 2006-2010: Nel 2006, è stato introdotto il framework JavaScript moderno AngularJS, che ha reso più facile creare applicazioni web complesse. Inoltre, sono stati introdotti anche nuovi motori JavaScript come V8, che ha migliorato notevolmente le prestazioni di JavaScript. Nel 2009 nasce Node.js ambiente di runtime JavaScript open-source basato su Chrome V8 engine, che consente di eseguire codice JavaScript lato server.
- ▶ 2010-2014: è stato introdotto il framework ReactJS, che ha semplificato la creazione di interfacce utente complesse.

JavaScript

storia

- ▶ 2015-2020: Nel 2015, è stato introdotto il nuovo standard **ECMAScript 6 (ES6)**, che ha introdotto nuove funzionalità avanzate come le **classi**, le **arrow function**, **const** e **let**. Ha anche introdotto il concetto di "**JavaScript asincrono**" con l'introduzione delle **Promises**, rendendo più facile la gestione di richieste asincrone (che dalla versione del 2017 si potranno gestire anche come **async/await**). Questa versione ha reso JavaScript più facile da leggere e scrivere.

Sono stati introdotti anche framework come Vue.js e React Native, che hanno semplificato la creazione di applicazioni mobili.



JavaScript

storia

-
- ▶ 2020-oggi: Negli ultimi anni, sono stati introdotti nuovi concetti come TypeScript, un linguaggio di programmazione basato su JavaScript che fornisce un sistema di tipi statici. Sono stati anche introdotti nuovi framework come AngularJS 2 e ReactJS Hooks, che semplificano la creazione di applicazioni web avanzate.

JavaScript compatibilità

Browser web [modifica | modifica wikitesto]

Engine	Browser	Conformità		
		ES5 ^[38]	ES6 ^[39]	2016+[40][41]
V8	Microsoft Edge 80	100%	98%	100%
SpiderMonkey	Firefox 73	100%	98%	82%
V8	Google Chrome 80, Opera 67	100%	98%	100%
JavaScriptCore (Nitro)	Safari 13	99%	99%	80%

Implementazioni lato server [modifica | modifica wikitesto]

Engine	Server	Conformità		
		ES5 ^[38]	ES6 ^[39]	2016+[40][41]
V8	Node.js 13.2	100%	98%	92%

JavaScript

Caratteristiche

- ▶ Il browser visualizza il documento HTML e **interpreta** (esegue) le eventuali istruzioni scritte in Javascript.
- ▶ È **guidato dagli eventi**: se l'utente genera un evento e esiste del codice JavaScript associato a quell'evento (event handler), questo viene eseguito
- ▶ Fornisce i costrutti di un linguaggio di programmazione: variabili, espressioni, istruzioni, ...
- ▶ Ricorda C e Java, oggi è un linguaggio completo dotato di numerose librerie con nulla da invidiare ad altri linguaggi ad alto livello

JavaScript

Cosa serve

- ▶ Un editor di testo + motore che interpreta JavaScript lato client (es. V8 della Google, in Chrome, Chakra di Microsoft, SpiderMonkey di Mozilla su Firefox)
 - Per debuggare si può usare "*Analizza elemento*" → *Debugger* in Firefox o F12 in Chrome

JavaScript

Come inserire uno script

1. Inserendo direttamente nel documento HTML in un qualunque punto dell'head (non potrà fare riferimento ad oggetti del <body> perché non sono ancora stati definiti) o del body

```
<script [type="text/javascript"]>  
    codice JavaScript  
</script>
```

Il browser interpreta il codice quando lo incontra e lo esegue. Se c'è un errore può:

- ▶ visualizzare il documento, ma non eseguire il codice errato
- ▶ visualizzare il documento parzialmente o in bianco perché l'esecuzione dall'alto al basso è interrotta

JavaScript

Come inserire uno script

2. Caricandolo da un file di testo esterno con estensione .js

```
<script [type="text/javascript"]  
src="myfile.js"></script>
```

- ▶ Lo script viene eseguito dopo aver trasferito il codice esattamente nel punto in cui è inserito il tag. Se non deve essere eseguito prima del caricamento del documento, si mette in fondo prima del </body> per non rallentarlo
- ▶ Il file .js non deve contenere tag HTML o elementi di altri linguaggi
- ▶ È il metodo consigliato se si vuole:
 - proteggere il codice sorgente
 - riutilizzare il codice in più documenti

JavaScript

Come inserire uno script

3. Incorporandolo all'interno dei tag HTML come valore dei nuovi attributi che sono stati introdotti per gestire gli eventi generati dall'utente

```
<div onMouseOver ="JavaScript: codice;"  
onMouseOut ="JavaScript: codice;"  
onClick ="JavaScript: codice;">  
Testo</div>
```

o cliccando su un link

```
<a href="JavaScript: codice  
JavaScript;"> Clicca qui  
</a>
```

JavaScript

Regole sintattiche

- ▶ Ogni istruzione inizia in una nuova riga o dopo il ;
Pertanto l'uso del ; è obbligatorio solo se si scrivono più istruzioni sulla stessa riga. È fortemente consigliato per evitare errori di interpretazione.
- ▶ Commenti

```
// su una riga sola ci vuole lo spazio prima  
/* commento su più righe */
```

- ▶ Gli identificatori possono iniziare con _ \$ o con una lettera e non possono iniziare con un numero, non possono contenere spazi o caratteri di punteggiatura
- ▶ Non si possono usare le parole riservate
- ▶ È case sensitive

JavaScript

Tipizzazione

- ▶ Tipizzazione: assegnazione del tipo alla variabile
 - Static: durante la compilazione
 - Dynamic:
 - dinamica forte, i valori assegnati alle variabili hanno dei tipi ben definiti
 - dinamica debole, le variabili possono riferirsi a valori di qualsiasi tipo, che possono cambiare dinamicamente in seguito a manipolazioni esterne.
- ▶ Javascript è debolmente tipizzato: le variabili non sono tipate, possono valere qualunque cosa, anche essere una funzione

JavaScript

Tipizzazione

- ▶ JavaScript è un linguaggio "debolmente tipato" quindi una variabile può cambiare tipo nel corso del suo ciclo di vita. Il tipo delle variabili è stabilito in fase di esecuzione (dynamic typing), dipende dall'ultima operazione di assegnamento eseguita

```
x = 10;  
x = "s";      //prima x è un numero poi stringa
```

- ▶ I tipi attualmente (ECMAScript 2022) sono
 - ▶ 7 tipi primitivi
 - ▶ 1 tipo Object

JavaScript

typeof() o typeof

La definizione stabilisce il nome della variabile, mentre il tipo dipende dall'ultima assegnazione.

Per verificare il tipo della variabile in un dato momento si usa `typeof()` o `typeof`, che può restituire la stringa del tipo corrispondente.

È *object* qualunque tipo non primitivo.

```
x = 10;                      //typeof(x)→ "number"  
x = "a";                      //typeof(x) → "string"  
x = new String();              //typeof(x )→ "object"  
typeof false;                  // → "boolean"  
typeof ({name: 'John', età: 34}); //→ "object"  
typeof (null) //→ "object"
```

JavaScript

Tipizzazione: 7 tipi primitivi

I tipi "primitivi" hanno degli oggetti wrapper in cui vengono convertiti automaticamente quando si utilizza un suo metodo o una sua proprietà.

```
s = "ciao".toUpperCase();
```

Type	typeof return value	Object wrapper
<u>Null</u>	"object"	N/A
<u>Undefined</u>	"undefined"	N/A
<u>Boolean</u>	"boolean"	<u>Boolean</u>
<u>Number</u> solo double a 64 bit	"number"	<u>Number</u>
<u>BigInt</u> ECMAScript 2020	"bigint"	<u>BigInt</u>
<u>String</u>	"string"	<u>String</u>
<u>Symbol</u>	"symbol"	<u>Symbol</u>

JavaScript

Variabili: tipi

- Se una variabile è dichiarata con `new`, allora è un oggetto wrapper corrispondente al tipo primitivo

```
x = new Number(1.2);
```

```
y = new Boolean(true);
```

```
s = new String("ciao");
```

`typeof()` restituisce sempre `object`

JavaScript

Null, Undefined

- ▶ **undefined**: è il valore di una variabile dichiarata, ma non inizializzata. Il tipo non è definito, per cui si considera che sia NaN

```
let x;           //Il valore è undefined  
typeof(x)       //undefined  
isNaN (x)       //true
```

- ▶ Le variabili definite possono essere svuotate del loro valore impostando il loro valore a **null** (indica che il valore è sconosciuto). Sono trasformate in oggetto. Significa vuoto, sconosciuto e NON puntatore nullo.

```
let y = 10;  
y = null;        // ora svuoto il valore è null  
typeof(y)        // object
```

JavaScript

Null, Undefined

- ▶ **null**: indica che alla variabile non è stato assegnato deliberatamente un valore.
- ▶ **undefined**: indica una variabile non inizializzata, cioè a cui non è stato ancora assegnato un valore e quindi non ha un tipo definito.

```
null == undefined //true uguali come VALORE
```

```
null === undefined //false diversi come TIPO
```

JavaScript

NaN

- ▶ **NaN**: è un numero (`typeof()` restituisce `number`) con il significato "non è un numero", che viene assegnato quando si fa un'operazione numerica che non restituiscono un numero

```
let x = 100 / "A"; // x = NaN (Not a Number)
```

Non si verifica con `x == NaN` ma con `isNaN(x)`

▶ Attenzione!

- ▶ `NaN == undefined` //false
- ▶ `isNaN(undefined)` //true
- ▶ `isNaN(null)` //false
- ▶ `NaN == null` //false
- ▶ `NaN === undefined === null` //false

JavaScript

Infinity

- ▶ **Infinity**: è un numero (typeof() restituisce number) che viene assegnato quando si deve assegnare un valore superiore al massimo memorizzabile o inferiore al minimo (-Infinity)

```
let x = 2 / 0;           // Infinity  
let y = -2 / 0;          // -Infinity
```

Si verifica con `x==Infinity` o con `isFinite(x)`

`Infinity > 1000 // true`

JavaScript

Boolean

- ▶ Un valore **falsy** (falseggiante) è qualcosa che restituisce FALSE. È considerato falsy:
 - false
 - 0 and -0 "" and " (empty strings)
 - null (vuoto, sconosciuto e NON puntatore nullo)
 - undefined(valore e quindi tipo non assegnato)
 - NaN (Not a Number)

Attenzione! null, undefined e NaN sono considerati falsy, ma non sono == false

```
if (!NaN) console.log("è un numero")
```



JavaScript

Tipi stringa

► Per le stringhe si può usare:

- indifferentemente ' o "
- ` Backticks (ALT+96) ci permettono di incorporare variabili ed espressioni in una stringa inserendole in \${...}

```
let name = "John";  
  
alert(`Hello, ${name}!`); //Hello, John!  
  
alert(`the result is ${1 + 2}`);  
// the result is 3
```

► NON esiste il tipo char

► Per interpretare in modo corretto i caratteri ',' e \ in una stringa, usare \" o \' o \\

JavaScript

Conversione tra tipi: implicita

- ▶ Operatore +: se almeno uno dei due operandi è una stringa, allora viene effettuata una concatenazione di stringhe, altrimenti viene eseguita una addizione.

```
x = 10 + "3"           //x="103"
```

```
x = true + null      //x=1
```

i valori *true* e *null* vengono convertiti in numeri

- ▶ Se l'operatore è un numerico si converte in numero

```
x = 10 * "3"          //x=30
```

- ▶ Per gli operatori relazionali *>*, *>=*, *<* e *<=*: se nessuno dei due operandi è un numero, allora viene eseguito un confronto tra stringhe, altrimenti viene eseguito un confronto tra numeri.



JavaScript

Casting implicito

► Conversione di tipo implicita

`x = 2 + "3"` x è stringa = "23", basta una stringa

`x ="2" * "3"` x è un numero = 6, l'operatore converte

`x = 2 * "3A"` non è possibile convertire in numero la
stringa quindi x = NaN

JavaScript

Conversione tra tipi: implicita

► Da tipo →NUMBER

Tipo	Valore numerico
undefined	NaN se x è undefined e faccio x+4 ottengo NaN
null	0 se x=null e faccio x+4 ottengo 4
booleano	1 se true, 0 se false
stringa	intero, decimale, zero o NaN in base alla specifica stringa

► Da tipo →BOOLEAN

Tipo	Valore booleano
undefined	false
null	false
numero	false se 0 o NaN, true in tutti gli altri casi
stringa	false se stringa vuota, true in tutti gli altri casi

JavaScript

Conversione tra tipi: implicita

► Da tipo → STRING

Tipo	Valore stringa
undefined	"undefined"
null	"null"
booleano	"true" se true "false" se false
numero	"NaN" se NaN, "Infinity" se Infinity la stringa che rappresenta il numero negli altri casi

Qual è il risultato delle espressioni seguenti?

"" + 1 + 0

7 / 0

"" - 1 + 0

" -9 " + 5

true + false

" -9 " - 5

6 / "3"

null + 1

"2" * "3"

undefined + 1

4 + 5 + "px"

"\$" + 4 + 5

"4" - 2

"4px" - 2



JavaScript

== e ===

- ▶ Per == e != vale: *se entrambi gli operatori sono stringhe allora viene effettuato un confronto tra stringhe, altrimenti si esegue un confronto tra numeri; unica eccezione è*

null == undefined

che è vera per definizione

- ▶ operatori di uguaglianza e disuguaglianza stretta (== e !==). Questi operatori confrontano gli operandi senza effettuare alcuna conversione. Quindi *due espressioni vengono considerate uguali soltanto se sono dello stesso tipo ed rappresentano effettivamente lo stesso valore*. Nel caso di due oggetti restituisce sempre false a meno che puntino alla stessa area

JavaScript

== e ===

- ▶ Attenzione al confronto tra tipi elementari e classi wrapper corrispondenti

```
"ciao" == new String("ciao") // true  
"ciao" === new String("ciao") // false
```

```
3 == new Number(3) // true  
3 === new Number(3) // false
```

JavaScript

==== e =====

1==true //T

1==>true //F

```
12=='12' //T
```

```
12==='12' //F
```

""==false //T

```
"""==>false //F
```

$\theta == " "$ // T

$\theta == " "$ // F

null ==

undefined //T

null ===

JavaScript

Conversione tra tipi: esplicita

```
s= String(10); //s="10"  
n= parseInt("10 gradi"); //n=10  
n= parseInt("11 gradi", 2); //n=112 ovvero 3
```

//il secondo parametro rappresenta la base numerica

La funzione `eval()` prende come argomento una stringa
e la valuta o la esegue come se fosse codice JavaScript

```
x='a'; y='ab'; z=(eval('x<y')); //z=true  
s1= "2+2"; z=eval(s1); //z=4  
s2= new String("2+2"); z=eval(s2); //z="2+2"  
x=eval(s2.valueOf()); //x=4  
x=10; y=20; z=eval("x+y+40") //z=70
```

JavaScript

Casting esplicito

► Conversione di tipo esplicita

VarStr=String(VarNum) ; da numero a stringa

VarNum=parseInt(VarStr[,BaseNumerica]) ;

da stringa a numero senza decimale (basta che la stringa inizi con un numero),

VarNum=parseFloat(VarStr) ; da stringa a float

VarNum=eval(VarStr) ; cerca di convertire in numerico
una qualunque espressione (tipo "2+2") se non ci riesce
restituisce NaN

VarNum=Number(VarStr) ; //restituisce un numero
 NON un oggetto

parseInt("39 gradi") //39

Number ("39 gradi") //NaN

JavaScript

Variabili

- ▶ Prima di essere adoperata una variabile può essere dichiarata (non necessario, il controllo è lasco)

[var|let] nome [=valore]; //dichiarazione con eventuale inizializzazione. Se non si assegna un valore la variabile è "undefined"

```
let foo, boo=true; //inizializzata a true solo boo
```

```
let x=y=z=3; //inizializzate tutte a 3, ma solo x ha let
```

```
let a="il",b='lo'; //sono due stringhe
```

- ▶ Una variabile non dichiarata viene automaticamente creata al primo utilizzo ed è accessibile da qualsiasi punto di uno script (è globale). L'utilizzo di var|let dà la possibilità di stabilire lo scope

JavaScript

Variabili: scope

Le variabili possono essere:

- ▶ **globali**: sono accessibili da qualsiasi punto dello script. Sono:
 - ▶ dichiarate senza `var|let` in un qualunque punto, anche dentro ad una funzione
 - ▶ dichiarate con `var|let` fuori da qualsiasi funzione.
- ▶ **locali**: dichiarate con `var|let` all'interno di una funzione e sono accessibili soltanto all'interno del suo corpo; occupano spazio solo per il tempo di esecuzione della funzione. Con `let` si posso dichiarare anche a livello di blocco.

JavaScript

Variabili var: hoisting(sollevamento)

Le dichiarazioni delle variabili vengono elaborate prima dell'esecuzione di qualsiasi codice. Quindi una variabile può essere usata prima che sia dichiarata. Viene fatto **l'hoisting** la dichiarazione della variabile "viene spostata" all'inizio del suo ambito, ma il valore verrà effettivamente assegnato al raggiungimento dell'istruzione. Pertanto si raccomanda di dichiarare sempre le variabili all'inizio del loro ambito.

```
console.log (greeter);  
var greeter = "say hello"
```

è interpretato come

```
var greeter;  
console.log(greeter); // greeter is undefined  
greeter = "say hello"
```



JavaScript

Variabili var: scope

Dichiarare una variabile all'interno di un blocco di codice {} con **var** NON crea un nuovo scope per la variabile dichiarata (come in C), resta visibile anche al di fuori del blocco.

È possibile dichiarare due volte la stessa variabile con var all'interno dello stesso blocco

Con var la variabile avrà scope di funzione e sarà utilizzabile (nel corpo della funzione) anche nelle righe che precedono la dichiarazione stessa (hoisting)

JavaScript

Variabili var: scope

```
var x = 0; // x globale
alert(typeof z); // undefined, z non esiste ancora
function a() {
    var y = 2; // y ha scope di funzione
    alert(x +" "+ y); // 0 2
    b(); // chiamando b() si crea z globale
    function b() { //genera ReferenceError con strict mode)
        x = 3; //assegna 3 alla var globale x
        y = 4; //assegna alla y di a() non crea una var globale
        z = 5; //crea una var globale
    }
    alert(x +" "+ y +" "+ z); // 3 4 5
}
a(); //chiama a() e quindi anche b()
alert(x +" "+ z); // 3 5
alert(typeof y); // undefined perchè y ha scope di funzione
```



JavaScript

Variabili let

A differenza di var che è inizializzata come undefined, la variabile let non è inizializzata.

Quindi se provi a usare una variabile let prima della sua dichiarazione, otterrai un errore ReferenceError (non viene fatto l'hoisting)

```
console.log (saluto);  
let saluto = "say hello"  
//ReferenceError: can't access lexical declaration  
'saluto' before initialization
```

Dal 2015 JavaScript si usa **let + strict mode** (stile moderno)



JavaScript

Variabili let: scope

let, definita dalle specifiche di ECMAScript 6 (2015), permette di dichiarare variabili limitandone la visibilità ad un blocco di codice, ad un'assegnazione, ad un'espressione in cui è usata

Inoltre una variabile dichiarata con **let** non sarà soggetta all'hoisting e NON è possibile dichiarare due volte la stessa variabile con **let/var** all'interno dello stesso blocco

```
var x = 10; //anche let  
var y;  
{  
    let x= 20 //nasconde quella esterna  
    y = x + 1; //21  
}  
y = x + y; //10+21
```

JavaScript

Variabili let: scope

```
let x = 10;  
if (true) {  
    let y = 20;  
    var z = 30;  
    console.log(x + y + z); // → 60  
}  
// y is not visible here  
console.log(x + z); // → 40
```

JavaScript

Costanti: primitivi

- ▶ Dallo standard ES6 è possibile dichiarare "costanti"

```
const PI=3.14;
```

- ▶ NON definisce un valore costante. Definisce un riferimento costante a un valore. Per questo motivo non possiamo cambiare i valori costanti primitivi

```
PI = PI + 10; // ERROR
```

- ▶ Ha scope a livello di blocco come `let`

```
var x = 10; // Here x is 10
{
    const x = 2; // Here x is 2
}
// Here x is 10
```

JavaScript

Costanti: ridichiarazione

- ▶ Non è possibile la ridichiarazione

```
var x = 2;           // Allowed  
const x = 2;         // Not allowed  
  
{  
    let x = 2;        // Allowed  
    const x = 2;       // Not allowed  
}
```

JavaScript

tipi primitivi e tipi referenza



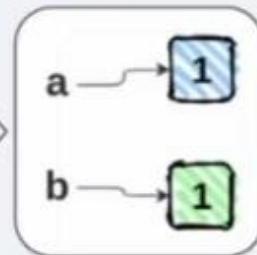
Values VS. References

Ogni volta che assegnamo un valore ad una variabile, viene creata una copia di quel valore. Quando creiamo un oggetto, al contrario, stiamo creando una referenza a quell'oggetto. Se a due variabili viene assegnata la stessa referenza, i cambiamenti all'oggetto si rifletteranno su entrambe le variabili.



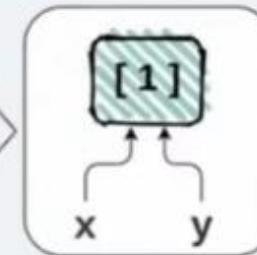
Values

```
let a = 1;  
let b = a;
```



References

```
let x = [1];  
let y = x;
```



00:14



67

67

JavaScript

tipi primitivi e tipi referenza

Primitive data types

Boolean

Null

Undefined

Number

BigInt

String

Symbol

Reference data types

Objects

Arrays

Functions

Dates

MDN JavaScript data types and data structures

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures



JavaScript

tipi primitivi e tipi referenza

```
// Primitives
```

```
let a = 2; let b = a;  
console.log(a, b) //2,2  
b = 3;  
console.log(a, b) //2,3
```

```
// Objects
```

```
const oggettoA = { valore: 2 };  
const oggettoB = oggettoA;  
console.log(oggettoA.valore, oggettoB.valore) //2,2  
oggettoB.valore = 3;  
console.log(oggettoA.valore, oggettoB.valore) //3,3
```

JavaScript

Costanti: oggetto

- ▶ NON definisce un valore costante. Per questo motivo, possiamo cambiare/aggiungere proprietà agli oggetti costanti, ma non riassegnarli.

```
const car = {model:"500", color:"white"};
```

```
// You can change/add a property:
```

```
car.color="red";
```

```
car.owner="John";
```

```
car = {model:"600", color:"red"}; // ERROR
```

JavaScript

Costanti: array

- ▶ NON definisce un valore costante. Per questo motivo, possiamo cambiare/aggiungere elementi ad un array costante, ma non riassegnarli.

```
const cars = ["Saab", "Volvo", "BMW"];
```

```
// You can change/add an element:  
cars[0] = "Toyota";  
cars.push("Audi");
```

```
cars =["Fiat", "Volvo", "Audi"]; //ERROR
```

JavaScript

Costanti: funzioni

```
const dimezza = function(n) {  
    return n / 2;  
};  
  
let n = 10;  
console.log(dimezza(100));    // → 50  
console.log(n);      // → 10  
let x = dimezza(n) //5
```

JavaScript

strict mode

Per rendere obbligatorio la dichiarazione delle variabili (consigliato, per supporto al controllo della correttezza del programma, tipo non modificare variabili omonime dichiarate altrove) si può inserire all'inizio del codice la stringa:

```
"use strict";
```

- ▶ Da un punto di vista sintattico, non si tratta di una istruzione vera e propria, ma di una stringa. Questo garantisce la compatibilità con le versioni precedenti (< standard 5) del linguaggio che semplicemente la ignoreranno senza generare errori.
- ▶ Una volta abilitato lo strict mode, ad ogni assenza di dichiarazione di variabili verrà segnalato un errore.

JavaScript

Funzioni

- ▶ Funzioni si possono dichiarare così (non si dichiara il tipo restituito, avrà uno scope)

```
function nome ([par1,...,parN]) {  
    [var local1,...]  
    ...  
    [return (espressione);] //anche oggetti  
}
```

- ▶ Si richiamera così: nome()

- Le funzioni possono essere richiamate prima della loro definizione, per il meccanismo dell'hoisting
- Se definite in una funzione sono visibili solo dentro essa

JavaScript

Funzioni

- ▶ Sono tutte funzioni, le procedure non restituiscono risultato return è facoltativo. Se non c'è restituisce undefined
- ▶ `typeof` su una funzione restituisce "function"
- ▶ La chiamata può avvenire anche prima della dichiarazione (non per le anonime) e con un numero anche diverso di parametri(quelli mancanti saranno `undefined` e quelli in eccesso non saranno considerati)
- ▶ Normalmente si dichiarano nell'HEAD
- ▶ Il passaggio dei parametri relativi a tipi di dato primitivi avviene sempre per valore per gli altri tipi di oggetti avviene sempre per riferimento.

JavaScript

Funzioni: esempio di scope

```
foo()          //per l'hoisting
function foo() {
    function bar() { //locale anche con var bar =
        console.log("Hello") }
    bar()
}
bar()          //non visibile

//foo1()        //non viene fatto l'hoisting
foo1 = function() {
    bar1 = function() {      //globale
        console.log("Hello 1") }
    bar1()
}
foo1()
bar1()          //visibile
```

JavaScript

Funzioni: scope chain

- ▶ *Gerarchia di scope* o scope chain: una funzione può accedere allo scope locale, allo scope globale ed allo scope accessibile dalla funzione in cui è stata definita (funzione esterna), il quale può essere a sua volta il risultato della combinazione del proprio scope locale con lo scope della sua funzione esterna e così via.
- ▶ in JavaScript *l'accesso allo scope della sua funzione esterna è consentito anche dopo che questa ha terminato la sua esecuzione.*

JavaScript

Funzioni: scope chain

```
var saluto = "Buongiorno";  
var visualizzaSaluto;  
function saluta(persona) {  
    var nc = persona.nome + " " + persona.cognome;  
    return function() {console.log(saluto + " " + nc); };  
}  
visualizzaSaluto =  
    saluta({nome: "Mario", cognome: "Rossi"});  
visualizzaSaluto();
```

- ▶ In questo caso la funzione `saluta()` non visualizza direttamente la stringa ma restituisce una funzione che assolve questo compito. Pertanto, quando la funzione restituita viene invocata, la funzione `saluta()` (la sua funzione esterna) ha terminato la sua esecuzione e quindi il suo contesto di esecuzione non esiste più. Nonostante ciò è ancora possibile accedere alla variabile `nc` presente nel suo scope locale.

JavaScript

Funzioni: con valori di default

- ▶ Dal ECMAScript 6 viene introdotta la possibilità di specificare dei valori di default:

```
function somma(x = 0, y = 0) {  
    let z = x + y;      return z;  
}
```

- ▶ Così, se al momento della chiamata non viene passato un argomento, ad esso viene assegnato il valore di default specificato, invece del valore undefined. Quindi, ad esempio, la chiamata somma() senza argomenti restituirà il valore 0 anzichè NaN.

```
somma(3) //x=3 e y=0
```

```
somma(undefined, 5) //x=0, y=5
```

JavaScript

Funzioni: con valori di default

- ▶ Senza i valori di default quando NON vengono passati dei parametri questi sono undefined

```
function myFunction(x, y) {  
    if (y === undefined) {  
        y = 0;  
    }  
    return x + y;  
}
```

myFunction(3)//3 perché y è undefined

JavaScript

Funzioni: array arguments

- ▶ Si può non definire alcun argomento nella definizione di una funzione ed accedere ai valori passati in fase di chiamata tramite un array predefinito: `arguments`
- ▶ Ad esempio, possiamo sommare un numero indefinito di valori con chiamate `somma(1,2)` o `somma(1,2,3)...`

```
function somma() {  
    var z = 0;      var i;  
    for (i in arguments) {  
        z = z + arguments[i];  
    }  
    return z;  
}
```



JavaScript

Funzioni: rest parameter

```
function eseguiOperazione(x, ...y) {  
    var z = 0;  
    switch (x) {  
        case "somma":  
            for (i in y) {  
                z = z + y[i];  
            } break;  
        case "moltiplica":  
            for (i in y) {  
                z = z * y[i];  
            } break;  
        default: z = NaN; break;  
    }  
    return z;  
}
```

JavaScript

Funzioni: rest parameter

- ▶ L'argomento x che rappresenta il nome dell'operazione da eseguire e y preceduto da tre punti che rappresenta il resto dei valori da passare alla funzione, che saranno disponibili sotto forma di vettore
- ▶ L'approccio è simile all'array predefinito arguments, ma mentre questo cattura tutti gli argomenti della funzione, il rest parameter cattura soltanto gli argomenti in più rispetto a quelli specificati singolarmente.

JavaScript

Funzioni: spread operator

- ▶ La stessa notazione del rest parameter può essere utilizzata nelle chiamate a funzioni che prevedono diversi argomenti. In questo caso si parla di **spread operator**, cioè di un operatore che sparge i valori contenuti in un array sugli argomenti di una funzione, come nel seguente esempio:

```
const addendi = [8, 23, 19, 72, 3, 39];  
somma(...addendi);
```

- ▶ La chiamata con lo spread operator è equivalente alla seguente chiamata:

```
somma(8, 23, 19, 72, 3, 39);
```

JavaScript

Funzioni di prima classe

- ▶ Si dice che un linguaggio di programmazione ha **funzioni di prima classe** quando le funzioni in quel linguaggio sono trattate come qualsiasi altra variabile. Ad esempio, in un tale linguaggio, una funzione può essere passata come argomento ad altre funzioni, può essere restituita da un'altra funzione e può essere assegnata come valore a una variabile.
- ▶ JavaScript ha funzioni di prima classe.

JavaScript

Espressione Funzione (Function expression)

Ad una variabile si può assegnare un'espressione funzione (con let avrà uno scope se no è globale).

```
[const|let] a = function ([par1,...,parN]) {  
    [let local1,...]  
    ...  
    [return (espressione);]  
}; //è consigliato mettere il ;
```

Si richiamerà così: a ()

JavaScript

Espressione Funzione

- Ad a viene passato tutto "il codice della funzione".

```
let a = sayHi;  
function sayHi() {  
    console.log( "Hello" ); }
```

```
console.log( a ); //shows the function code
```

- Con var, const e let non possono essere richiamate prima della loro dichiarazione

```
sayHi("John"); // error!  
let sayHi = function(name) {  
    console.log(`Hello, ${name}`); }
```

JavaScript

Funzioni di callback

```
function ask(question, yes, no) {  
    if (confirm(question)) yes()  
    else no(); }  
  
function showOk() {  
    alert( "You agreed." ); }  
  
function showCancel() {  
    alert( "You canceled." ); }  
  
ask("Do you agree?", showOk, showCancel);
```

I parametri `showOk`, `showCancel` sono dette **funzioni di callback**, in quanto passiamo una funzione come parametro e ci aspettiamo che sarà eseguita in seguito (non al momento del passaggio dei parametri)

JavaScript

Funzione anonime

```
function ask(question, yes, no) {  
    if (confirm(question)) yes()  
    else no();  
}  
  
ask( "Do you agree?",  
    function() {alert("You agreed.");} ,  
    function() {alert("You canceled.");}   
);
```

Non sono richiamabili da nessuna altra parte
perchè non hanno nome

JavaScript

Funzione freccia

```
let func =  
  (arg1, arg2, ...argN) => expression;
```

- ▶ Si crea una funzione anonima che ha gli argomenti dichiarati nelle tonde e restituisce il valore dell'espressione a destra della freccia=>. Corrisponde a:

```
let func =  
  function(arg1, arg2, ...argN) {  
    return expression; };
```

- ▶ Si usano soprattutto come funzioni di callback.
- ▶ Non possono essere usate per definire metodi di un oggetto

JavaScript

Funzione freccia

- ▶ Se ha un solo argomento le () si possono omettere

```
const double = n => n * 2;
```

- ▶ Se non ne ha si mette ()

```
const sayHi = () => console.log ("Hello!");
```

- ▶ Se contiene più di un'istruzione si usano {} e l'istruzione return

```
const sum = (a, b) => {  
    let result = a + b;  
    return result;  
};  
console.log( sum(1, 2) ); // 3
```

JavaScript

Funzione freccia

Esempio di uso come funzioni di callback anonime

```
let age = prompt("What's your age?", 18);  
let welcome = (age < 18) ?  
  () => alert('Hello') :  
  () => alert("Greetings!");  
welcome();
```

JavaScript

Funzione freccia: limiti

- ▶ Non hanno un loro this, per cui non si usano nei metodi di un oggetto

```
let skills = {  
    stamina: 9,  
    decreaseStamina: () => {  
        this.stamina--;  
    }  
}
```

Darà un errore perché non si può accedere al valore "this.stamina"

Inoltre le arrow functions non possono essere utilizzate in combinazione con l'operatore "new" e quindi come costruttori.

JavaScript

Espressione Funzione vs freccia

Function literals

(a.k.a. *anonymous functions*, *function expressions*,
lambda expressions)

```
const multiply = Function(a, b) {  
    return a * b;  
};
```

They make certain coding patterns easier...

Arrow functions ES2015

```
const logValue = x => {  
    console.log(x);  
    return x;  
};  
  
const sum = (a, b) => { return a + b; };  
const mult = (a, b) => a * b; // implicit return
```

A less verbose literal function syntax...

JavaScript

Funzioni self-invoking

- ▶ Una funzione può dichiararsi e contemporaneamente chiamarsi se termina con ()
- ▶ Nel caso di espressioni funzione. Non si può più chiamare con foo()

```
let foo=function () {  
    let x = "Hello!!"; }();
```

- ▶ Nel caso di una funzione anonima

```
(function () {  
    let x = "Hello!!";      // I will invoke  
    myself  
} )();
```

- ▶ Non si può fare con la dichiarazione classica

JavaScript

try..catch

Serve per gestire un errore. La clausola finally viene eseguita in ogni caso, anche se c'è return

```
try {  
    foo()  
} catch(err) {  
    alert(err.message) ;  
    return 10 ;  
} finally {  
    alert("tutto ok") ;  
}
```

Per generare un errore si utilizza l'istruzione

```
throw "Errore" //sarà assegnato a err
```

...

```
catch(err) {  
    alert(err) ; //visualizza "Errore"
```

JavaScript

Istruzioni

► Blocchi di istruzioni

```
{  
    x=2;  
    y=6;  
}
```

► Assegnazione x=2

```
x=2;  
y= ( z=0 ) ;          //sia y che z varranno 0
```

► Assegnazione condizionale

```
x = ( (Cond) ? valVero : valFalso );
```

Esempio

```
x= ( (y>0) ?2:3) ;    //se y>0,x=2 se no x=3
```

JavaScript

Istruzioni

- ▶ Operatori su stringhe (per string, null, caratteri speciali, oggetti)
 - ▶ `s=s+"ciao";` concatenazione anche in forma compatta
`s+="ciao"`
- ▶ Caratteri speciali
 - ▶ `\f` avanzamento pagina
 - ▶ `\n` inizio riga
 - ▶ `\t` tabulazione
 - ▶ `\\" back slash`
 - ▶ `\u00E8` \u seguito dal codice Unicode (nell'esempio è)
- ▶ Operatori polimorfi
 - ▶ `+ += == != =` se operano tra tipi diversi si ha prima una conversione e poi si esegue l'operazione. Si da precedenza ai tipi string

JavaScript

Istruzioni

- ▶ Operatori numerici (per int, float e bool)
- ▶ - + * / %(MOD) ++ --
- ▶ $x/y;$ restituisce un float
- ▶ $x = parseInt(x/y);$ equivale $x=x \text{ DIV } y$
- ▶ $\text{Math.floor}(0.7);$ arrotonda per difetto $\rightarrow 0$
- ▶ $\text{Math.ceil}(0.2);$ arrotonda per eccesso $\rightarrow 1$
- ▶ $\text{Math.round}(0.7);$ arrotonda
- ▶ $\text{Math.random}();$ restituisce un numero tra [0..1)
- ▶ $\text{Math.sqr}();$
- ▶ $\text{Math.min}(x,y);$ restituisce il minore tra i due
- ▶ $\text{Math.max}(x,y);$ restituisce il maggiore tra i due
- ▶ $+"2"$ + unario converte in numero $\rightarrow 2$
- ▶ $2^{**}3$ 2^3
- ▶ Le operazioni errate sui numeri non bloccano mai l'esecuzione (assegnano NaN al risultato)

JavaScript

Istruzioni

Come in C:

-= *=, /= ...

- ▶ L'assegnamento restituisce il valore assegnato

```
let a = 1;  
let b = 2;  
let c = 3 - (a = b + 1);  
alert( a ); // 3  
alert( c ); // 0
```

JavaScript

Istruzioni

► Operatori logici

► || && ^ ! //OR AND XOR NOT

► Operatori relazionali

- <, > <= ==, !=, <= (anche per stringhe)
- ===, !== (ug. e dis stretta senza conversione di tipi: due espressioni vengono considerate uguali soltanto se sono dello stesso tipo e rappresentano effettivamente lo stesso valore.)

```
X = 5;
```

```
X == '5'; //true
```

```
X === '5' //false
```

JavaScript

Istruzioni Condizionali

► IF

```
if (cond)
    bloccoVero;
[else if
    bloccoFalso;]
[else      bloccoAltro;]
```

► Case

```
switch (espressione)
{case cost1:blocco1;
 [break;]
[case cost2:blocco2;]
 [break;]
[default: blocco;]
}
```

JavaScript

Istruzioni Condizionali

► While

```
while (cond)  
    Blocco;
```

► Do while (repeat che cicla per vero)

```
do  
    Blocco;  
while (cond);
```

- break; //esce dal blocco
- continue ; //ricomincia il blocco di un ciclo

JavaScript For

► For

```
for(esprIniz;cond;passo;  
    blocco;
```

si esegue esprIniz, si verifica cond, se è vera si esegue blocco. si esegue passo e si ricomincia verificando cond ...

Per lavorare più comodamente con gli array JavaScript prevede due varianti del for:

- `for...in` per gli oggetti e i vettori
- `for...of` per le collezioni come: Array, String,...

JavaScript

For ... in

- ▶ For in scorre le proprietà di un oggetto (in un vettore sono gli indici)

```
let quantita = [12, 34, 45, 7, 19];
let totale = 0;
for (let indice in quantita) {
    totale = totale + quantita[indice];
}
```

Non bisogna specificare la lunghezza dell'array né l'istruzione di modifica della condizione. JavaScript rileva che la variabile quantita è un array ed assegna ad ogni iterazione alla variabile indice il valore dell'indice corrente (è una stringa però).

JavaScript

For ... of

- ▶ For of scorre i valori presenti in oggetti iterabili ovvero collezioni come: Array, Map, Set, String,...

```
let quantita = [12, 34, 45, 7, 19];  
let totale = 0;  
for (let valore of quantita) {  
    totale = totale + valore;  
}
```

- ▶ Ad ogni iterazione JavaScript assegna alla variabile valore il contenuto di ciascun elemento dell'array.
- ▶ Fa parte delle specifiche di ECMAScript 6 e potrebbe non essere disponibile in engine JavaScript meno recenti.

JavaScript With

► With

```
with (Math) {  
    floor(random() * quanti) + aPartireDa;  
}
```

Attenzione non usare dentro il `with (Math)` variabili con lo stesso nome delle funzioni di `Math`(es `min`, `max`,...) se no non funziona

JavaScript

Istruzioni di Input

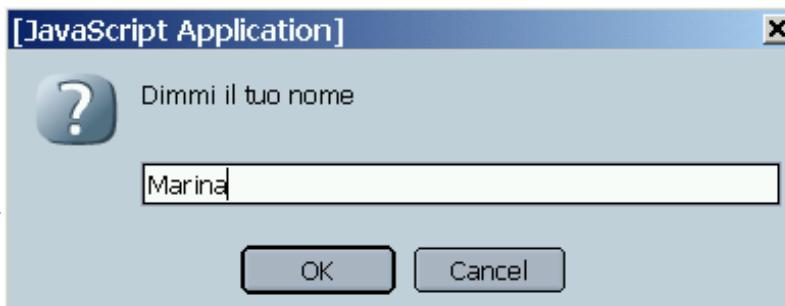
Finestre di dialogo: possono essere modali (bloccanti) se non permettono la prosecuzione dell'esecuzione del resto del codice fino alla loro chiusura, oppure non modali (non bloccanti)

```
x=window.prompt("Dimmi il tuo nome", "");
```

Anche solo `prompt()`, il secondo parametro è il valore di default. Apre una finestra modale. Restituisce:

- ▶ se si introduce qualcosa e si preme OK, restituisce sempre una stringa anche se si introducono numeri
- ▶ `null` se preme ANNULLA
- ▶ stringa vuota se non si inserisce nulla e non c'è un valore di default

Si può quindi valutare `var==undefined` oppure `==null`

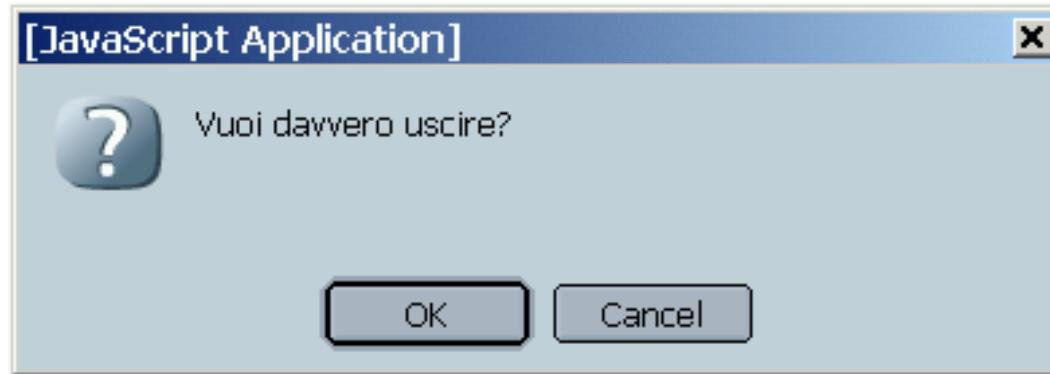


JavaScript

Istruzioni di Input

```
window.confirm("Vuoi davvero uscire?")
```

apre una finestra modale che restituisce true (OK) o false (CANCEL)



JavaScript

Istruzioni di Output

```
window.alert("Hello world");
```

apre una finestra modale che restituisce undefined

Anche solo alert()

```
window.status ("Hello world");
```

viene scritto nella barra di stato

```
console.log("Hello world");
```

viene scritto a console durante il debug

JavaScript

Istruzioni di Output

```
document.write("Hello world");
```

viene scritto nel flusso del documento HTML al momento dell'esecuzione

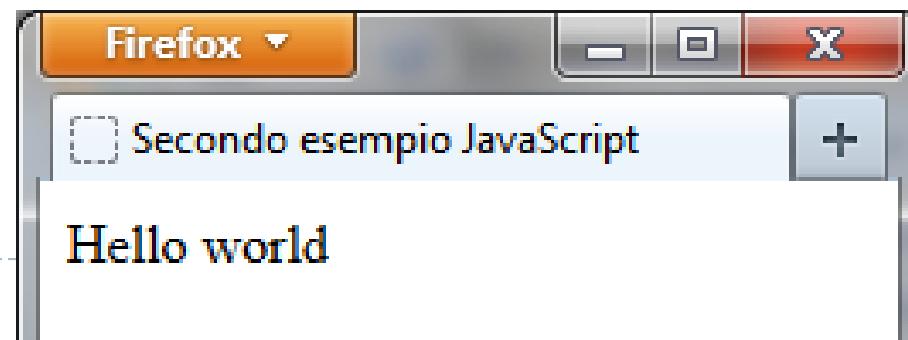
.writeln mette un a capo nel documento (non mette un
! Usa <pre>)

Con .open() e .close()

```
document.open();
```

```
document.write("Hello World");
```

```
document.close();
```



Oggetti in Javascript

di Roberta Molinari

JavaScript

Oggetti

Javascript gestisce oggetti:

- ▶ definiti dall'utente
- ▶ integrati (o interni): gli oggetti di base
 - Array
 - Date
 - String
 - Math
 - Number
 - Boolean
 - Function
- ▶ riflessi dal browser o browser dipendenti BOM:
forniti dall'ambiente del browser (*window, document, location, status, history, navigator*)
- ▶ riflessi dall'HTML DOM: sono gli elementi del documento HTML



Oggetti definiti dall'utente

di Roberta Molinari

JavaScript

Oggetti definiti dall'utente

- ▶ Javascript è un **linguaggio object based** (si possono usare gli oggetti), ma non object oriented (non ci sono classi, un oggetto non è istanza di una classe non c'è una vera ereditarietà) **basato sui prototipi**.
- ▶ **Un oggetto è una collezione di proprietà**, cioè di elementi caratterizzati da un nome. Una sorta di array associativo che è possibile costruire e modificare dinamicamente
- ▶ Si possono creare oggetti in vari modi
 - con la notazione letterale {nome:val,...}
 - definendo una funzione costruttore
 - Con il costrutto class

JavaScript

Oggetti {chiave: valore} – object literal

- ▶ Per creare un oggetto con la notazione letterale si definiscono le sue proprietà come chiave: valore (hashmap)

```
var person = { //anche su una riga sola
    "first Name" : "John",
    lastName : "Doe",
    age : 50
};
```

- ▶ Una proprietà può assumere qualsiasi valore, compreso un altro oggetto (come le strutture in C)

```
var persona = {
    nome: "Mario",
    cognome: "Rossi",
    indirizzo: {via: "Via Garibaldi", num: 15}
};
```

JavaScript

come accedere alle proprietà di un oggetto

▶ Dot Notation

person.name

person.introduceSelf()

person.address?.city // Optional chaining

▶ Bracket Notation

person['name']

person['introduceSelf']()

const nome = 'name';

person[nome];



JavaScript

Oggetti: funzioni costruttori - Constructor function

- ▶ Un costruttore è una normale funzione invocata con l'operatore new

```
function Persona() { //typeof function  
    this.nome="Mario";  
    this.cognome="Rossi";  
    let x = 5;      //privata  
}
```

this è una parola chiave (non una variabile) che indica l'oggetto che possiede il codice. La parola this dichiara una proprietà public, se non c'è è un attributo privato

```
let mr = new Persona();  
mr.nome="Mario"; mr.cognome="Rossi"; mr.eta=50;
```

JavaScript

Oggetti: funzioni costruttori

```
function User(name) {  
    this.name = name;  
    this.isAdmin = false;  
}
```

In altre parole, `new User(...)` fa qualcosa del genere:

```
function User(name) {  
    // this = {};  (implicito)  
  
    // aggiungiamo proprietà a this  
    this.name = name;  
    this.isAdmin = false;  
  
    // return this;  (implicito)  
}
```

JavaScript

Oggetti: funzioni costruttori

- ▶ Possiamo prevedere dei parametri nel costruttore

```
function Persona(n, c) {  
    this.nome= (n) ;  
    this.cognome= (c) ;  
}
```

- ▶ ATTENZIONE! se nella creazione dell'oggetto omettiamo new, quello che otterremo non sarà la creazione di un oggetto ma l'esecuzione della funzione, con risultati imprevedibili.
- ▶ Con new typeof → object
- ▶ Senza new è undefined

JavaScript

Oggetti: funzioni costruttori

- Possiamo prevedere dei valori iniziali in modo diverso da quello previsto con parametri di default che si scrive : `function Persona(n="", c="")`

```
function Persona(n, c) {  
    this.nome=(n || "");      //se n è false->""  
    this.cognome=(c || "");  
}
```

e in questo caso nella creazione si può fare in 2 modi

```
//con undefined, come con parametri di default  
p1 = new Persona(undefined, "Rossi"); //nome→""  
//con null, non possibile con par di default  
p2 = new Persona(null, "Rossi"); //nome→",
```

JavaScript

Oggetti: proprietà

- ▶ I nomi delle proprietà non hanno le restrizioni dei nomi delle variabili, ma se le infrangono vanno tra ""

```
{ "primo-nome": "Mario",  
  "secondo.nome": "Rossi" }
```

- ▶ Per accedere alle proprietà degli oggetti si usa la dot notation o la notazione degli array associativi (obbligatoria se i nomi non seguono le regole dei nomi delle variabili)

```
a = oggetto.nome_proprietà  
a = oggetto["nome_proprietà"]  
x = "nome_proprietà";  
a = oggetto[x]
```

JavaScript

Oggetti: proprietà

- ▶ Per ottenere tutte le proprietà si utilizza il for-in

```
for (campo in p1) {  
    alert( campo + " " + p1[campo] )  
    //NON p1.campo  
}
```

- ▶ Si può anche usare il metodo **forEach** molto più veloce del precedente, ma itera solo sulle proprietà dell'oggetto non ereditate dal prototipo

```
Object.keys(p1).forEach(function(campo) {  
    alert(campo + " : " + p1[campo] ) }  
);
```

JavaScript

Oggetti: proprietà

- ▶ Si possono aggiungere proprietà pubbliche dinamicamente che apparterranno solo all'oggetto

```
persona.hobby = "scacchi"
```

- ▶ Si possono anche eliminare dinamicamente le proprietà ai singoli oggetti (Dopo l'eliminazione della proprietà, ogni tentativo di accesso ad essa restituirà il valore `undefined`)

```
delete persona.hobby
```

JavaScript

Oggetti: metodi

- ▶ Un metodo non è altro che una funzione assegnata ad una proprietà. Si può aggiungerlo direttamente all'oggetto

```
const persona = {  
    nome : "Mario", cognome : "Rossi",  
    visualizza : function () {  
        return this.nome + this.cognome  
    } }
```

- ▶ Si possono aggiungere dinamicamente metodi pubblici

```
pers1.fun=function () {} //solo a pers1
```

JavaScript

Oggetti: metodi

- ▶ Si può aggiungerlo alla funzione costruttore

```
function Persona (n, c) {  
    this.nome = n;  
    this.cognome = c;  
    this.visualizza = function () {  
        return this.nome + this.cognome  
    }  
}
```

- ▶ Se dichiaro al suo interno una funzione senza il this, è una funzione NON un metodo

- ▶ Per richiamarlo

```
var pers1=new Persona ("Mario", "Rossi")  
alert(pers1.visualizza())
```

JavaScript

Oggetti: metodi

- ▶ I metodi si possono anche dichiarare così
 1. Si definisce separatamente la funzione
 2. Si assegna alla classe nello stesso modo che una proprietà

```
function visNomeCogn() {  
    return this.nome+this.cognome}  
  
Persona = function(n,c) {  
    this.nome=n;      this.cognome=c;  
    this.visualizza = visNomeCognome //senza()  
}
```

non assegniamo alla proprietà il risultato della chiamata alla funzione, ma la funzione stessa tramite il suo nome. La proprietà visualizza, dal momento che contiene una funzione, è di fatto un metodo.

JavaScript

Oggetti: destructuring

- ▶ Il destructuring è un'espressione JavaScript che consente di separare le proprietà di un oggetto (o i valori di un array) in variabili distinte.

```
const user = {  
    id: 42,  
    isVerified: true  
};
```

```
const {id, isVerified} = user;
```

```
console.log(id); // 42  
console.log(isVerified); // true
```

JavaScript

Oggetti: new Object e oggetti vuoti

- ▶ Per creare un oggetto sfruttando l'oggetto Object, che è alla base di qualsiasi oggetto JavaScript, si usa `new Object()`

```
var persona = new Object ({  
    nome: "Mario",  
    cognome: "Rossi"  
}) ;
```

- ▶ Per creare oggetti vuoti (senza proprietà)

```
var auto = new Object () ; //typeof object
```

Equivale a

```
var auto = { } ;
```



JavaScript

Oggetti: classi → prototipo

- ▶ Il linguaggio Javascript supporta il concetto di ereditarietà attraverso l'uso dei prototipi (objects prototypes). **Un prototipo è un oggetto in cui vengono definiti un insieme di proprietà e metodi che si desidera vengano condivise da tutte le istanze di un determinato tipo di oggetto.**
- ▶ Tutte le funzioni derivano da Function.prototype, gli array da Array.prototype. In cima alla gerarchia c'è Object.prototype che contiene metodi e proprietà comuni a tutti gli oggetti tra cui toString()

JavaScript

Oggetti: prototipi

- ▶ Per aggiungere una proprietà/metodo a tutte le istanze si usa `prototype` (simula l'ereditarietà)

```
Persona.prototype.hobby="scacchi"
```

```
Persona.prototype.fun=function () {...}
```

- ▶ La nuova proprietà non è direttamente agganciata a ciascun oggetto, ma accessibile come se fosse una sua proprietà. Il prototipo di un oggetto è una sorta di riferimento ad un altro oggetto.

JavaScript

Oggetti: prototipi ed eredità

- ▶ Il meccanismo su cui si basa l'ereditarietà prototipale fa sì che, se una proprietà non si trova in un oggetto, viene cercata nel suo prototipo.
- ▶ Il prototipo di un oggetto può a sua volta avere un altro prototipo. In questo caso la ricerca di una proprietà o di un metodo risale la catena dei prototipi fino ad arrivare all'oggetto Object, il prototipo base di tutti gli oggetti.
- ▶ Anche gli oggetti predefiniti di JavaScript hanno un prototipo di riferimento, perciò si possono modificare.
- ▶ Se si cancella la proprietà di un oggetto, non si cancella dal suo prototipo, ma se la cancelli dal prototipo si cancella da tutti gli oggetti (o modifichi)

JavaScript

Oggetti: prototipi

```
function Animale(verso){  
    this.verso=verso;  
    this.parla=function(){console.log(this.verso+'\n');};  
}  
  
function Cane(){;}  
  
Cane.prototype = new Animale('BAU BAU');  
  
function Gatto(){  
    this.faFusa=function(){console.log('PRRR\n');}; }  
  
Gatto.prototype = new Animale;  
  
Gatto.prototype.verso='MIAO';  
  
var v= []; v.push(new Gatto()); v.push(new Cane());  
for(var i=0;i<v.length;i++){  
    v[i].parla();  
    if (v[i] instanceof Gatto) v[i].faFusa();  
}
```

JavaScript

Oggetti: prototipi

Si può fare anche con gli oggetti letterali

```
let animal = {  
    eats: true,  
    walk() {  
        alert("Animal walk");  
    }  
};
```

```
let rabbit = {  
    jumps: true,  
    __proto__: animal  
};
```

```
// walk viene ereditato dal prototype  
rabbit.walk(); // Animal walk
```

JavaScript

Oggetti ES6: class

```
class MyClass {  
    prop = value; // proprietà sarà nell'oggetto istanziato non nel prototipo  
    #propPrivata  
    constructor(...) { // costruttore  
        // ...  
    }
```

				
Chrome 49	Edge 12	Firefox 45	Safari 9	Opera 36
Mar, 2016	Jul, 2015	Mar, 2016	Oct, 2015	Mar, 2016

```
method(...) {} // metodo
```

```
get something(...) {} // metodo getter
```

```
set something(...) {} // metodo setter
```

```
toString(){} //return una stringa
```

```
}
```

- ▶ **MyClass** è tecnicamente una funzione (che corrisponde a `constructor`), mentre i metodi vengono scritti in `MyClass.prototype`.



JavaScript

Oggetti ES6: class

```
// Declaration  
class Quadrato { constructor(lato) {this.lato = lato;}}  
// Expression; the class is anonymous but assigned to a  
// variable  
const Quadrato = class {  
    constructor(lato) {this.lato = lato;}  
};  
// Expression; the class has its own name  
const Quadrato = class Quad {  
    constructor(lato) {this.lato = lato;}  
};
```

In ogni caso Quadrato è una "function" e per instanziare un oggetto si fa

```
const q = new Quadrato(10);
```

JavaScript

Oggetti ES6: class

- ▶ I metodi e gli attributi sono automaticamente pubblici. Per dichiararli private si usa `#nome` (non può esserci un costruttore private) e devono essere dichiarati prima dell'uso
- ▶ Prima si usava per convenzione `_nome`, ma era comunque visibile dall'esterno.
- ▶ Per richiamare i metodi `get` e `set`:

get: si invoca come una proprietà (senza `()`)

```
get name() { ... }  
console.log(obj.name);
```

set: si invoca con un assegnamento alla proprietà (senza `()`)

```
set name(value) {...}  
obj.name =
```

ATTENZIONE! In un contesto di ereditarietà preferire
`getName()` e `setName(value)` alla Java



JavaScript

Oggetti ES6: class

```
class User {  
    #eta = 10;    #name;  
    constructor(name) { this.#name = name; }  
    get name() { return this.#name; }  
    set name(value) {  
        if (value.length < 4) {  
            console.log("Nome troppo corto");  
        } else this.#name = value;  
    }  
}  
  
let user = new User("John"); //non invoca il set  
console.log(user.name); // invoca il get -> John  
user.name = "Rob"// invocato il set  
console.log(user.name)  
// invoca il get -> Nome troppo corto e non cambia
```

JavaScript

Oggetti: class

`class` non introduce semplicemente una semplificazione sintattica (syntax sugar) per dichiarare una funzione costruttore:

1. Una funzione creata attraverso `class` viene etichettata dalla speciale proprietà interna `[[IsClassConstructor]]: true`. Quindi non è esattamente uguale che crearla manualmente.
2. A differenza di una normale funzione, il costruttore di una classe può essere richiamato solo attraverso la parola chiave `new`
3. I metodi delle classi non sono numerabili. La definizione di una classe imposta il flag `enumerable` a `false` per tutti i metodi all'interno di "prototype". Questo è un bene, dato che non vogliamo visualizzare i metodi quando utilizziamo un ciclo `for .. in` per visualizzare un oggetto. Si visualizzeranno solo le proprietà pubbliche
4. Il contenuto di una classe viene sempre eseguito in `strict`.

JavaScript

Oggetti ES6: ereditarietà

```
class Rabbit extends Animal {
  hide() {
    alert(`\$ {this.name} hides!`);
  }

  stop() {
    super.stop(); // richiama il metodo stop() dal padre
    this.hide(); // and then hide
  }
}

let rabbit = new Rabbit("White Rabbit");

rabbit.run(5); // White Rabbit runs with speed 5.
rabbit.stop(); // White Rabbit stands still. White Rabbit
               hides!
```

JavaScript

Oggetti ES6: ereditarietà

Secondo le specifiche, se una classe ne estende un'altra e non ha un suo metodo constructor viene generato il seguente constructor “vuoto”:

```
class Rabbit extends Animal {  
    // generato per classi figlie senza un costruttore  
    // proprio  
    constructor(...args) {  
        super(...args);  
    }  
}
```

JavaScript

Oggetti ES6: ereditarietà

```
class Animal {  
    constructor(name) {  
        this.speed = 0;  
        this.name = name;  
    } // ...  
}  
  
class Rabbit extends Animal {  
    constructor(name, earLength) {  
        super(name);  
        this.earLength = earLength;  
    } // ...  
}  
  
let rabbit = new Rabbit("White Rabbit", 10);  
alert(rabbit.name); // White Rabbit  
alert(rabbit.earLength); // 10
```

il costruttore genitore utilizza sempre i suoi campi dati: usare get e set alla Java per avere i comportamenti noti.

JavaScript

Oggetti ES6: static ATTENZIONE!

```
class BaseClassWithPrivateStaticField {  
    static #PRIVATE_STATIC_FIELD;  
    static basePublicStaticMethod() {  
        return this.#PRIVATE_STATIC_FIELD;  
    }  
}  
class SubClass extends  
    BaseClassWithPrivateStaticField {}
```

```
SubClass.basePublicStaticMethod();  
// TypeError: Cannot read private member  
#PRIVATE_STATIC_FIELD from an object whose class  
did not declare it
```

Dà errore perchè `this` del metodo si riferisce al `this` di
SubClass e non a `BaseClassWithPrivateStaticField`

JavaScript

Oggetti ES6: static

```
class Counter {  
    static count = 0;  
  
    constructor() {  
        Counter.count++;  
    }  
    static getCount() {  
        return Counter.count;  
    }  
}  
  
const c1 = new Counter();  
console.log(Counter.getCount()); // Output: 1  
  
const c2 = new Counter();  
console.log(Counter.count); // Output: 2
```



JavaScript

For ... in

- ▶ `for in` scorre le proprietà di un oggetto (anche quelle ereditate)

```
var obj = {a: 1, b: 2, c: 3};  
for (let prop in obj) {  
    console.log(`obj.${prop} =  
        ${obj[prop]}`);  
}
```

Restituisce il nome (stringa) della proprietà dell'oggetto

JavaScript

Oggetti: `hasOwnProperty()`

- ▶ `hasOwnProperty` può essere utilizzato per determinare se un oggetto ha la proprietà specificata come proprietà diretta di tale oggetto; a differenza dell'operatore `in`, questo metodo non controlla una proprietà nella catena di prototipi dell'oggetto.

```
let buz = { fog: 'stack' };  
for (let name in buz) {  
    if (buz.hasOwnProperty(name))  
        console.log(buz[name]);  
    else  
        console.log(name);  
}
```

JavaScript

Oggetti: array di oggetti

```
let T = [];  
  
//caricamento vettore  
T.push({prd:'vite', prx:2});  
T.push({prd:'dato', prx:1});  
T.push({prd:'ruota', prx:5});  
  
for(let i=0; i<T.length; i++)  
    P.innerHTML += T[i].prd + ' ' + T[i].prx + '<br>;
```

JavaScript

Oggetti: array di oggetti

```
let T = [];
function pro(x,y,z) {
    this.prd = x;    this.prx = y;  this.qta = z;
    this.print = function() {
        return this.prd + ' ' + this.prx  };
}
//caricamento vettore
T.push(new pro('vite',2,50));
T.push(new pro('dato',1,72));
T.push(new pro('ruota',5,25));

for(let i in T)
    P.innerHTML += T[i].prd + ' ' + T[i].qta + '<br>';
P.innerHTML += '<hr>';
for(let el of T)
    P.innerHTML += el.print()+'<br>';
```

JavaScript

Funzioni come oggetti

- ▶ In Javascript, anche se il `typeof` su funzioni restituisce `function`, sono descrivibili come oggetti con metodi e proprietà
- ▶ `arguments.length`: proprietà = al numero di argomenti
- ▶ `toString()`: applicato alla funzione restituisce la funzione come stringa

```
function myFunction(a, b) {  
    return a * b;  
}  
var txt = myFunction.toString();
```

Javascript

Trasmissione dei dati

- ▶ I dati viaggiano attraverso la rete sempre in formato stringa. Quando un client riceve una stringa di dati dalla rete, per poterli elaborare li deve trasformare in oggetto. Quando invece deve trasmettere un oggetto in rete, prima di trasmetterlo lo deve trasformare in stringa.
- ▶ Il processo di trasformazione di una stringa in un oggetto si definisce **de-serializzazione** o **parsificazione (parsing)**
- ▶ Il processo inverso di trasformazione di un oggetto in stringa si definisce **serializzazione**

Javascript

JSON

- ▶ **JSON** (JavaScript Object Notation) è un formato adatto per lo scambio di dati e la serializzazione. Si basa su un sottoinsieme del linguaggio di programmazione JavaScript.
- ▶ È un formato di testo completamente indipendente dal linguaggio di programmazione, ma utilizza convenzioni conosciute dai programmatore di linguaggi della famiglia del C
- ▶ Non è possibile salvare tipi Date o metodi
- ▶ Validatore JSON on line <https://jsonlint.com/>

JSON

È basato su due strutture supportate da tutti i linguaggi di programmazione moderni (così da permettere un interscambio di dati):

1. Un insieme di coppie **nome/valore**. (Realizzato nei vari linguaggi come un oggetto, un record, uno struct, un dizionario, una tabella hash, un elenco di chiavi o un array associativo)
2. Un **elenco ordinato** di valori. (Nella maggior parte dei linguaggi questo si realizza con un array, un vettore, un elenco o una sequenza)

In JSON queste strutture sono implementate con:

1. **oggetti**
2. **array**

JSON

oggetti

- ▶ Un **oggetto** è una serie non ordinata di *nomi/valori*. Un oggetto inizia con { (parentesi graffa sinistra) e finisce con } (parentesi graffa destra).
- ▶ Ogni **nome** è una stringa racchiusa da " seguito da : (due punti) e la coppia di *nome/valore* sono separate da , (virgola).

JSON

oggetto

object

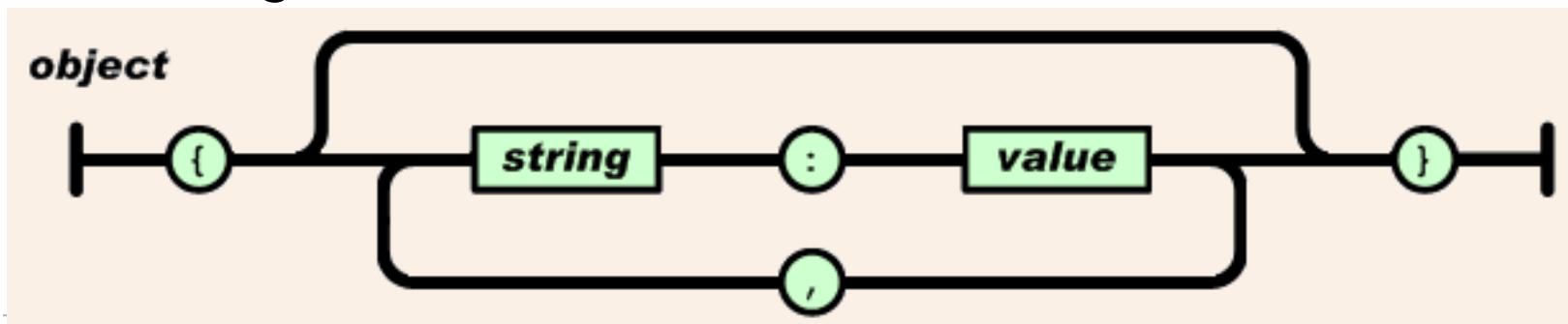
```
{  
  { members }  
}
```

members

pair
pair , members

pair

string : value



array

- ▶ Un **array** è una raccolta ordinata di valori. Comincia con [e finisce con]. I valori sono separati da , (virgola).
- ▶ Un **valore** può essere una stringa tra virgolette doppie, o un numero intero o a virgola mobile, o vero o falso o nullo, o un oggetto o un array. Queste strutture possono essere annidate.
- ▶ In JavaScript un attributo di un oggetto poteva anche essere una funzione, una data o undefined. Inoltre le stringhe possono essere racchiuse anche da '

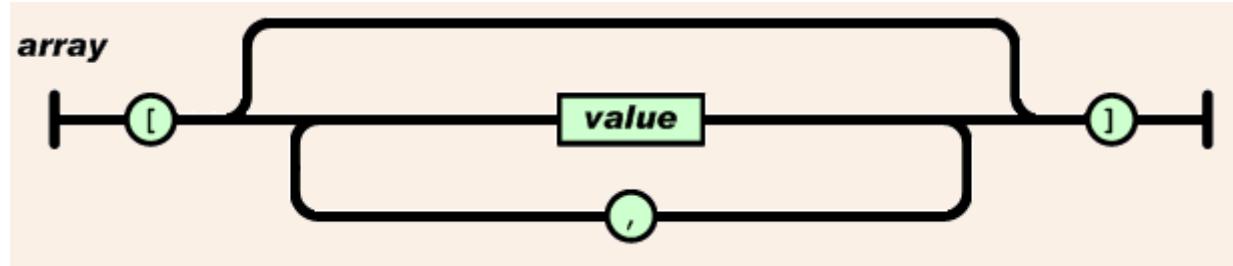
JSON

array

array []

[elements]

elements



value

value , elements

value

string

number

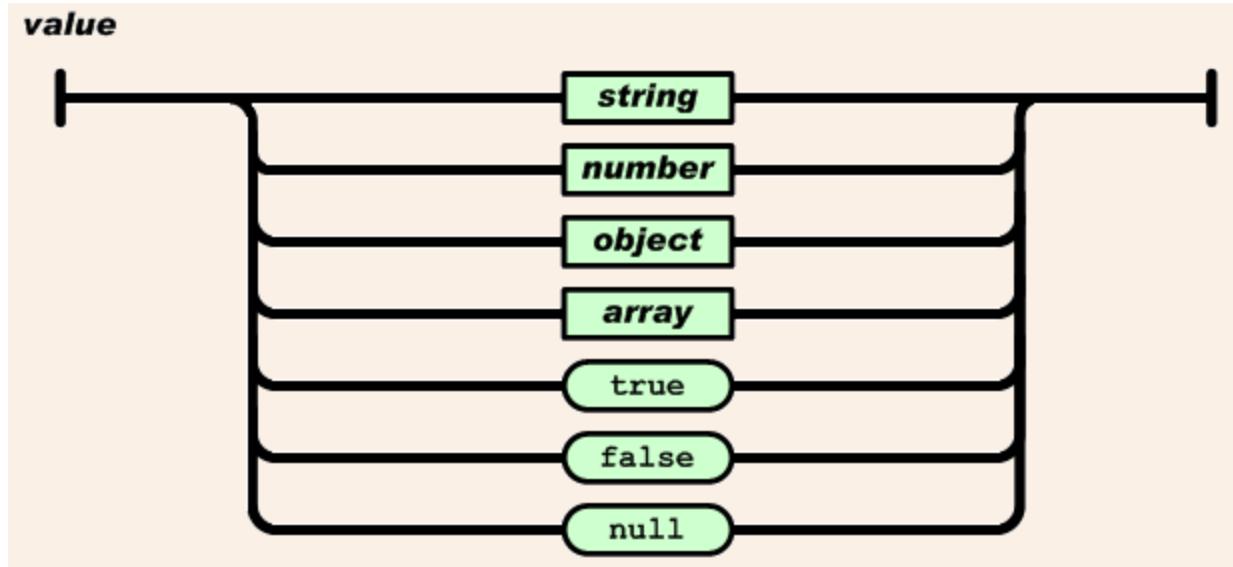
object

array

true

false

null



Javascript oggetti e array Json

- ▶ È possibile creare oggetti javascript da oggetti JSON con il metodo `JSON.parse(string)`

```
const obj = JSON.parse('{"nome": "Ugo",  
"voti": {"o": 8, "s": 7}, "media": 7.5}');
```

- ▶ Le proprietà e le stringhe devono essere scritte tra "" (non si può fare con ')
- ▶ Gli oggetti possono essere anche complessi
- ▶ Analogamente è possibile creare vettori js da array JSON (saranno sempre come `typeof Object`)

```
const vett = JSON.parse('[1,2,3]');
```

Javascript

oggetti e array Json

- ▶ È possibile ottenere la stringa JSON da oggetti/array javascript con il metodo `JSON.stringify(obj)`
- ▶ I metodi non vengono inseriti nella stringa

```
var obj = { name: "John",
            age: function () {return 30;},
            city: "New York"};
var myJSON = JSON.stringify(obj);

//{"name":"John","city":"New York"}
```

JavaScript

Oggetti: copia

- ▶ **Shallow copy:** si copiano tutte le proprietà solo al primo livello (se sono tipi semplici, per tipi ref copiano le referenze!).
 - ▶ Si può fare con **Spread operator**

```
const originalObject = { a: 1, b: 2 };
const copiedObject = {...originalObject};
//{ a: 1, b: 2 }
```

- ▶ o con **Object.assign()**

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };
const returnedTarget = Object.assign(target, source);
// { a: 1, b: 4, c: 5 }
```

Javascript

Copia di oggetti con Object.assign

- ▶ È possibile copiare le proprietà e i metodi di un oggetto con `Object.assign(obj1, obj2, obj3, ...)`
- ▶ Se hanno lo stesso nome, le proprietà e i metodi (non ereditati) del primo oggetto saranno sovrascritte con quelle omonime dei successivi.

```
function Obj1() { this.val = 0, this.funz = () => 0,  
    this.altro = 'resta' }  
function Obj2() { this.val = 10, this.funz = () => 10,  
    this.inPiu = 'nuovo' }  
  
var obj1 = new Obj1(); var obj2 = new Obj2();  
obj1 = Object.assign(obj1, obj2)  
console.log(obj1.funz()) //10  
console.log(obj1)  
//Object{val:10,funz:funz(),altro:"resta",inPiu:"nuovo"}
```

JavaScript

Oggetti: copia

- ▶ **Deep copy:** JSON.parse(JSON.stringify()) nel passaggio
ogg → string → ogg si possono perdere dei dati come le date

```
const copiedObject =  
  JSON.parse(JSON.stringify(originalObject));
```

- ▶ **structuredClone** (recentemente implementato in Chrome98,
<https://developer.mozilla.org/en-US/docs/Web/API/structuredClone>)
- ▶ librerie (es. cloneDeep di Lodash <https://lodash.com/>)

JavaScript ricapitolando

```
var x1 = {} ;           // new object
var x2 = "" ;           // new primitive string
var x3 = 0 ;             // new primitive number
var x4 = false ;          // new primitive boolean
var x5 = [] ;            // new array object
var x6 = /() /           // new regexp object
var x7 = function() {} ;  
                         // new function object
```

Oggetti integrati

di Roberta Molinari



JavaScript

Oggetti integrati

Javascript fornisce i seguenti oggetti integrati:

- **Array**
- **Date**
- **String**
- **Math**
- **Number**
- **Boolean**
- **Function**

- ▶ È sconsigliato istanziare oggetti **Array**, **String**, **Number**, **Boolean** (per es. `new String()`), invece se ne potranno comunque usare i metodi e gli attributi



Funzioni e proprietà globali o di sistema: non appartengono a nessun oggetto specifico, quindi si indicano senza "."

JavaScript

Number

- ▶ Number (**sono sempre floating point a 64 bit**)
- ▶ **operazioni aritmetiche +,-,*,/,% , x++, --x, +=**
- ▶ **Funzioni globali**

-`isNaN(v)` **verifica se v non è un numero**

-`isFinite(v)`, `isInteger(v)` **verifica se v è finito, intero**

-`parseFloat(str)`, `parseInt(str)` **Cercano all'inizio di str un numero decimale o un intero. Se non lo trovano restituiscono NaN.**

`parseFloat(1.3)` → 1.3 `parseInt(1.3)` → 1

-`Number(ogg)` **Converte oggetto in un numero, se possibile. Se no ritorna NaN o =0 se non c'è ogg (es. con "")**



JavaScript

Costanti statiche Number

Costante	Valore restituito
Number.MAX_VALUE	Numero più grande che può essere rappresentato in JavaScript. Equivale a circa 1,79E+308.
Number.MIN_VALUE	Il numero più vicino a zero che può essere rappresentato in JavaScript. Equivale a circa 5,00E-324.
Number.NaN	Valore che non è un numero, ma il suo typeof è Number. Appare come risultato di operazioni che non restituiscono un numero come 0/0 Nei confronti di uguaglianza, NaN non corrisponde ad alcun valore, incluso se stesso. Per verificare se un valore è equivalente a NaN, utilizzare la funzione <code>isNaN</code>.
Number.NEGATIVE_INFINITY	Valore minore del massimo numero negativo che possa essere rappresentato in JavaScript. In JavaScript i valori NEGATIVE_INFINITY vengono visualizzati come <code>-infinity</code>.
Number.POSITIVE_INFINITY	Valore maggiore del massimo numero negativo che possa essere rappresentato in JavaScript, o risultato

JavaScript

Oggetto Math

- ▶ Proprietà **dell'oggetto Math**
 - E **costante di Eulero**
 - PI **pi greco**
- ▶ Metodi
 - abs (val) **valore assoluto**
 - round (val) **arrotondamento**
 - ceil (val) **arrotondamento per eccesso**
 - floor (val) **arrotonda per difetto**
 - random () **numero casuale tra 0 e 1**
 - pow (base, exp) **elevamento a potenza**
 - sqrt (val) **radice quadrata**



JavaScript

Boolean

- ▶ Boolean **possibili valori** true, false

```
var pagato = true;
```

```
var consegnato = false;
```

- ▶ **Operatori logici:** &&, ||, !

JavaScript String

- ▶ String

```
var nome = "Mario"; //anche 'Mario'
```

```
var empty = ""; //vuota Number("")→0
```

```
var empty2 = new String(); //vuota
```

```
var str2 = new String("Altra");
```

- ▶ **Con new String() typeof restituisce Object**
- ▶ **Assegnazione con =**
- ▶ **Si possono usare sia " che '**
- ▶ **Confronto con == !=**
- ▶ **Funzione globale String(oggetto) **converte****
oggetto nella stringa che rappresenta il suo
valore,



JavaScript

String

- ▶ **Concatenazione di stringhe con +**

```
nome = nome + "!" //anche nome +=!"
```

- ▶ **Dalla versione 6 si può usare l'apice inverso (Alt+96 in Windows, AltGr + ' in Linux) e l'inserimento tramite \${} in cui il suo contenuto viene valutato prima della concatenazione**

```
let myPet = 'armadillo'  
alert(`I own a pet ${myPet}.`)
```

JavaScript

String

- ▶ Proprietà **degli oggetti String**
 - `length` **lunghezza della stringa**
- ▶ Metodi **degli oggetti String**
 - `charAt(pos)` **equivale a `s[pos]`, che non si deve fare**
 - `charCodeAt(pos)` **restituisce il codice Unicode di pos**
 - `subString(start, end)` **restituisce i caratteri $\in [start..end]$ (primo 0)**
 - `toUpperCase()` / `toLowerCase()` **restituiscono la stringa modificata, ma non modificano l'oggetto a cui sono applicati**
 - `vett=stringa.split(",")` **usa il carattere specificato come separatore per individuare gli elementi del vettore**
 - `s=str1.concat(" ", str2)` **restituisce la stringa ottenuta dalla concatenazione di `str1+" "+str2`**



JavaScript

String

► Metodi **degli oggetti String**

- `indexOf/lastIndexOf(str[, pos])` **posizione della prima/ultima occorrenza della string str cercata a partire dalla posizione pos facoltativo. -1 se non la trova**
- `search(str)` **come la precedente, ma non può specificare da dove iniziare la ricerca, invece può usare espressioni regolari da cercare.** Es `s.search(/mamma/i);`
- `replace(old, new)` **restituisce la stringa ottenuta sostituendo la prima occorrenza di old con new.** `old` può essere un **espressione regolare** con i parametri `/i (insensitive) /g (sostituzione globale)`



JavaScript

String encode()

- ▶ **encodeURIComponent () è pensata per codificare i valori di eventuali parametri passati in un URI. Codifica una stringa lasciando inalterate cifre, lettere e i caratteri +-*/._@ e rimpiazzando tutti gli altri caratteri con la codifica esadecimale preceduta da un carattere percentuale (%).**
- ▶ **encodeURI () come la precedente, ma esclude dalla codifica i caratteri , /?:@&=+\$#**

```
var param= encodeURIComponent("Che cos'è?");  
var encodedURI = encodeURI("http://www.html.it/a  
b.php?x=") + param;
```

- ▶ **Le funzioni decodeURIComponent(), decodeURI() eseguono il procedimento contrario**

JavaScript

Date

- ▶ Una variabile di tipo Date rappresenta un istante temporale (data ed ora) rappresentante i millisecondi passati dal 1-1-1970 (<0 per antecedenti)

```
var adesso = new Date(); //data e ora corrente  
var natale2012 = new Date("Dec 12,25");  
var fineAnno2013 = new Date(2013,11,31,23,59,00);  
var fineAnno2014 = new Date("2014-12-31T23:59");
```

- ▶ I mesi e i giorni della settimana partono da 0
- ▶ Se aggiungo 1 ai giorni o ai mesi tiene conto degli anni bisestili e aggiorna l'anno (dicembre + 1). Lo stesso con le ore e i minuti.
- ▶ Si possono fare operazioni aritmetiche (considerando il risultato in millsec) a

JavaScript

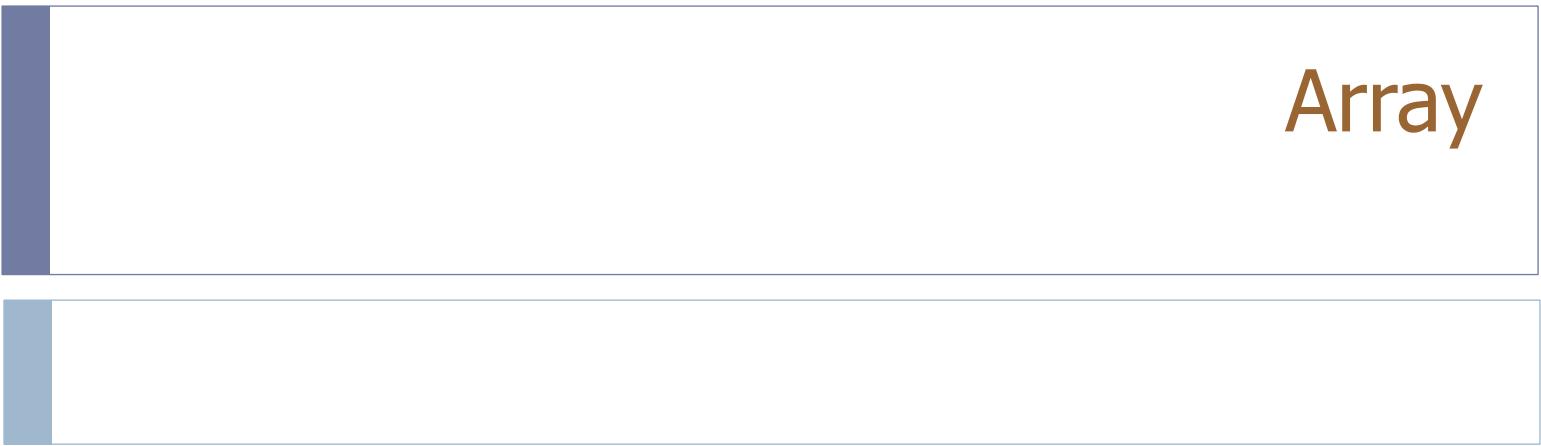
Date

► Metodi degli oggetti Date

- `getXXX ()` restituisce il valore della caratteristica **XXX** della data (es. `getFullYear()`).
- `setXXX (val)` imposta il valore della caratteristica **XXX** della data (es. `setFullYear(2013,1,1)`);
- `toString ()` restituisce la data come stringa

Caratteristica	Significato	Caratteristica	Significato
Date	Giorno del mese	Hours	Ore
Day	Giorno della settimana	Month	Mese
FullYear	Anno	Seconds	Secondi
Minutes	Minuti	Time	millsec dal 1-1-70





Array

Linguaggi WEB lato CLIENT

200

200

JavaScript

Array

- ▶ È un insieme di elementi di tipo arbitrario con primo indice 0 (array *denso*). È un particolare oggetto i cui campi sono reperibili tramite indice (typeof restituisce object). Si definisce come array literal:

```
const v = [];  
const a = [1, 2, 3];  
const b = [10, "ciao", True];
```

- ▶ Oppure si può usare il costruttore Array, ma si preferisce il primo modo

```
const v = new Array(); //vuoto meglio var v=[];  
const w = new Array("Qui", "Quo", "Qua");  
const c = new Array(3); //array di 3 celle undef.  
const d = new Array(3, 4); //array di 2 celle con 3 e 4
```

JavaScript

Array

- ▶ Non è necessario specificare la dimensione
- ▶ typeof restituisce object, per cui per sapere se è un , array, dalla versione 5 si può utilizzare

```
Array.isArray(vet) ;
```

```
vet instanceof Array
```

- ▶ Si può visualizzare in toto

```
alert(v) //compare una lista separata da ,
```

- ▶ Se usati come parametri sono passati per referenza e come par formali si dichiara solo il nome

```
function f(V, N)
```

- ▶ Possono essere restituiti come valore di una funzione da assegnare ad un vettore, anche se non ha senso essendo oggetti fare v1=v2 (diventano puntatori alla stessa area).

JavaScript

Array

- ▶ Se si assegna un valore ad un elemento non presente questo viene creato

```
var v = [];
```

```
v[0] = 1;
```

```
v[3] = 5; //1 e 2 sono undefined
```

- ▶ Se si accede al valore di un elemento non presente si ottiene un valore indefinito

```
var v = [];
```

```
var a = v[0]; //undefined
```

- ▶ Per aggiungere elementi in fondo

```
newLeng = v.push(6);
```

```
//equivale a v[v.length]=6;
```

JavaScript Array

▶ Proprietà degli oggetti Array

- `length` lunghezza del vettore (è di RW), se si cambia il suo valore si cambia la dimensione del vettore

▶ Metodi

- `sort(fConf)` per ordinare il vettore secondo la funzione `fConf` (invia i valori a `fConf` e ordina i valori in base al valore restituito $<0,=0,>0$). Se non c'è `fConf` è ordinato alfabeticamente (secondo ASCII) $1 < A < a$)
 - a) Per ordinare in base ai numeri
- `v.sort(function(a, b) {return a - b})`
- `reverse()` inverte l'ordine degli elementi

JavaScript Array

► Metodi

- `join(c)` unisce i valori in una stringa concatenata con il carattere `c`
- `primoEl = v.shift()` elimina e restituisce il primo elemento
- `newLeng = v.unshift(newPrimoEl)` aggiunge un nuovo elemento al posto 0 e restituisce la nuova lunghezza
- `newLeng = push(val)` e `val = pop()` come i precedenti, ma con l'ultimo elemento
- `delete vett[0]` assegna `undefined` al primo elemento (la lunghezza non viene modificata)
- `concat(arr1 [, arr2,...])` concatena gli array

```
var myChildren = myGirls.concat(myBoys);
```

JavaScript Array

Metodi

- ▶ `splice(pos, n[, e1,...,en])` a partire dalla posizione pos, elimina n elementi e inserisce e1,...

```
var fruits = ["Banana", "Orange", "Apple"];
fruits.splice(1, 0, "Lemon", "Kiwi");
```

Senza gli ultimi parametri elimina solo gli elementi

```
fruits.splice(0, 1); //elimina il primo elemento
```

- ▶ `slice(da, a)` crea un nuovo array prelevando gli elementi dalla posizione da fino ad a esclusa (se non c'è a fino all'ultimo elemento)

```
x=fruits.slice(1, 3);
y=fruits.slice(2);
```

JavaScript

Array ES6: destructuring e spread operator

- ▶ Il **destructuring** è un'espressione JavaScript che consente di separare i valori di un array (o le proprietà di un oggetto) in variabili distinte.

```
const foo = ['one', 'two', 'three'];
const [red, yellow, green] = foo;
console.log(red); // "one"
console.log(yellow); // "two"
console.log(green); // "three"
```

- ▶ La **spread syntax** consente di "espandere" gli elementi di un array. Nella pratica è utilizzata per aggiungere elementi ad un array o oggetto, combinare array o oggetti e copiare array o oggetti(shallow copy).

...array

JavaScript

Array: Array.prototype.forEach()

- ▶ Il metodo **forEach** esegue la funzione passata come parametro per ciascun elemento dell'array.

```
const array1 = ['a', 'b', 'c'];
```

```
for (i = 0; i < array1.length; i++) {  
    console.log(array1[i]);  
}
```

Equivale a

```
array1.forEach((element) => console.log(element));
```

JavaScript

Array: Array.prototype.map()

- ▶ Il metodo **map** crea un nuovo array popolato con i valori ritornati dalla funzione passata come parametro. Questa funzione viene chiamata per ogni elemento dell'array originale.

```
const esMap = [1, 2, 3, 4, 5].map((number) =>  
    number * 2);  
  
console.log(esMap);  
// [2, 4, 6, 8, 10]
```

JavaScript

Array: Array.prototype.map()

```
arr.map(function(element, index, array){ }, this);
```

- ▶ La funzione callback function() è invocata su ogni elemento dell'array, e il metodo map() passa sempre alla funzione l'elemento corrente (element), l'indice (index) dell'elemento corrente, e l'intero oggetto array.
- ▶ L'argomento this è usato dentro la funzione callback. Come default il suo valore è undefined

JavaScript

Array: Array.prototype.map()

```
let ar = [2, 3, 5, 7];
debugger;
ar = ar.map(function (element, index, array) {
    console.log(element); //2 3 ...
    console.log(index);   //0 1 ...
    console.log(array);   // [2,3,...]
    console.log(this);    // 80
        return element + 1; //equivale a foreach element =
element +1
}, 80);
console.log(ar);
```

JavaScript

Array: Array.prototype.filter()

- ▶ Il metodo **filter** crea un nuovo array contenente tutti gli elementi che passano il test implementato nella funzione passata come parametro.

```
const words = ['spray', 'limit', 'elite', 'exuberant',
'destruction', 'present'];
```

```
const result = words.filter(word => word.length > 6);
```

```
console.log(result);
```

```
// ["exuberant", "destruction", "present"]
```

JavaScript

Array: Array.prototype.reduce()

- Il metodo **reduce** esegue la funzione passata come argomento per ciascun elemento dell'array. Questa funzione, ad ogni iterazione avrà due parametri, il primo sarà il valore ritornato dal calcolo precedente mentre il secondo l'attuale elemento dell'array. Il risultato finale è un singolo valore.

```
const array1 = [1, 2, 3, 4]; // 0 + 1 + 2 + 3 + 4  
const initialValue = 0;  
const sumWithInitial = array1.reduce(  
  (previousValue, currentValue) => previousValue + currentValue,  
  initialValue  
);  
console.log(sumWithInitial); // 10
```

JavaScript

Array multidimensionali

- ▶ Non si possono definire array bidimensionali, ma un array può essere un elemento di un array. Questo consente di definire array multidimensionali o matrici.

```
const mat = new Array(N);  
for (k=0; k<N; k++) mat[k] = new Array(M);
```

Oppure

```
const mat = [[4,3,1],[8,2,7],[8,6, 1]];
```

- ▶ Per accedere al numero 8 indicheremo le sue coordinate nel seguente modo :

```
var otto = mat[1][0]; // [riga][colonna]
```

JavaScript

Oggetti: ToString

Tutti gli oggetti JavaScript sono basati su Object e condividono alcuni metodi: `toString()` e `valueOf()`.

- ▶ `toString()` restituisce una versione in stringa dell'oggetto:

```
var x = new Object(32);  
x.toString(); //restituisce "32"
```

- ▶ Nel caso di oggetto non riconducibile ad un tipo di dato primitivo sarà restituita la stringa [object Object]:

```
var persona = new Object({ nome: "Mario",  
cognome: "Rossi" } );  
persona.toString(); //restituisce "[object Object]"
```

JavaScript

Oggetti: ToString

123456..toString(36)

- ▶ I ".." non sono un errore. Se vogliamo chiamare un metodo direttamente da un numero, come `toString` nell'esempio sopra, abbiamo bisogno di inserire due punti ...
- ▶ Se inseriamo un solo punto: `123456.toString(36)`, otterremo un errore, perché la sintassi JavaScript implica una parte decimale a seguire del primo punto. Se invece inseriamo un ulteriore punto, allora JavaScript capirà che la parte decimale è vuota e procederà nel chiamare il metodo.
- ▶ Potremmo anche scrivere `(123456).toString(36)`.

JavaScript

Oggetti: ValueOf

- ▶ Il metodo `valueOf()` restituisce il corrispondente valore del tipo di dato primitivo associato all'oggetto:

```
var x = new Object(32);
```

`x.valueOf()`

// restituisce 32

Oggetto	Valore restituito
Array	Restituisce l'istanza della matrice.
Boolean	Valore Boolean.
Date	Il valore memorizzato dell'ora espresso in millisecondi a partire dalla mezzanotte dell'1 gennaio 1970 in formato UTC.
Funzione	Funzione stessa.
Number	Valore numerico.
Oggetto non associato a un tipo primitivo	Oggetto stesso. È l'impostazione predefinita.

JavaScript

Oggetti: `toString` e `valueOf`

- ▶ È da sottolineare come JavaScript chiama implicitamente questi metodi quando è necessario effettuare delle conversioni o quando in un'espressione è richiesto il valore primitivo dell'oggetto.



Oggetti riflessi BOM e DOM

di Roberta Molinari

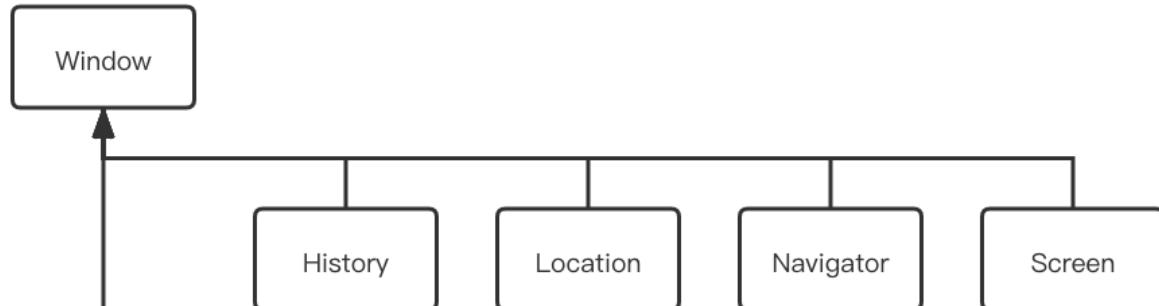
Javascript

Oggetti riflessi

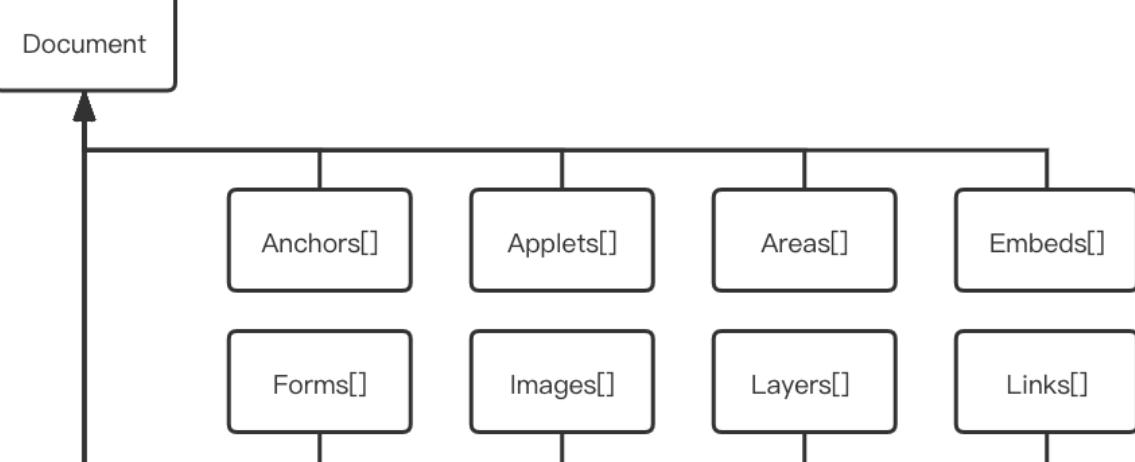
- ▶ Javascript gestisce gli oggetti riflessi dal browser secondo 2 modelli a oggetti
 - **BOM**: il Browser Object Model fornisce l'accesso alle varie caratteristiche e all'ambiente del browser: finestra, schermo, cronologia,...
 - **DOM**: il Document Object Model fornisce l'accesso agli elementi che compongono il contenuto della finestra, ovvero il documento con le varie componenti HTML
- ▶ Javascript scomponete una pagina web in una serie di oggetti in relazione gerarchica tra loro, ciascuno dotato di proprietà e metodi.

BOM e DOM

BOM(Browser Object Model)



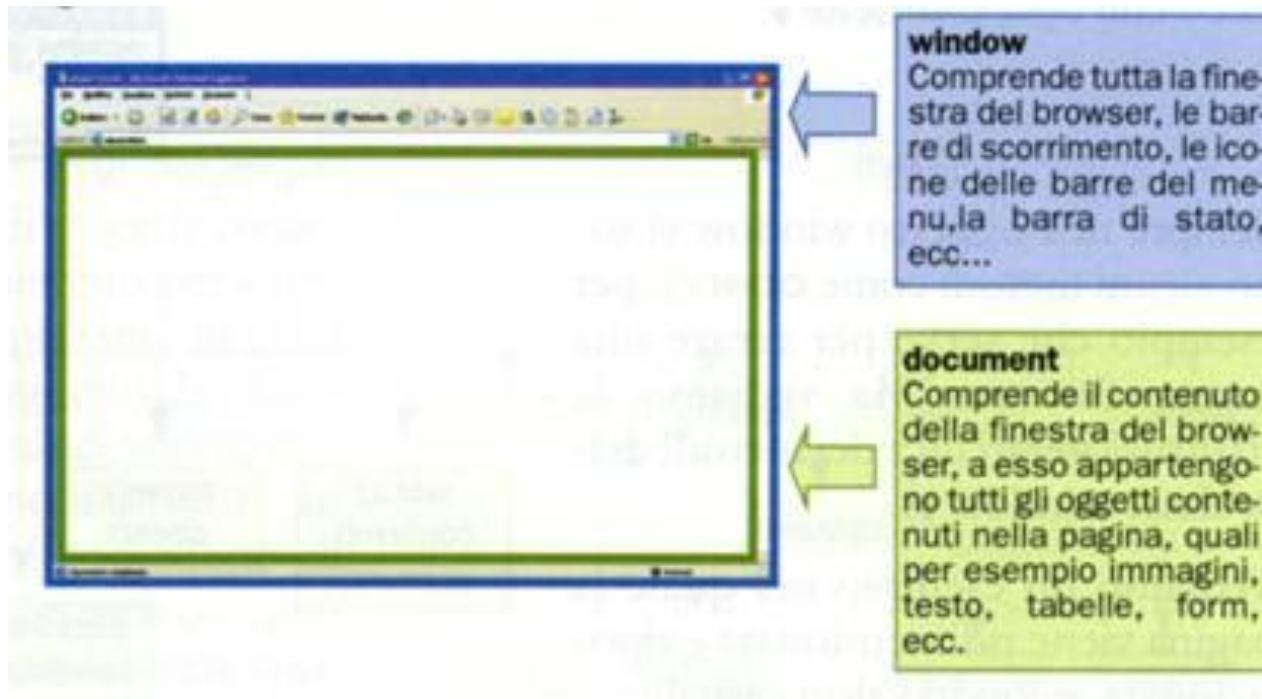
DOM(Document Object Model)



- ▶ forms, images,... sono vettori il cui primo elemento ha indice 0

Javascript BOM

- ▶ **Window:** la finestra principale del browser, l'oggetto di livello massimo (sinonimo: self). Possono esserci più oggetti window contemporaneamente attivi



Javascript

BOM: window

- ▶ Window: la finestra principale del browser
 - ▶ Proprietà
 - *status* testo nella barra di stato (IE)
 - ▶ Metodi
 - *open()* *close()*
 - *alert()* *prompt()* *confirm()*
 - *setTimeout()* *clearTimeout()*
 - *setInterval()* *clearInterval()*
 - ▶ Eventi
 - *onload*
 - *onresize*
 - *onunload*

Javascript

window.open()

```
w1 = open ("url", "nome" [, "impostazioni"])
```

apre una nuova finestra e `w1` è il suo riferimento. Se mancano i primi parametri inserire ("","", "Nuova Finestra",...)

- ▶ *nome* può essere:
 - il nome della finestra (Nota: il nome non specifica il titolo della nuova finestra), può essere usato in un target di una form
 - _blank (valore predefinito),
 - _parent e _top (per i frame)
 - _self per cui si va al nuovo URL che sostituisce la pagina corrente
- ▶ Le specifiche della finestra vanno separate da una virgola e sono illustrate nella tabella che segue. Quelle mancanti sono considerate false

*E.s. "toolbar=no, width=40, height=40,
scrollbars=0, menubar=no"*

Javascript window.open() specifiche

channelmode=yes no 1 0	Whether or not to display the window in theater mode. Default is no. IE only
directories=yes no 1 0	Obsolete. Whether or not to add directory buttons. Default is yes. IE only
fullscreen=yes no 1 0	Whether or not to display the browser in full-screen mode. Default is no. A window in full-screen mode must also be in theater mode. IE only
height=pixels	The height of the window. Min. value is 100
left=pixels	The left position of the window. Negative values not allowed
location=yes no 1 0	Whether or not to display the address field. Opera only
menubar=yes no 1 0	Whether or not to display the menu bar
resizable=yes no 1 0	Whether or not the window is resizable. IE only
scrollbars=yes no 1 0	Whether or not to display scroll bars. IE, Firefox & Opera only
status=yes no 1 0	Whether or not to add a status bar. IE
titlebar=yes no 1 0	Whether or not to display the title bar. Ignored unless the calling application is an HTML Application or a trusted dialog box
toolbar=yes no 1 0	Whether or not to display the browser toolbar. IE and Firefox only
top=pixels	The top position of the window. Negative values not allowed
width=pixels	The width of the window. Min. value is 100

Javascript

window.opener

- ▶ La proprietà `opener` restituisce un riferimento alla finestra che ha creato la finestra corrente con il metodo `window.open()`. In questo modo è possibile restituire valori alla finestra di origine (madre).

window.opener.close() //chiude la madre

```
opener.document.getElementById("txtMadre").  
innerHTML="testo dalla figlia";  
//txtMadre deve essere dichiarato prima
```

JavaScript

setTimeout() clearTimeout()

window.setTimeout (funz, millisec)

- ▶ Passati millisec millisecondi, si esegue la funzione funz 1 sola volta. Il primo parametro può essere:

- il nome di una funzione di callback `setTimeout(funz, 30)`
- la stringa contenente le istruzioni da eseguire (chiamata alla funzione o istruzioni estese) `setTimeout("alert(' ! ')", 30)`
- dichiarazione della funzione

`setTimeout(function () {alert('ciao')}, 3000)`

o con le funzioni freccia

`setTimeout(()=>{alert('ciao')}, 3000)`

- ▶ Se si vuole evitare l'esecuzione si deve assegnare ad una variabile la funzione e poi eseguire una **.clearTimeout** prima che passi il tempo

```
t=window.setTimeout("funz()", millisec);
```

...

```
window.clearTimeout(t);
```

JavaScript

setInterval() clearInterval()

window.setInterval(funz, millisec)

- ▶ Come setTimeout, ma l'esecuzione viene eseguita ogni millisec millisecondi. Per il primo parametro valgono le stesse regole di setTimeout
- ▶ Se si vuole interromperne l'esecuzione si deve assegnare ad una variabile la funzione e poi eseguire una .clearInterval
 - t=window.setInterval("funz()", millisec);
 - ...
 - window.clearInterval(t);

Javascript BOM

- ▶ **Location:** contiene proprietà relative alla posizione del documento corrente come il suo URL
- ▶ **History:** la lista degli URL visitati nella sessione attuale
- ▶ **Navigator:** contiene alcuni dettagli sul browser in uso (es. nome e versione).
- ▶ **Screen:** contiene informazioni relative allo schermo del device utilizzato dall'utente.

- ▶ Il **DOM Document Object Model** è una standard, indipendente dal linguaggio e dalla piattaforma, di rappresentazione e di interazione con documenti strutturati (HTML o XML) visti come un insieme di oggetti. È un'interfaccia, (API), ideata dal consorzio W3C, che permette di accedere agli elementi di una pagina. Il DOM consente a programmi e script di accedere e aggiornare dinamicamente il contenuto, la struttura e lo stile di un documento.
- ▶ Lo standard W3C DOM si suddivide in:
 - Core DOM: modello standard per tutti i tipi di documenti
 - XML DOM: modello standard per i documenti XML
 - HTML DOM: modello standard per i documenti HTML

HTML DOM

- ▶ Il DOM HTML è un **modello a oggetti** di un'**interfaccia di programmazione** per HTML. Definisce:
 - Gli elementi HTML come **oggetti**
 - Le **proprietà** di tutti gli elementi HTML
 - I **metodi** per accedere a tutti gli elementi HTML
 - Gli **eventi** per tutti gli elementi HTML
- ▶ **HTML DOM** è uno **standard che specifica come ottenere, modificare, aggiungere o eliminare elementi HTML.**

<https://software.hixie.ch/utilities/js/live-dom-viewer>

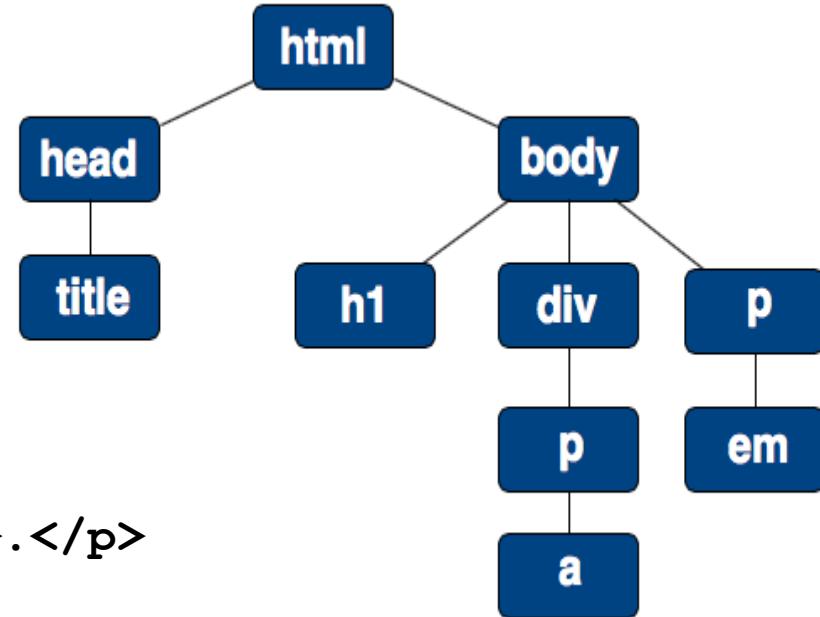
HTML DOM

- ▶ Nel DOM HTML, una pagina HTML è rappresentata come un albero, in cui ogni elemento è genitore e/o figlio di un altro, tranne HTML (che è la root).
- ▶ Gli oggetti nell'albero DOM potranno essere indirizzati e modificati usando metodi sugli oggetti.
- ▶ Non è corretto pensare al DOM come ad una rappresentazione equivalente, o alternativa, del codice HTML infatti:
 - ▶ il DOM riunisce in un'unica struttura tutto il markup contenuto nella pagina, sia esso HTML, XML o altro.
 - ▶ il contenuto del markup è statico mentre il contenuto del DOM è dinamico, in quanto può essere modificato dall'utente o da altri meccanismi (ad esempio con codice JavaScript).

HTML DOM

Struttura ad albero di un documento

```
<html>
  <head>
    <title>Titolo pagina</title>
  </head>
  <body>
    <h1>Testo del titolo</h1>
    <div>
      <p>Testo del paragrafo
        <a href="pagina.htm">
          Testo del link</a>
      </p>
      <p>Secondo <em>paragrafo</em>. </p>
    </body>
</html>
```



HTML DOM

i nodi

Secondo lo standard W3C HTML DOM, tutto in un documento HTML è un **nodo**:

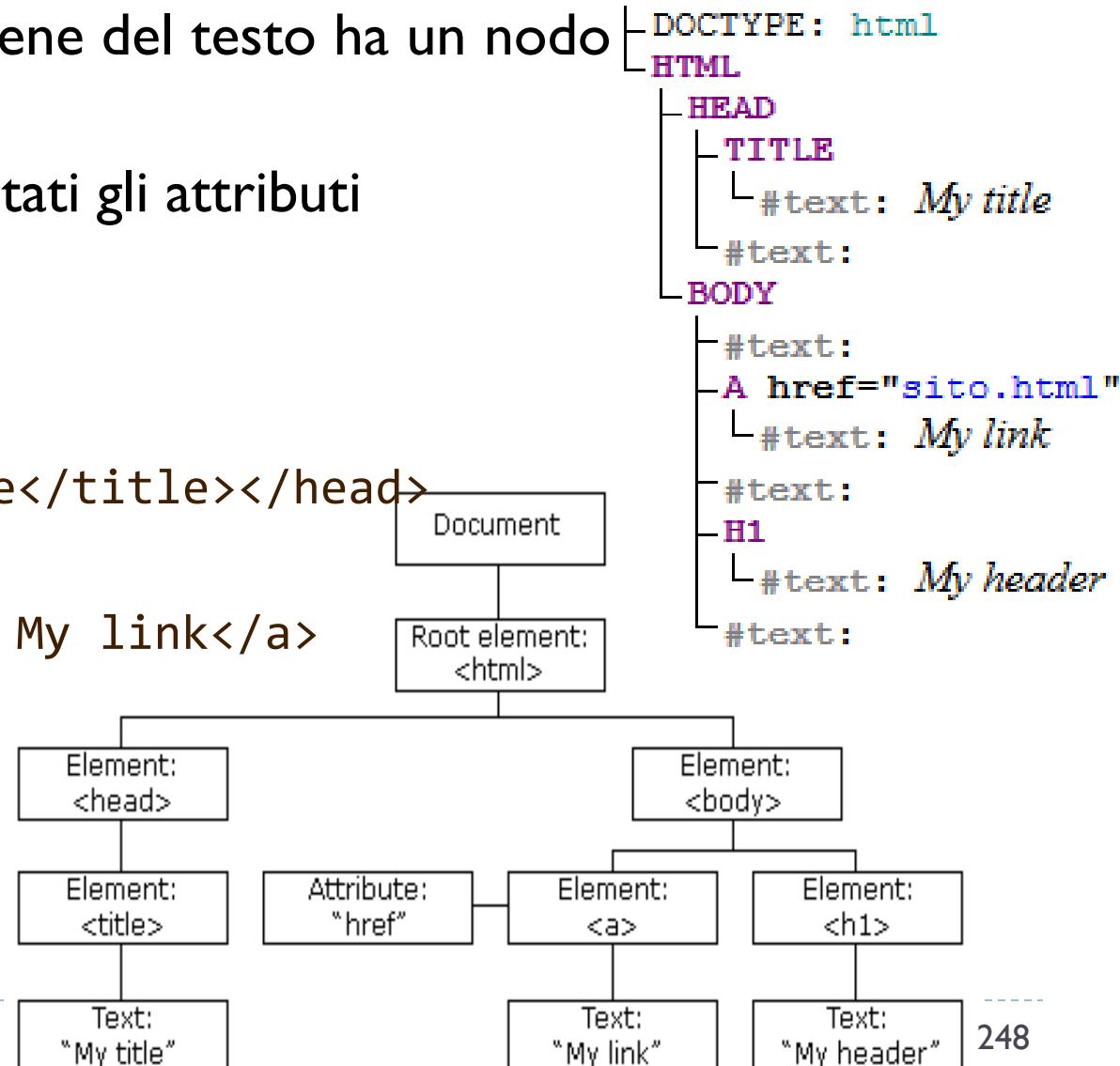
- ▶ L'intero documento è un **nodo documento**
- ▶ Ogni tag HTML è un **nodo elemento** che può contenere altri nodi
- ▶ Il testo all'interno degli elementi HTML sono **nodi testo**
- ▶ Ogni attributo HTML è un **nodo attributo** (obsoleto)
- ▶ Tutti i commenti sono **nodi commento**

HTML DOM

Struttura ad albero di un documento

- ▶ Ogni nodo che contiene del testo ha un nodo di tipo testo (foglia)
- ▶ Vengono anche riportati gli attributi

```
<!DOCTYPE HTML>  
<html>  
<head><title>My title</title></head>  
<body>  
<a href="sito.html"> My link</a>  
<h1>My header</h1>  
</body>  
</html>
```



verifica compatibilità

- ▶ Per verificare che il browser supporti almeno il DOM di livello 1 (1998) definito dal W3C utilizzare la seguente funzione:

```
function domw3c() {  
    if(document.getElementById) {  
        alert(":) DOM Supportato!");  
    } else {  
        alert(":(" DOM NON Supportato!");  
    }  
}
```

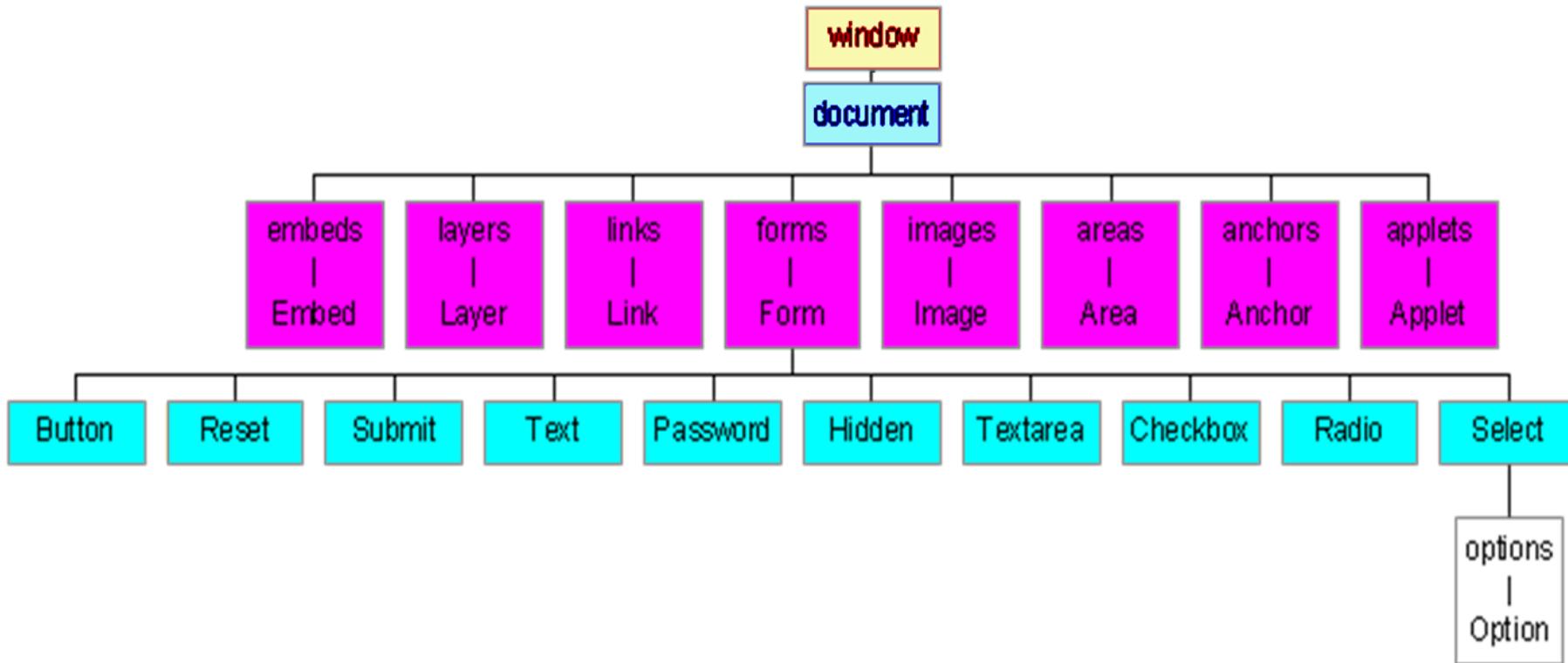
DOM

document

Document: Parte del documento tra <body> e </body>, inserito nella finestra principale del browser. È la finestra attiva.

- ▶ Tutti gli oggetti contenuti nel documento sono visti come sue proprietà, poiché risiedono al suo interno. Questi sono individuabili con vettori con indice che parte da 0 e segue l'ordine con il quale i tag (es. per le immagini) vengono trovati nella pagina in una scansione dall'alto verso il basso (es. ogni elemento del vettore images è un'immagine presente nella pagina).

DOM document



DOM

document: proprietà

- ▶ `.title` il titolo del documento
- ▶ `.URL` URL completo del documento

Le seguenti proprietà possono appartenere anche a tutti i nodi i di tipo elemento

- ▶ `.id` valore dell'id
- ▶ `.firstChild`, `.lastChild` primo/ultimo nodo dell'elemento
- ▶ `.parentNode` nodo padre dell'elemento

DOM

document: proprietà

- ▶ `.alinkColor, .linkColor, .vlinkColor`
colore link
- ▶ `.bgColor, .fgColor` colore sfondo, testo
- ▶ `.domain` dominio del server in cui è caricato il documento
- ▶ `.lastModified` data ultima modifica
- ▶ `.referrer` se il documento corrente è stato raggiunto da un altro ne restituisce l'URL

document: metodi di selezione

- ▶ `getElementById("id")` restituisce il riferimento all'elemento caratterizzato univocamente da quell'ID
- ▶ `getElementsByTagName("tag")` ritorna un array di tutti gli elementi del tag specificato
- ▶ `getElementsByName("name")` ritorna un array di tutti gli elementi con quel Name
- ▶ `querySelectorAll("selCSS")` ritorna la lista di tutti i selettori CSS che si chiamano selCSS (tipo "p b" o "p.class" o "#id")
- ▶ `querySelector("selCSS")` restituisce il primo

DOM

document: modifica degli elementi

- ▶ Si devono assegnare i nuovi valori alle seguenti proprietà degli elementi
 - ▶ `el.innerHTML` – contenuto HTML di un elemento
 - ▶ `el.innerText` - contenuto testuale di un elemento esclusi i tag HTML
 - ▶ `el.nomeAttributo` – imposta un attributo
 - ▶ `el.style.proprietà` – imposta lo style css
- ▶ Oppure si modificano con i seguenti metodi
 - ▶ `elem.insertAdjacentHTML(where, html)`
inserisce un codice HTML in un punto specificato dell'elemento
 - ▶ `el.setAttribute(attributo, valore)`

DOM

document: modifica insertAdjacentHTML

elem.**insertAdjacentHTML**(where, html)

- ▶ Il primo parametro è una parola-codice che specifica dove inserire in relazione a elem. Deve essere una delle seguenti:
 - ▶ "beforebegin" – inserisce html immediatamente prima di elem,
 - ▶ "afterbegin" – inserisce html dentro elem, all'inizio,
 - ▶ "beforeend" – inserisce html dentro elem, alla fine,
 - ▶ "afterend" – inserisce html immediatamente dopo elem.

document: metodi di creazione

Per creare dinamicamente nuovo nodi nel DOM

- ▶ `document.createTextNode(stringa)`
restituisce un text Node con il contenuto = stringa
- ▶ `document.createElement(nomeTag)`
restituisce un elemento con il nome tag specificato (viene
assegnato alla proprietà nodeName dell'elemento)
- ▶ `appendChild(el)` inserisce l'elemento el dopo
l'ultimo figlio del nodo a cui è applicato il metodo
- ▶ `insertBefore(newnode, existingNode)`
inserisce prima dell'existingNode il newNode al nodo
a cui è applicato. Se existingNode=null lo mette in
coda

DOM

document: metodi di modifica confronto

html

elemento.innerHTML = 'stringa_html'

Pro: semplice

Cons:

- perdita riferimenti ai vecchi nodi
- perdita vecchi event handlers
- lentezza
- rischi sicurezza

Elemento.InsertAdjacentHTML(dove, 'stringa_html')

Pro: veloce, semplice

Cons:

- rischi sicurezza
- potenzialmente meno manutenibile

Pausa (k)

Elementi DOM

Crearli prima con:

- document.createElement('tag')
- elemento.cloneNode(deep: true/false)

Configurarli:

- elemento.style = ...
- elemento.classList.Add/Remove()
- elemento.attributo = ...

Inserirli / spostarli / eliminarli

- elemento.InsertAdjacentElement(dove, cosa)
- elemento.append(elemento, elemento, ...)
- elemento.prepend(elemento, elemento, ...)
- elemento.before(elemento, elemento, ...)
- elemento.after(elemento, elemento, ...)
- elemento.replaceWith(elemento, elemento, ...)
- elemento.remove(elemento, elemento, ...)

Pro: sicurezza, manutenibilità

Cons: scomodo con configurazione ricca



DOM

document: metodi di eliminazione

Per eliminare dinamicamente nodi nel DOM

- ▶ `el.removeChild(nodo)` restituisce il nodo eliminato o `NULL` se il nodo non esiste
- ▶ `el.replaceChild(new, old)` sostituisce un nodo con un altro
- ▶ `hasChildNodes()` verifica se un nodo ha dei figli

```
while(el.hasChildNodes()){  
    el.removeChild(el.firstChild)  
}
```

DOM

document: metodi di eliminazione

Esempi per eliminare tutti i "figli" di un elemento

```
// Get the <ul> element with id="myList"  
var list =  
    document.getElementById("myList");
```

```
// As long as <ul> has a child node, remove it  
while (list.childNodes.length) {  
    list.removeChild(list.firstChild);  
}
```

Oppure se non ho il riferimento al padre, "elimino il figlio del proprio padre" finchè ce ne sono

```
let list = document.getElementsByTagName(<li>);  
while(list.length>0) {  
    list[0].parentNode.removeChild(list[0]);  
}
```

DOM

document: metodi - esempio

```
<div id="myDiv"></div>
```

con

```
let newP = document.createElement("p");
newP.setAttribute("class", "my-class");
let t = document.createTextNode("Nuovo");
newP.appendChild(t);
document.getElementById("myDiv").appendChild
(newP);
```

Si ottiene

```
<div id="myDiv">
  <p class="my-class">Nuovo</p>
</div>
```

DOM

document: metodi per le tavole

Per aggiungere più facilmente gli elementi di una tabella si possono usare i seguenti metodi

```
const table = document.createElement("table");
//crea la tabella
const row = table.insertRow();
//aggiunge una riga <tr>
const cellora = row.insertCell();
//aggiunge una cella <td>
```

DOM

document: metodi per elemento radio

ATTENZIONE!

Per i radio, il testo che compare dopo non è nel suo innerHTML.

```
opt = document.createElement("input")
opt.type="radio"; opt.name= "sesso" ; opt.value="M"
document.body.appendChild(opt)
testo = document.createTextNode("Maschio")
document.body.appendChild(testo)
```

```
opt = document.createElement("input")
opt.type="radio"; opt.name= "sesso" ; opt.value="F"
document.body.appendChild(opt)
testo = document.createTextNode("Femmina")
document.body.appendChild(testo)
```

document: metodi di modifica

- ▶ `open()` apre e cancella il contenuto attuale
- ▶ `close()` chiude e rende visibili le modifiche apportate
- ▶ `write(s)` `writeln(s)` scrive la stringa di testo `s` nel documento HTML (`writeln` le termina con '`\n`'). Se non c'è stata una `open` aggiunge in fondo

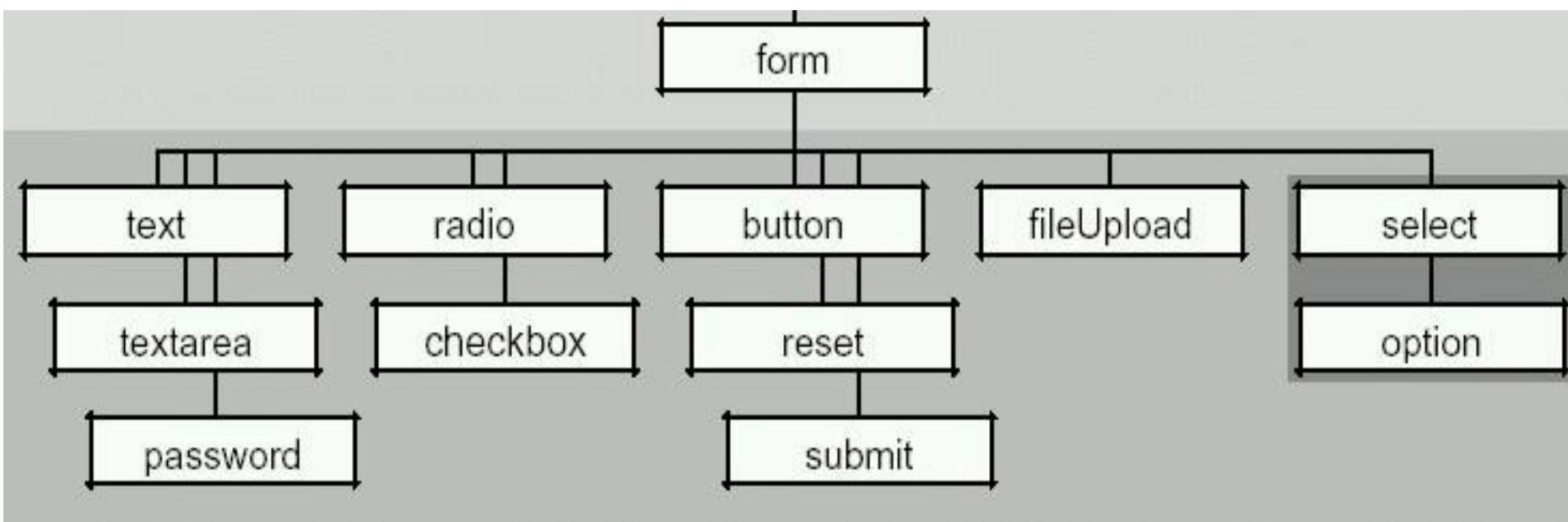
Javascript

Oggetti riflessi HTML

- ▶ document.images
- ▶ document.links
- ▶ document.anchors
- ▶ document.cookies
- ▶ document.layers **i livelli presenti**
- ▶ document.applets
- ▶ document embeds **gli oggetti multimediali**
- ▶ document.forms

Javascript document.forms

- ▶ a ogni form viene associato un vettore di nome elements che contiene tutti gli oggetti contenuti nel form stesso.



Javascript

Riferimento agli oggetti riflessi

► Per fare riferimento alla prima immagine, di nome/id *imgImmagine*, presente nella pagina

-`window.document.images[0]` //rif. Assoluto

-`document.images["imgImmagine"]`

-`document.imgImmagine`

-`imgImmagine` //rif. relativo

► Per fare riferimento al primo elemento del form *frmDati*, che è una casella di testo di nome/id *txtNomeCognome*:

-`window.document.forms["frmDati"].elements[0]`

-`document.frmDati.elements[0]`

-
- ▶ id è valido per qualsiasi elemento HTML, mentre il name si applica solo a pochi elementi: a, input, textarea
 - ▶ Un elemento di input (e textarea) richiede che il name sia impostato se si desidera che sia inviato.

```
<input type="text" id="myText" name="fname"  
value="Mickey">
```

Javascript

Riferimento agli oggetti riflessi

► Si possono usare i metodi getElement per riferirsi agli oggetti del documento: per riferirsi all'immagine di nome ***imgImmagine*** e id univoco ***img1*** si fa

```
document.getElementById("img1")
```

```
document.getElementsByName("imgImmagine")
```

– Per riferirsi a tutte le immagini

```
x=document.getElementsByTagName("img");
```

```
x.item(k) . ...
```

```
x[k] . ...
```

► Nota: il "nomeId" è case sensitive

Javascript

Proprietà .innerHTML

È possibile modificare il contenuto di un nodo (quello che compare tra il tag di apertura e il relativo tag di chiusura) attraverso la modifica della sua proprietà .innerHTML

```
document.getElementById(id).innerHTML  
= codice_HTML;
```

dove codice_HTML è una stringa contenente il codice HTML da inserire all'interno del nodo.

Attenzione a dove si vuole inserire il nuovo codice HTML, in quanto il metodo innerHTML sostituisce tutto il contenuto del nodo al quale viene applicato, con il nuovo codice specificato.

```
<div id="testo"> <i>vecchio testo</i> </div>
```

```
document.getElementById("testo").innerHTML =
```

Javascript

Riferimento agli oggetti riflessi

▶ Per modificare un attributo

```
.setAttribute ("nome", "val")
```

▶ Per recuperare un attributo

```
.getAttribute ("nome") ;
```

▶ Per eliminare un attributo

```
.removeAttribute ("nome") ;
```

▶ Per verificare se possiede quell'attributo

```
.hasAttribute ("nome") ;
```

```
var x = document.getElementById("myAnchor")
if (x.hasAttribute("target")) {
    x.setAttribute("target", "_self");
}
```



Nota: il "nome" NON è case sensitive

274

Javascript

Riferimento agli oggetti riflessi

- ▶ Per modificare un attributo si può fare anche direttamente

.nomeAttributo

```
var x = document.getElementById("myAnchor")
x.target=_self; var y= x.target;
```

Nota: il "nome" dell'attributo è case sensitive

- ▶ Per modificare un attributo dello style

.style.nomeAttributo

```
var x = document.getElementById("myAnchor")
x.target=_self
var y= x.target
```



Javascript Abbreviazioni

- ▶ **Uso di this per riferirsi all'oggetto corrente**

```
<input type ='text' name='txtNome'  
onblur='numero(this.value)'>
```

- ▶ **Uso di with per non ripetere il riferimento completo all'oggetto**

```
with (oggetto) {comandi}
```

- ▶ **Definire una variabile che contenga il riferimento ad un oggetto**

```
var OggImmagine = document.imgImmagine;  
OggImmagine.width += i;
```

Javascript Form

► Proprietà

- name, action, enctype, method, target
- elements **vettore degli elementi del form**
- length **numero di elementi compresi hidden e pulsanti**

► Metodi

- reset(), submit()

► Gestore Eventi

- onClick, onSubmit, onReset

Javascript Button

**Include anche i pulsanti speciali *submit*,
*reset***

► Proprietà

- name, type (**button**, **reset** o **submit**), value

► Metodi

- click()

► Gestore Eventi

- onClick, ondblclick



Javascript

Text

► Proprietà

- name, type (**password**), value, length, defaultValue

► Metodi

- focus(), blur(), select()

► Gestore Eventi

- onFocus **imposta lo stato attivo**
- onBlur **elimina lo stato attivo**
- onSelect **seleziona il testo nella text**

Javascript Textarea

► Proprietà

- name, type, value, length, cols, rows, defaultValue

► Metodi

- focus(), blur(), select()

► Gestore Eventi

- onFocus, onBlur, onSelect, onChange

Javascript Checkbox

► Proprietà

- name, type, value, checked

► Metodi

- click()

► Gestore Eventi

- onClick

Se hanno un solo nome si genera un array



Javascript Radio

► Proprietà

- name, type, value, checked, defaultchecked
- Length (**numero di pulsanti presenti**)

► Metodi

- click()

► Gestore Eventi

- onClick

Ha un solo nome si genera un array



Javascript Select

► Proprietà

- name, value, checked, text (**tra option**)
- type (select-one, select-multiple),
- length (**numero di option**)
- options (**oggetti option contenute**)
- selectedIndex (**indice opzione selezionata, il primo se multiple**)

► Metodi

- blur(), focus()

► Gestore Eventi

- onBlur, onChange, onFocus



Javascript

Select.options

► Proprietà

- lenght **(numero di option)**
- selectedIndex **(imposta o restituisce l'indice dell'opzione selezionata (il primo=0), il primo se multiple)**

► Metodi

- add(option[, index]) **Aggiunge la option in fondo o nella posizione eventualmente indicata**
- remove(index) **elimina la option con indice index**

```
var c = document.createElement("option");
c.text = "Prova";
mySelect.options.add(c, 1);
```

JavaScript

Schema oggetti riflessi HTML

Oggetto	Proprietà	Metodi	Eventi
button	name – type – value - disabled	click - dblclick	onClick - ondblclick
text	name – type – value – size – defaultValue – value.length - disabled	focus – blur - select	onFocus – onBlur – onSelect – onChange
textarea	name – value – size – cols – rows - disabled	focus – blur - select	onFocus – onBlur – onSelect – onChange
checkbox	name – type - length value – checked (T F) - defaultChecked – disabled (di un [k])	click	onClick
radio	name – type – length value - checked (T F) - defaultChecked – disabled (di un [k])	click	onClick
select	name – value – type(select-one; select-multiple) - length – selectedIndex – options - disabled	focus – blur	onFocus – onBlur – onChange
			285



JavaScript

Schema oggetti riflessi HTML

- ▶ In realtà i metodi e gli eventi sono utilizzabili da tutti gli oggetti, ma quelli in elenco sono i più sensati
- ▶ `type` è una proprietà di sola lettura
- ▶ `blur()` fa perdere il focus e non lo assegna a nessuno
- ▶ `disabled` rende non "toccabile" dall'utente
- ▶ `default` restituisce il valore o le impostazioni di `checked` iniziali
- ▶ Check | Radio `[k]` .`value` è quello impostato con `value`
- ▶ Check | Radio .`length`= quanti oggetti check o radio hanno lo stesso nome. Per riferirsi ai vari radio si usa `form`.
`Check | Radio [k]`
- ▶ Nella select comunque qualcosa è selezionato e il suo indice è reperibile con `select.selectedIndex` o `select.options.selectedIndex`, mentre il suo `value` (che è sempre di tipo string) è reperibile con `select.value` o con `select.options[select.selectedIndex].value`
- ▶ `Select.options[k].text`= testo che segue il tag option

Javascript

Esempio controllo input

```
function checkData() {  
    var err = ""; name=document.getElementById("username");  
    if (name.value=='' || name.value.length<2)  
        err += " Manca nome\n";  
    news=document.getElementsByName ("news")  
    if (!news[0].checked && !news[1].checked)  
        err += "Seleziona...\n";  
    if (document.getElementById("combo").selectedIndex==0)  
        err+="seleziona..."  
    if (tel.value=="" || isNaN(tel.value))  
        //devono essere in quest'ordine perché isNaN("")=false  
        err+="Formato numero di telefono sbagliato"  
  
    if (err=="") document.forms ["myForm"] .submit();  
    else alert (err);  
}
```



Javascript

Esempio controllo input regex

Nel caso di caselle di testo, si possono usare le regex per verificare la correttezza del formato del testo.

Si dichiara una costante REGEX con la stringa di confronto e poi si usa il metodo REGEX.test(testoDaVerificare)

```
const REGEX = /^[a-zA-Z ]+$/  
if( txtNome.value === "" || !REGEX.test(txtNome.value) ) {  
    err += "Formato nome scorretto\n";  
}
```

allows YYYY/M/D and periods instead of slashes	/^\d{4}[\./]\d{1,2}[\./]\d{1,2}\$/
YYYY-MM-DD and YYYY-M-D	/^\d{4}\-\d{1,2}\-\d{1,2}\$/
YYYY-MM-DD	/(\d{4})-(\d{2})-(\d{2})/
nomi e cognomi	/^[\a-zA-Z]+\$/

Javascript

Esempio mostra nascondi

```
function mostra() {  
    document.getElementById("content").style.display="block";  
}  
  
function nascondi() {  
    document.getElementById("content").style.display="none";  
}  
  
<div id="content">  
    Questo è il div da mostrare o nascondere  
</div>  
<input type="button" value="Mostra" onclick="mostra()" />  
<input type="button" value="Nascondi"  
onclick="nascondi()" />
```

Javascript modificare stili CSS

Una delle caratteristiche più utili di Javascript, e del DOM in particolare, la possibilità di aggiungere e modificare stili CSS degli elementi di pagina

```
var el=document.getElementById("bigtext") ;  
el.style.fontSize="50px" ;
```

► **essendo float una parola riservata in Js: gli equivalenti saranno quindi cssFloat per browser quali Opera, Mozilla e Safari, mentre styleFloat per Internet Explorer. Basterà impostarle entrambe per ottenere il risultato voluto in maniera cross-browser:**

```
var el=document.getElementById("box") ;  
el.style.styleFloat="left" ;  
el.style.cssFloat="left" ;
```

Javascript modificare stili CSS

Se le dichiarazioni CSS da aggiungere mediante Javascript dovessero essere diverse c'è un'alternativa che si rivela più leggera in termini di peso: intervenire sull'attributo HTML style degli elementi di pagina, proprio come se si impostasse uno stile in linea, attraverso il metodo setAttribute.

```
function so_applyStyleString(obj,str){  
if(document.all && !window.opera)  
    obj.style.setAttribute("cssText",str);  
else  
    obj.setAttribute("style",str);  
}
```

```
var s="float:right;width:10em;border:1px dotted  
#CCC;padding:5px"  
var divs=document.getElementsByTagName("div");  
for(i=0;i<d.length;i++){  
    if(divs[i].className=="pullquote")  
        so_applyStyleString(divs[i],s);
```



Javascript modificare stili CSS

La proprietà className consente di accedere sia in lettura che in scrittura alle classi CSS attribuite sia nel markup che da Javascript stesso.

```
var el=document.getElementById("menu") ;  
el.className="open" ;
```

Per non perdere le classi precedenti

```
el.className + = " open" ;
```

Javascript modificare stili CSS

Attraverso la proprietà style degli elementi DOM è possibile accedere a proprietà CSS in Javascript. La variabile contentColor conterrà effettivamente un valore non nullo *solo se la proprietà CSS è stata in precedenza impostata mediante Javascript.*

```
var el=document.getElementById("content");
var contentColor=el.style.backgroundColor;
```

document.defaultView.getComputedStyle del DOM consente di accedere a valori CSS non precedentemente impostati mediante Javascript.

```
function getStyleProp(x,prop) {
    if(x.currentStyle) //ci sono differenze con IE
        return (x.currentStyle[prop]);
    if(document.defaultView.getComputedStyle)
        return (document.defaultView.getComputedStyle(x,"") [pro
p]);
    return (null);
}
```



Javascript modificare stili CSS

- ▶ Una buona pratica in soluzioni basate su CSS e Javascript è tenere il CSS funzionale e/o presentazionale della soluzione arricchita in un *CSS esterno e separato* rispetto a quello che riguarda layout e presentazione della pagina senza Javascript. Detto ciò, i modi di procedere sono sostanzialmente tre.
- ▶ Il primo e più semplice è linkare nella sezione head sia il javascript sia il CSS:

```
<script type="text/javascript" src="nifty.js"></script>
<link rel="stylesheet" type="text/css" href="nifty.css">
```
- ▶ Questa soluzione ha un piccolo svantaggio, ovvero far scaricare all'utente il CSS necessario anche se Javascript non può girare. Gli utenti che navigano con Javascript disabilitato o con browser non DOM-compatibili saranno una percentuale che oscilla tra il 5 e il 10%, ma servire un CSS di cui non potranno beneficiare uno spreco di byte sia per l'utente che per il server.
- ▶ Il secondo e il terzo approccio linkano il CSS solo nel caso in cui Javascript sia abilitato. Ma come fare? Semplice¹⁹⁴, basta includere il CSS tramite il javascript stesso: ecco così la seconda soluzione.



Gli eventi in Javascript

di Roberta Molinari

JavaScript

programmazione sincrona e asincrona

- ▶ La **programmazione sincrona** esegue un task alla volta e solamente quando il task precedente è completato viene eseguito il task successivo.
- ▶ La **programmazione asincrona** è una tecnica che consente al programma di iniziare un potenziale task di lunga durata e, invece che attendere l'esecuzione del task, continuare ad eseguire altri task mentre il precedente è in esecuzione.

JavaScript

contesto di esecuzione

- ▶ Il codice JavaScript viene eseguito all'interno del proprio **execution context**.
- ▶ Il codice di una funzione viene eseguito all'interno del **function execution context**, mentre il codice globale dentro il **global execution context**. Ciascuna funzione ha il suo execution context.
- ▶ Il **call stack** è uno stack con una struttura LIFO, utilizzato per memorizzare tutti gli **execution context** creati durante l'esecuzione del codice.

sincrono e asincrono

- ▶ JavaScript è un linguaggio **sincrono (mono-thread)** di default: le istruzioni sono svolte una dopo l'altra (**codice bloccante**)
- ▶ Un codice svolto in modalità **asincrona** se l'esecuzione di una istruzione non interrompe l'esecuzione di altre istruzioni (**codice non bloccante**). Questo è anche possibile in Javascript.
- ▶ **Asincrono:** evento che non si verifica contemporaneamente

▶ Esempio codice **sincrono**

```
console.log(" Io ");
console.log(" mangio ");
console.log(" il gelato ");
```

▶ Esempio di codice **asincrono**

```
console.log(" Io ");
// Il blocco qui sotto verrà mostrato dopo due secondi
setTimeout(()=>{
    console.log(" mangio ");
}, 2000);
// prima si mostra questo
console.log(" il gelato ")
```

JavaScript

Funzioni e Call stack

- ▶ In JavaScript, puoi creare e modificare una funzione, usarla come argomento, restituirla da un'altra funzione e assegnarla a una variabile.
- ▶ Il motore JavaScript utilizza una struttura dati chiamata **function execution stack** o **Call Stack**. Lo scopo dello stack è tenere traccia della funzione corrente in esecuzione in questo modo:
 - ▶ Quando il motore JavaScript richiama una funzione, la aggiunge allo stack e l'esecuzione ha inizio.
 - ▶ Se la funzione attualmente eseguita chiama un'altra funzione, il motore aggiunge la seconda funzione allo stack e avvia l'esecuzione.
 - ▶ Una volta terminata l'esecuzione della seconda funzione, il motore la estrae dallo stack.
 - ▶ Il controllo torna indietro per riprendere l'esecuzione della prima funzione dal punto in cui l'aveva lasciata l'ultima volta.
 - ▶ Una volta terminata l'esecuzione della prima funzione, il motore la estrae dallo stack.
 - ▶ Continua in questo modo fino a quando non c'è niente da mettere nella pila.

callback

- ▶ Una funzione di **callback** viene eseguita al completamento di un'operazione asincrona.

```
function callback(){  
    // fa qualcosa  
}  
  
function fun(param){  
    // fa qualcos'altro  
    param();  
}  
  
fun(callback);
```

Esempio tipico è la funzione

```
setTimeout(()=>{alert('ciao')}, 3000)
```

callback: esecuzione

- ▶ JavaScript mantiene una coda di funzioni di callback. Si chiama **callback queue** o **task queue**, è una struttura dati di tipo FIFO, quindi la funzione di callback che per prima entra in coda ha l'opportunità di uscire per prima.
- ▶ Il motore JavaScript continua a eseguire le funzioni presenti nella call stack. Ma le funzione di callback non sono inserite direttamente nello stack, quindi non si tratta di codice bloccante.
- ▶ Il motore esegue un **event loop** o **loop degli eventi**, ovvero esamina periodicamente la coda e estrae una funzione di callback dalla coda solo quando lo stack è vuoto.
- ▶ Quindi la funzione di callback viene generalmente eseguita come qualsiasi altra funzione nello stack.

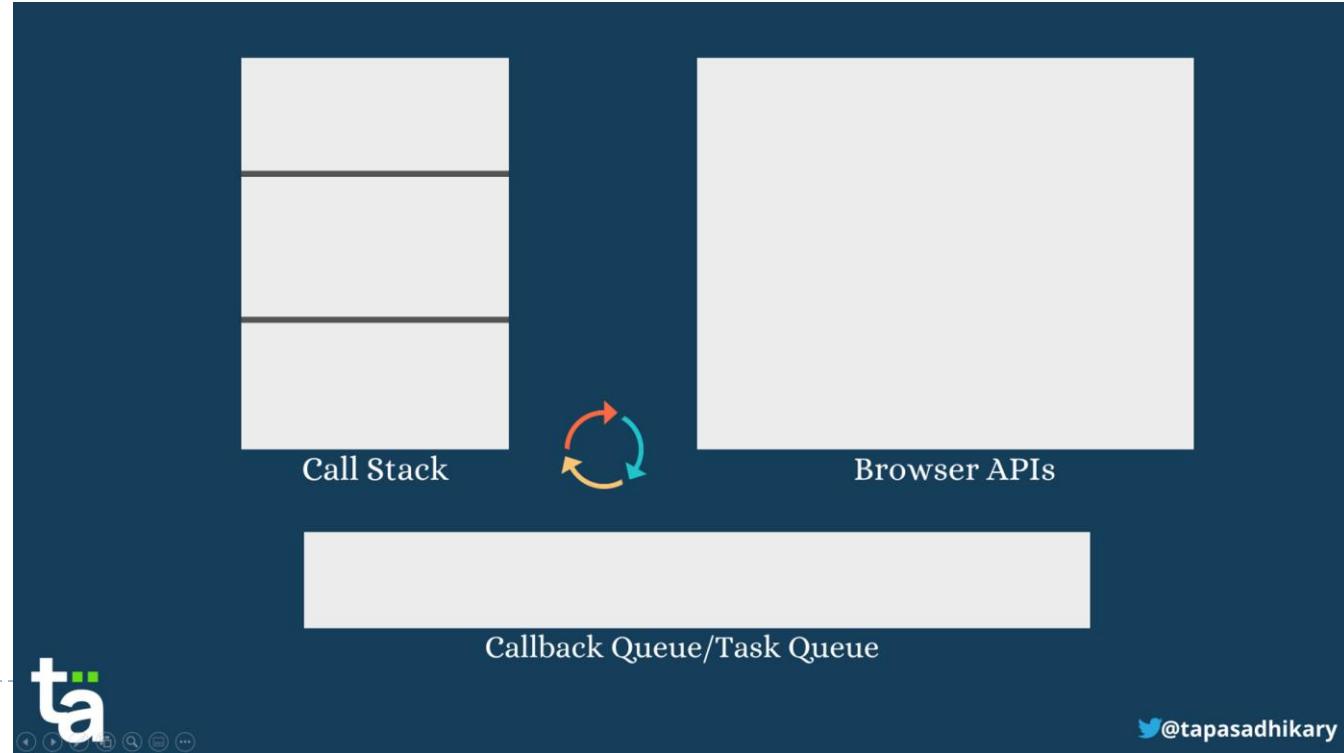
JS

callback: esecuzione

```
function f1() {console.log('f1');}  
function f2() {console.log('f2');}
```

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

Si avrà a console
main
f2
f1



callback: esecuzione

Poiché abbiamo specificato zero come intervallo di tempo per `setTimeout()`, ci aspetteremmo che l'esecuzione della funzione `f1()` sia immediata.

Invece la funzione `f1()` verrà eseguita al termine dell'intera esecuzione della funzione `f2()` e del `main`.

Ciò deriva dal fatto che il verificarsi dell'evento di scadenza del timer non fa altro che inserire un messaggio nella coda degli eventi, ma il runtime JavaScript è già occupato ad eseguire la funzione `f2()` del `main`, per cui eseguirà la funzione `f1()` soltanto al termine delle sue attività. Ecco il perché dell'output

`main`

`f2`

`f1`

JavaScript

Programmazione asincrona

- ▶ **setTimeout()** è una funzione asincrona che esegue un blocco di codice o una callback con un ritardo impostato in millisecondi.
- ▶ Una **callback** non è altro che una funzione che viene passata ad un'altra funzione (higher order function), che la eseguirà al momento opportuno. Le callbacks sono state il principale modo in cui la programmazione asincrona è stata implementata in JavaScript.
- ▶ Gli **Event handlers** sono a tutti gli effetti una forma di programmazione asincrona. Non è possibile sapere in anticipo quando l'utente cliccherà un certo elemento, quindi definiamo un event handler per l'evento click. Questo event handler accetta una funzione di callback che verrà chiamata al manifestarsi dell'evento.

JavaScript moderno

Programmazione asincrona moderna

- ▶ Con ES6 è possibile gestire la programmazione asincrona con le **Promise**
- ▶ Da ES2017 si può gestire con le **async/await**
- ▶ Ultimamente sono state introdotte delle tecnologie con cui è possibile creare dei cosiddetti worker per eseguire JS multithread (normalmente è single thread)

JavaScript

Gli eventi

Per gestire un evento si deve invocare il suo **gestore (handler)**:

1. Assegnando l'handler (possono anche essere istruzioni) direttamente all'attributo associato all'evento nel tag HTML:

```
<IMG onClick = "nomeFunz()">
```

2. Associando la **funzione di callback** alle relative proprietà dell'oggetto:

```
window.onResize = nomeFunz; //senza parentesi
```

3. Tramite addEventListener sull'oggetto DOM (così si ha maggiore flessibilità)

```
body.addEventListener("load", nomeFunz)
```

JavaScript

Gli eventi

Se si utilizza il primo metodo e la funzione restituisce true, viene anche eseguita l'operazione associata di default all'evento, se restituisce false non viene eseguita:

```
<a href="#" onClick="nomeFunz ()">  
"# indica di posizionarsi ad inizio pagina, se  
nomeFunz () restituisce false, non verrà eseguita
```

```
<input type="submit"  
onClick="return confirm('Confermi?')">
```

JavaScript

Gli eventi

Se si utilizza addEventListener l'associazione con la funzione può essere fatta in questo modo

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

```
function myFunction() {  
    document.getElementById("demo").innerText = "Hey"; }
```

o con una funzione anonima (si deve fare così se ci sono dei parametri)

```
document.addEventListener("click", function() {  
    document.getElementById("demo").innerText = "Hey"; } );
```

```
document.onclick = () => innerText = "Hey";
```

```
document.onclick = ()=> alert("Hey");
```

JavaScript

Gli eventi

- ▶ Se l'oggetto ha già un gestore per quell'evento, non si sovrascrive, ma si aggiunge
- ▶ Per eliminare un handler

```
.removeEventListener("click", myFunction)
```

- ▶ Attenzione!

Nel tag HTML si usa l'attributo "**onclick**" nel codice
Javascript si usa l'evento "**click**"

JavaScript

Gli eventi

Evento	Tag	Descrizione
abort		L'utente fallisce il caricamento di un'immagine.
blur	<body> <frameset> <frame> <input type = "text"> <textarea> <select>	Un documento perde il focus dell'input. Un frame perde il focus dell'input. Un frame perde il focus dell'input. Un campo di testo perde il focus dell'input. Un'area di testo perde il focus dell'input. Un elemento di selezione perde il focus dell'input.
change	<input type = "text"> <textarea> <select>	Un campo di testo viene modificato e perde il focus dell'input. Un'area di testo viene modificata e perde il focus dell'input. Un elemento di selezione viene modificato e perde il focus dell'input.
click	<a> <input type = "button"> <input type = "submit"> <input type = "reset"> <input type = "radio"> <input type = "checkbox">	L'utente fa clic su un collegamento. Viene selezionato un pulsante. Viene selezionato il pulsante Submit. Viene selezionato il pulsante Reset. Viene selezionato un pulsante di opzione. Viene selezionata una casella di controllo.

JavaScript

Gli eventi

Evento	Tag	Descrizione
error	 < body> < frameset>	Si è verificato un errore nel caricamento di un'immagine. Si è verificato un errore nel caricamento di un documento. Si è verificato un errore nel caricamento di un set di frame.
focus	<body> <frameset> <frame> <input type= "text"> <textarea> <select>	Un documento diventa attivo con l'input. Un set di frame diventa attivo con l'input. Un frame viene diventa attivo con l'input. Un campo di testo diventa attivo con l'input. Un'area di testo diventa attiva con l'input. Un elemento di selezione diventa attivo con l'input.
load	 <body> <frameset>	L'utente ha caricato e visualizzato un'immagine. Il caricamento del documento è completato. Il caricamento del documento è completato.

JavaScript

Gli eventi

Evento	Tag	Descrizione
mousemove		L'utente muove il mouse
mouseout	<a> <area>	L'utente allontana il cursore del mouse dal collegamento. Il cursore del mouse viene spostato fuori dalla mappa immagine.
mouseover	<a> < area>	Il cursore del mouse viene spostato su un collegamento. Il cursore del mouse viene spostato sull'area di una mappa immagine.
reset	<form>	Viene selezionato il pulsante Reset.
resize	<body> <frameset>	L'utente modifica le dimensioni della finestra. L'utente modifica le dimensioni del frame
select	<input type "text"> <textarea>	Il cursore del mouse viene spostato su un campo di testo. Il cursore del mouse viene spostato all'interno di un'area di testo.
submit	<form>	Viene selezionato il pulsante Submit.
unload	<body> <frameset>	L'utente esce dal documento. L'utente esce dal set di frame.

- ▶ Perchè il body possa sentire degli eventi del mouse, deve avere un testo o un'altezza impostata (`style="height:500px"`)

JavaScript

L'oggetto Event: propagazione

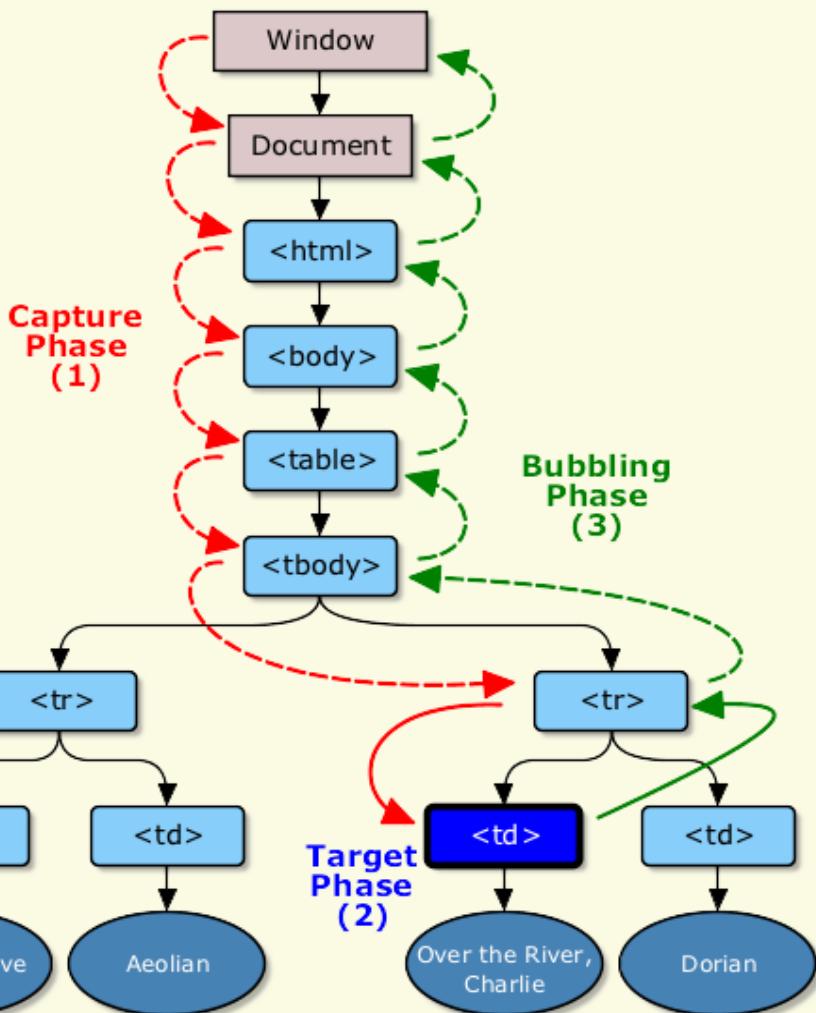
Secondo le specifiche del W3C, la propagazione di un evento avviene in tre fasi:

Capture phase o fase di cattura	In questa fase l'evento si propaga dalla radice del DOM verso l'elemento destinatario effettivo
Target phase	In questa fase l'evento raggiunge l'elemento destinatario
Bubble phase o fase di propagazione	Questa è la fase in cui l'evento risale l'albero del DOM partendo dall'elemento target fino a raggiungere la radice, passando quindi dagli stessi nodi attraversati nella fase di cattura.

Durante le 3 fasi, un oggetto *event* associato all'evento viene passato agli eventuali gestori incontrati lungo il cammino

JavaScript

L'oggetto Event: propagazione



- ▶ Normalmente, nella fase di **capturing** i gestori ignorano l'evento e lo fanno fluire verso l'elemento target.
- ▶ Una volta raggiunto l'elemento **target** viene eseguito il codice del gestore associato all'evento.
- ▶ Nella fase di **bubbling** vengono eseguiti i gestori dell'evento che si incontrano man mano che si va verso la radice del DOM

JavaScript

L'oggetto Event

- ▶ Il DOM prevede che ad ogni gestore di eventi venga passato come parametro l'oggetto **event** contenente informazioni su di esso.

```
function gestoreEvento(e) {  
    //e contiene un'istanza dell'oggetto event  
    e.target.id  
    //id dell'oggetto che ha scatenato l'evento  
}
```

JavaScript

L'oggetto Event

- ▶ Per assegnare un event handler attraverso l'HTML che gestisca l'oggetto *event*, occorre specificarlo tra i parametri passati

```
<p id="myP" onclick="gestoreEvento(event)">...
```

- ▶ Mentre non cambia nulla se l'evento è definito via Javascript (come funzione di callback o funzione anonima), e si potrà utilizzare invocandolo direttamente nel codice della funzione con **event**

```
document.getElementById("myP").onclick = gestoreEvento  
const gestoreEvento = function() {  
    alert(event.target.id)  
};
```

- ▶ Attenzione se ci sono dei parametri, allora va indicato prima di questi

```
const gestoreEvento = (e,s) => alert(e.target.id + s);
```

JavaScript

L'oggetto Event: Proprietà

- ▶ **target** restituisce un riferimento al nodo obiettivo del flusso d'evento nella sua fase di capturing (punto di partenza della successiva fase di bubbling)
- ▶ **currentTarget** restituisce un riferimento al nodo per il quale il flusso d'evento sta passando
- ▶ **relatedTarget** nel caso di un evento di *onmouseover*, la proprietà contiene un riferimento all'elemento dal quale il mouse proviene, cioè l'elemento appena lasciato. Nel caso dell'*onmouseout*, contiene un riferimento all'elemento verso il quale il mouse è diretto, ovvero l'elemento nel quale ci si sposta.

JavaScript

L'oggetto Event: Proprietà

- ▶ screenX, screenY restituisce la coordinata X/Y del puntatore del mouse relativa allo schermo
- ▶ x, y restituisce la coordinata X/Y rispetto alla finestra del browser
- ▶ button restituisce quale pulsante del mouse ha cambiato il proprio stato, ovvero sia stato premuto o rilasciato. I valori restituiti possono essere:
 - 0 : pulsante sinistro [1 per Netscape]
 - 1 : pulsante centrale se possiede 3 pulsanti [2 per Netscape]
 - 2 : pulsante destro [3 per Netscape]
- ▶ key carattere premuto da tastiera

JavaScript

L'oggetto Event: Proprietà

- ▶ `type` restituisce una stringa che descrive il tipo di evento, come ad esempio: "click", "mouseover", ecc
- ▶ `cancelable` indica se l'azione di default di un evento possa essere cancellata (true|false)
- ▶ `preventDefault()` consente di cancellare l'azione di default per quegli eventi per cui la proprietà `cancelable` restituisce il valore true
- ▶ `stopPropagation` questo metodo permette di interrompere la propagazione dell'evento, arrestando l'*Event Flow* indipendentemente che sia nella sua fase di capturing o in quella di bubbling

JavaScript

L'oggetto Event: esempio

```
<div id="mainDiv">  
    <p id="p1">Clicca su questo paragrafo</p>  
    <p id="p2">Altro paragrafo</p>  
</div>  
  
...  
  
var myDiv = document.getElementById("mainDiv");  
var myP = document.getElementById("p1");  
var handler = function() { alert(this.id) };  
myDiv.addEventListener("click", handler);  
myP.addEventListener("click", handler);
```

Cliccando su primo paragrafo otteniamo prima l'id del primo paragrafo e poi quello del <div>.

JavaScript

L'oggetto Event: esempio

- ▶ Per invertire l'ordine di gestione si usa un terzo parametro opzionale del metodo addEventListener() che abilita la gestione dell'evento nella fase di capturing

```
myDiv.addEventListener("click", handler, true);
```

- ▶ Per fare in modo che venga eseguito solo un gestore dell'evento si blocca la propagazione

```
var handler = function(e) {  
    console.log(this.id);  
    //this elemento su cui si è verificato l'evento  
    e.stopPropagation();  
};
```

JavaScript

Drag & drop: eventi

Evento	Tag	Descrizione
ondragstart	tutti gli elementi con attributo draggable="true"	si sta trascinando l'elemento
ondragover	tutti gli elementi con attributo draggable="true"	quando un elemento viene trascinato su una destinazione di rilascio valida
ondrop	tutti gli elementi con attributo draggable="true"	si è rilasciato un elemento sopra questo

- `.dataTransfer` Restituisce un oggetto contenente i dati trascinati/rilasciati o inseriti/eliminati. Il metodo `dataTransfer.setData("key", value)` imposta una coppia chiave/valore e `getData("key")` restituisce il valore.
- Per impostazione predefinita, gli elementi trascinati non possono essere rilasciati in altri elementi. Per consentire un inserimento, dobbiamo impedire la gestione predefinita dell'elemento trascinato e dell'elemento "ricevente". Si deve chiamare per entrambi il metodo `e.preventDefault()`

JavaScript

Drag & drop: esempio

Trascina la scritta nell'altro paragrafo

```
<script>  
function allowDrop(e) {e.preventDefault();}  
function drag(e) {  
    e.dataTransfer.setData("text", e.target.id);}  
function drop(e) {  
    e.preventDefault();  
    var data = e.dataTransfer.getData("text");  
    e.target.appendChild(document.getElementById(data));}  
</script>  
  
<div id="div1" ondrop="drop(event)"  
     ondragover="allowDrop(event)"></div>  
<div id="drag1" draggable="true"  
     ondragstart="drag(event)">ciao</div>
```

I file

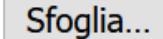
Javascript lato client

input type="file"

- ▶ Per motivi di sicurezza non è possibile leggere o scrivere file sul computer del client, senza autorizzazione
- ▶ Con HTML5 si può leggere un file dopo che l'utente l'ha selezionato con una `<input type = "file">`

```
<input type="file" id="fileSelector"  
accept=".jpg, .jpeg, .png">
```

- ▶ Viene definito una label contenente il nome del file selezionato tramite il pulsante "Sfoglia"

 Sfoglia... Nessun file selezionato.

- ▶ l'attributo accept limita i tipi del file selezionabile
- ▶ l'attributo multiple permette selezioni multiple

Javascript laterale

File API

Un **MIME type** (Multipurpose Internet Mail Extensions) è uno standard che indica la natura e il formato di un documento o di un file o insieme di byte: text/html o video/mp4

- ▶ Sono un'interfaccia per l'accesso in lettura ai file selezionati dall'utente ed al loro contenuto.
- ▶ Le API definiscono tre tipi di oggetto:
 - ▶ **FileList** Rappresenta una lista di file. Possiede la proprietà `.length`
 - ▶ **File** Rappresenta un singolo file. Possiede le proprietà `.name` `.size` `.type` (tipo MIME del file, se è possibile determinarlo, se no stringa vuota)
`.lastModified` (in mill)
 - ▶ **FileReader** È l'oggetto che consente l'accesso in lettura al contenuto del file

Javascript lato client

FileList

- ▶ Dopo che l'utente ha selezionato un file, l'oggetto `input_file` contiene nella proprietà `.files` (di tipo `FileList`), la lista di file selezionati. Ogni elemento della lista è di tipo `File`

```
<input id="fileSelector" multiple type="file"  
onchange="pullfiles()">
```

```
function pullfiles() {  
    let files = fileSelector.files  
    let s = "Selezionati " + files.length + "  
    file:\n"  
    for (f of files) //files[k]  
        s += f.name +" "+ f.size + "\n"  
    alert (s)  
}
```

Javascript lato client

FileReader

- ▶ L'oggetto `FileReader` consente alle applicazioni Web di leggere in modo asincrono il contenuto dei file memorizzati sul computer dell'utente, utilizzando gli oggetti `File`
- ▶ Gli oggetti `File` possono essere ottenuti da un oggetto `FileList` restituito a seguito della selezione di file da parte dell'utente mediante l'elemento `<input>`, o dall'oggetto `DataTransfer` di un'operazione di trascinamento della selezione o dall'API `mozGetAsFile()` su un `HTMLCanvasElement`.

Javascript lato client

FileReader

- ▶ `onerror` l'evento si genera quando si verifica un errore durante l'operazione di lettura.
- ▶ `onload` l'evento si genera quando l'operazione di lettura viene completata correttamente.
- ▶ `FileReader.readAsText()` inizia a leggere il contenuto del file specificato, una volta terminato, l'attributo `result` contiene il contenuto del file come stringa di testo. È possibile specificare un nome di codifica.
- ▶ `FileReader.readAsDataURL()` inizia a leggere il file specificato, interpretandolo come data URL (rappresentazione in bas64 del contenuto del file)

Javascript lato client

FileReader

1. Si deve creare un'istanza dell'oggetto **FileReader**,
2. Assegnare un **gestore** all'evento **load** dell'oggetto
3. Invocare il metodo **readAsText()**/**readAsDataURL()** passandogli l'oggetto *File* da leggere.
4. Al termine del caricamento del file viene generato l'evento **load**, in corrispondenza del quale si accede al contenuto del file tramite la proprietà **result** di **target** messo a disposizione dall'oggetto **event**.

Javascript lato client

FileReader di testo

```
function caricaTesto(event) {  
    // prende il primo file letto  
    const file = event.target.files[0];  
  
    const reader = new FileReader();  
    // si assegna il gestore dell'evento load: se  
    la lettura va a buon fine esegue la  
    funzione anonima che visualizza il  
    contenuto  
    reader.addEventListener('load', event => {  
        alert(event.target.result)  
    });  
    // legge il file come un file di testo  
    reader.readAsText(file);
```

Javascript lato client

FileReader di immagini

```
function caricaImmagine(event) {  
    // prende il primo file letto  
    const file = event.target.files[0];  
  
    const reader = new FileReader();  
    // legge il file come un file di testo  
    reader.readAsDataURL(file);  
  
    // se la lettura va a buon fine esegue la  
    // funzione anonima che visualizza il  
    // contenuto  
    reader.addEventListener('load', event => {  
        img.src = event.target.result;  
    }) ;
```



Approfondimento

Linguaggi WEB lato CLIENT

334

334

JavaScript

Oggetti Map

- ▶ L'oggetto Map contiene delle coppie di chiave-valore. Qualsiasi valore (sia oggetti che primitivi) possono essere utilizzati sia come chiavi che valori.

```
const map1 = new Map();
map1.set('a', 1);
map1.set('b', 2);
map1.set('c', 3);
console.log(map1.get('a')) // 1
map1.set('a', 97);
console.log(map1.get('a')) // 97
console.log(map1.size) // 3
```

JavaScript

Oggetti Set

- ▶ Gli oggetti **Set** sono collezioni di valori. Ogni valore può essere presente una sola volta, ciò rende il Set una collezioni di valori univoci.

```
const mySet1 = new Set()
```

```
mySet1.add(1) // Set [ 1 ]
```

```
mySet1.add(5) // Set [ 1, 5 ]
```

```
mySet1.add(5) // Set [ 1, 5 ]
```

```
mySet1.add('some text') // Set [ 1, 5, 'some text' ]
```

```
mySet1.has(1) // true
```

```
mySet1.has(3) // false, since 3 has not been added to the set
```

```
mySet1.size // 3
```



Promise Async/await

callback

- In caso di programmi complessi l'utilizzo delle callback porta a scrivere codice del tipo detto "callback hell"

Callback Hell

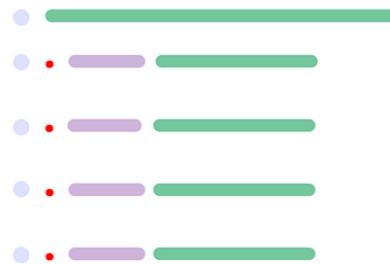
```
setTimeout(()=>{
    console.log("prendo l'ordine")
    setTimeout(()=>{
        console.log(`prendo il cono`)
        setTimeout(()=>{
            console.log(`metto il gelato sul cono`)
            setTimeout(()=>{
                console.log("gelato servito")
            },500)
        },3000)
    },500)
},2000);
```



promise

- ▶ Nascono per risolvere il "callback hell"
- ▶ Un promise è un oggetto JavaScript che ti consente di effettuare chiamate asincrone.
- ▶ Produce un valore quando l'operazione asincrona viene completata correttamente o genera un errore se non viene completata.
- ▶ Le Promises sono la base della programmazione asincrona del moderno JavaScript.

Promises Format



promise creazione

```
let promise = new Promise(function(resolve, reject) {  
    // Do something and either resolve or reject  
});
```

- ▶ Dobbiamo passare al costruttore una funzione (**executor function**), che accetta due argomenti (resolve, reject), che sono funzioni di callback che restituiscono il risultato. (le funzioni si possono chiamare come vogliamo)
- ▶ Nel corpo della promise, una chiamata riuscita è indicata dalla chiamata della prima funzione resolve, mentre gli errori sono indicati dalla chiamata della seconda funzione reject. Queste funzioni possono essere chiamate una sola volta.

promise esempio sincrono

```
let myPromise = new Promise((resolve, reject) => {
  let randomNumber = Math.floor(Math.random() * 2);
  if (randomNumber === 1) {
    //nella () il risultato restituito
    resolve('risolta! ' + randomNumber);
  } else { //nella () il risultato restituito
    reject(new Error('rigettata! ' + randomNumber));
  }
});
myPromise.then((res) => {//res è il risultato della resolve
  console.log('La Promise è: ' + res);
}).catch((err) => {//err è il risultato della reject
  console.log('La Promise è: ' + err.message);
}).finally(() => {console.log('fine')});//eseguita in ogni caso
);
```

promise

- ▶ Altri modi per dichiarare una promessa

```
let promise = () =>{
    return new Promise( (resolve, reject)=>{
        // codice } )
}
```

oppure

```
function promise(){
    return new Promise( (resolve, reject) =>{
        // codice } )
}
```

```
promise().then() //esecuzione
```

promise esempio asincrono

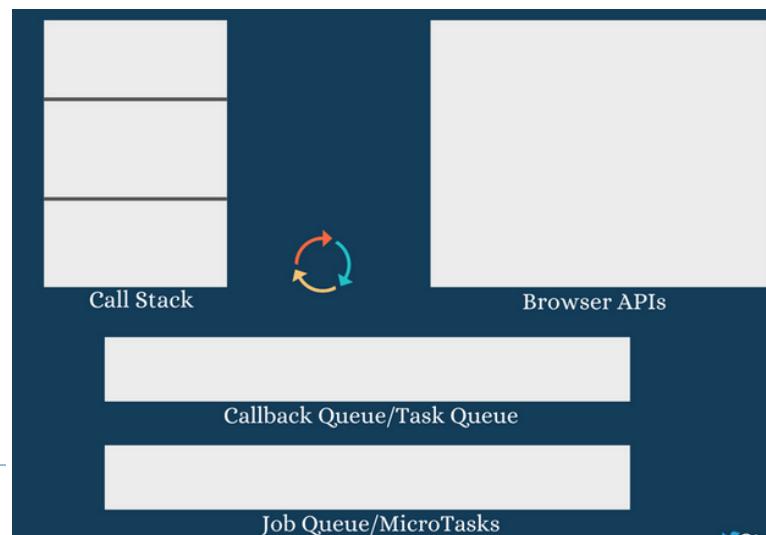
```
function doSomethingAsync(val) {  
    return new Promise((resolve, reject) => {  
        setTimeout(() => resolve(val), 1000);  
    });  
  
    console.log('Prima della Promise');  
  
    doSomethingAsync(3).then((result) => {  
        console.log('La Promise ha restituito:', result);  
    });  
  
    console.log('Dopo la Promise');
```

promise: priorità d'esecuzione

- ▶ Possiamo classificare la maggior parte delle operazioni asincrone in JavaScript come:
 - ▶ **Eventi o funzioni di API browser/API Web.** Questi includono metodi come setTimeout o gestori come click, mouse over, scroll e altri. Sono considerati **macrotask** e hanno minore priorità.
 - ▶ **Promise.** Un oggetto JavaScript unico che consente di svolgere operazioni asincrone. Sono considerate **microtask** e hanno maggior priorità.

promise: priorità d'esecuzione

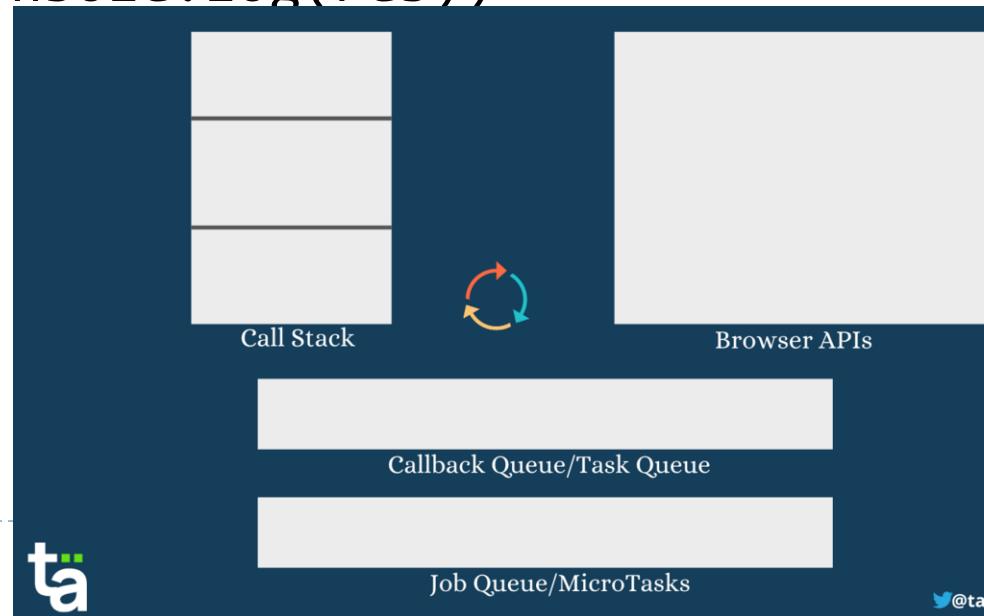
- ▶ Per eseguire le promise, il motore JavaScript non usa la stessa callback queue come per le API del browser. Usa un'altra coda chiamata **the Job Queue o coda dei lavori**.
- ▶ Ogni volta che si è in presenza di una promise, la funzione entra nella coda dei lavori. Il loop degli eventi funziona come al solito, ma, per esaminare le code quando la stack è libera, dà la priorità alla job queue dei microtask rispetto alla callback queue dei macrotask.



JS

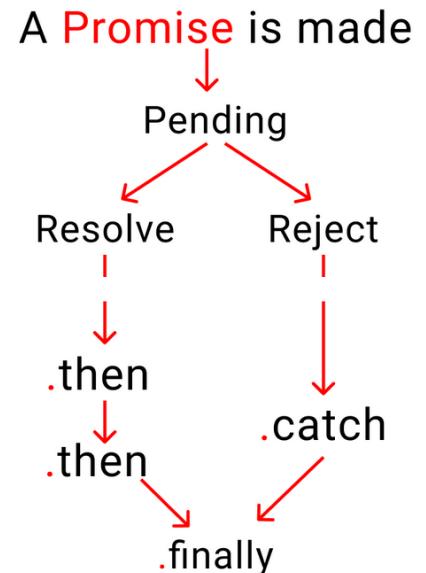
promise: esecuzione

```
function f1() {console.log('f1');}
function f2() {console.log('f2');}
function main() {
    console.log('main');
    setTimeout(f1, 0);
    new Promise((resolve, reject) =>
        resolve('I am a promise')
    ).then((res) => console.log(res))
    f2();
}
main();
//si ottiene
main
f2
I am a promise
f1
```



promise: proprietà state

- ▶ Un oggetto promise ha una proprietà interna state che può essere:
 - **Pending:** è lo stato iniziale: la richiesta di esecuzione di un'attività asincrona è partita ma non abbiamo ancora ricevuto un risultato
 - **Resolved (risolta) o fulfilled:** vuol dire che è andato a buon fine: l'attività asincrona restituisce un valore
 - **Rejected (respinta):** vuol dire che qualcosa è andato storto: perché si è verificata un'eccezione oppure perché il valore restituito non è considerato valido
- ▶ Una promessa che viene risolta o respinta è chiamata **settled (saldata)**.
- ▶ La proprietà determina quale metodo viene eseguito se il `then()` o il `catch()`.
In ogni caso si esegue il `finally`



promise: proprietà result

- ▶ Un oggetto Promise ha la proprietà `result` che può avere i seguenti valori:
 - ▶ **undefined** : all'inizio, quando il valore dello stato è pending.
 - ▶ **valore** : quando la promessa viene risolta (value).
 - ▶ **errore** : quando la promessa viene respinta (error).
- ▶ Queste proprietà sono interne. Sono inaccessibili al codice, ma sono ispezionabili: saremo in grado di ispezionare il valore di state e result utilizzando uno strumento di debugger, ma non saremo in grado di accedervi direttamente utilizzando il programma.

The screenshot shows a browser's developer tools with the "Console" tab selected. On the left, the file tree shows "main.js" is the active script. In the main pane, a promise object is being inspected. The promise has the following properties:

- `__proto__: Promise`
- `[[PromiseState]]: "fulfilled"`
- `[[PromiseResult]]: "{"count":1117, "next": "https://api.pokemongo.net�/pokemons?limit=1&offset=1117"}"`

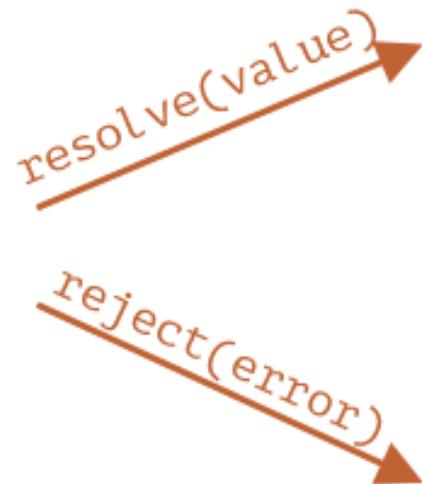
The code in "main.js" is partially visible, showing a function that returns a promise and logs its result to the console.

```
// This will resolve
let promise = getPromise();
promise.then(
  (result) => {
    res.set = { count: 1117, next: "https://api.pokemongo.net�/pokemons?limit=1&offset=1117" };
    console.log(promise);
    console.log(result);
  }
);
```

promise: state result

```
new Promise(executor)
```

state: "pending"
result: undefined



state: "fulfilled"
result: value

state: "rejected"
result: error

promise .then()

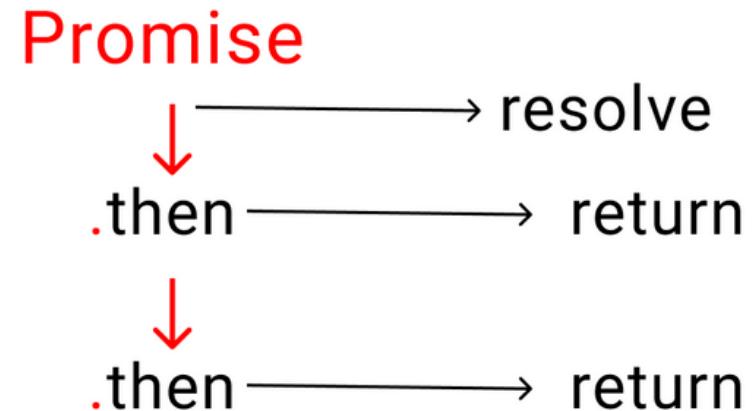
- ▶ `then()` permette di conoscere l'esito di una promessa.
- ▶ Restituisce sempre una Promise, per cui è possibile creare una catena di Promise (promise chain)
- ▶ Accetta due funzioni come argomenti (`result`, `error`). La prima è la funzione che verrà eseguita nel caso in cui la promise venga risolta, la seconda verrà eseguita se la promise viene rigettata. Si può anche passarne solo una:

```
promise.then(  
  (result) => { console.log(result); }  
  
  ,  
  (error) => { console.log(error); }  
);
```

promise .then()

- ▶ Viene svolto solo se la **promise** è **risolta** e si può:
 - ▶ ritornare un'altra promessa
 - ▶ restituire un valore incluso il valore `undefined`
 - ▶ generare un errore con `throw`

```
funzPrinc()  
.then(funz1)  
.then(funz2);
```



ovvero

```
funzPrinc().then(funz1).then(funz2);
```

promise .then()

```
const prendiOrdine = () => {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("prendo l'ordine");
      resolve();                                [...]
    }, 2000);
  });
};

function prendiCono() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log(`prendo il cono`);
      resolve();
    }, 500);
  });
}

prendiOrdine()
  .then(() => prendiCono())
  .then(() => mettiGelato())
  .then(() => gelatoServito());
```

promise .catch()

- ▶ Questo metodo viene svolto solo se la **promise** è **reject**.
- ▶ Si deve mettere dopo il then()
- ▶ Si usa se si gestisce in un solo modo il fallimento della promessa. fallisce

```
.catch(()=>{  
    //faccio qualcosa  
})  
funzPrinc()  
.then(funz1)  
.then(funz2)  
.catch(funz3);
```

ovvero

```
funzPrinc().then(funz1).then(funz2).catch(funz3);
```

promise .finally()

- ▶ Il metodo **.finally()** viene eseguito indipendentemente dal fatto che la promise sia risolta o respinta.

```
.finally(()=>{  
    //faccio qualcosa  
})
```

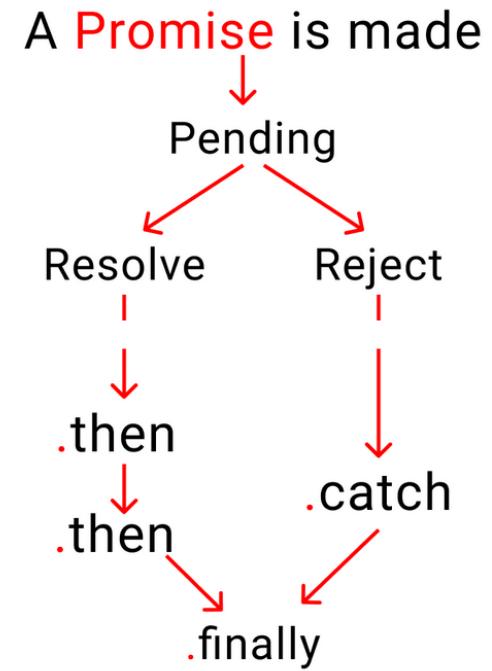
```
funzPrinc()  
.then(funz1)  
.then(funz2)  
.catch(funz3)  
.finally(funz4);
```

ovvero

```
funzPrinc().then(funz1).then(funz2).catch(funz3).finally(funz4);
```

promise in conclusione

```
function funz(){  
    return new Promise ((resolve, reject)=>{  
        if(true){  
            resolve("promise soddisfatta")  
        }else{  
            reject("c'è un errore")  
        }  
    })  
}  
funz() // esegui codice  
.then() // prossimo step  
.then() // prossimo step  
.catch() // c'è un errore  
.finally() // fine della promise [optional]
```



promise in conclusione

```
const promise = new Promise((resolve, reject) => {
    // fa una richiesta di rete
    if (response.status === 200) {
        resolve(response.body)
    } else {
        const error = { ... }
        reject(error)
    }
})
```



```
promise.then(res => {
    console.log(res)
}).catch(err => {
    console.log(err)
})
```

promise: Promise.all()

- ▶ Alle volte è necessario combinare delle promise, invece che eseguirle una dopo l'altra. Ad esempio è necessario che più promise vengano risolte ma non dipendono una dall'altra. In un caso del genere è possibile lanciarle tutte contemporaneamente ed essere poi notificati quando tutte le promises vengono risolte.
- ▶ Il metodo **Promise.all()** consente proprio questo: accetta un array di promises, e ritorna una singola promise con un array di valori nel metodo then.

async/await

- ▶ **Async/Await** permette di scrivere codice asincrono in modo "sincrono", in modo più leggibile e scalabile delle Promise. (ES2017)
- ▶ Quando si definisce una funzione come `async`, restituirà sempre una promessa (anche se non esplicitato nel codice, lo fa JavaScript).
- ▶ La parola chiave `async` consente di dichiarare una funzione come asincrona, cioè che contiene un'operazione asincrona.

```
function funz(){  
    return new Promise( (resolve, reject) =>{ // codice } )  
}
```

usando **async**

```
async function funz() {  
    // codice in cui possiamo usare await  
}
```

oppure

```
const funz = async () => {  
    //codice in cui possiamo usare await  
}
```

async/await

- ▶ La parola chiave **await** fa sì che JavaScript attenda fino a che il risultato della promise non è stabilito e lo restituisce
- ▶ await sospende l'esecuzione di una funzione in attesa che la Promise associata ad un'attività asincrona venga risolta o rigettata. (in realtà si può anche usare con una funzione sincrona)
- ▶ la parola chiave await può essere usata soltanto all'interno di funzioni marcate con async.
- ▶ Le funzioni `async` utilizzano `try...catch` per gestire gli errori.
- ▶ Insieme ad `async/await`, puoi usare anche i gestori `.then()`, `.catch()` e `.finally()`, che sono una parte centrale delle promise.

async/await

```
async function funz(){
    try{
        // codice
        await abc(); // si attende l'esito della funzione
    }
    catch(errore){
        console.log("abc non esiste", errore)
    }
    finally{
        console.log("il codice viene eseguito comunque")
    }
}
funz() // esegue codice
```

async/await

```
function syncF() {  
    return new Promise((resolve, reject) => {  
        let randomNumber = Math.floor(Math.random() * 2);  
        if (randomNumber === 1) { //nelle () il risultato restituito  
            resolve('risolta! ' + randomNumber);  
        } else { //nelle () il risultato restituito  
            reject(new Error('rigettata! ' + randomNumber));  
        }  
    });  
};  
  
async function runSyncF() {  
    try {  
        const res = await syncF();  
        console.log('La Promise è: ' + res)  
    } catch (err) {  
        console.log('La Promise è: ' + err.message);  
    } finally { //eseguita in ogni caso  
        console.log('fine');  
    }  
}  
runSyncF();
```

async/await

```
function syncFunction() {  
    return new Promise((resolve) => {  
        setTimeout(() => {resolve("Risposta sincrona");}, 1000);  
    });  
}  
  
async function runSyncCode() {  
    try {  
        const result = await syncFunction();  
        console.log(result);  
    } catch (error) {console.error(error.message);}  
}  
  
runSyncCode();
```



API WEB
fetch



API Web

- ▶ **API** (Application Programming Interface): insieme di regole che guidano il modo in cui un software o un sistema comunica con un altro.
- ▶ Le **API Web** sono strumenti integrati nel browser e utilizzabili dal motore Javascript quando il codice viene eseguito all'interno di un browser. Sono per esempio:
 - ▶ Metodi di manipolazione del DOM
 - ▶ Metodi legati al tempo `setTimeout()`
 - ▶ Gestori degli eventi
 - ▶ `fetch()`
- ▶ Anche se usati nel codice js, possono essere asincroni, poiché lavorano in thread separati appartenenti al browser. Quando una chiamata API è pronta, la sua callback viene inserita nella coda dei task.

API fetch: richiesta GET

- ▶ **fetch()** è un meccanismo che ti consente di effettuare semplici chiamate AJAX (Asynchronous JavaScript and XML) con JavaScript.
- ▶ Asincrono significa che puoi usare **fetch** per effettuare una chiamata a un'API esterna senza interrompere l'esecuzione di altre istruzioni. In questo modo, le altre funzioni del sito continueranno a essere eseguite anche quando una chiamata API non è stata risolta.
- ▶ Quando una risposta (dati) viene restituita dall'API, le attività asincrone (fetch) riprendono. Il corpo della risposta può essere ad esempio un file json.
- ▶ Quando parliamo di API, dobbiamo anche parlare di endpoint. Un **endpoint** è semplicemente un URL univoco che chiavi per interagire con un altro sistema. Es di richiesta GET

```
fetch('https://esempio.it/api/posts/1');
```

API fetch risposta

- ▶ l'oggetto in risposta alla **fetch()** contiene un bel po' di informazioni oltre il corpo, incluso il *codice di stato*, le *intestazioni* e altre *informazioni*.
- ▶ L'API di gestione della fetch **restituisce una promise**: pertanto è necessario annidare un metodo **then()** per gestire la risoluzione.
- ▶ I dati restituiti dall'API non sono generalmente in una forma utilizzabile. Quindi si convertono usando il metodo **json()** per poterli utilizzare:

```
const url = "https://www.thecocktaildb.com/api/json/v1/1/random.php"
fetch(url)
  .then(response => response.json()) //converte in json
  .then(data => {console.log(data)})
  .catch(err => {console.log(err)});
```

ATTENZIONE: perché il codice funzionideve essere eseguito in VS code → "Open in Live Server", dopo averne installato l'estensione

API fetch con async

```
const getData = async () => {
  try {
    const response =
      await fetch("https://www.thecocktaildb.com/api/json/v1/1/random.php")
    const data = await response.json()
    console.log(data)
  } catch (err) {
    console.log(err)
  }
}
getData()
```

Se la richiesta ha esito positivo, riceve un corpo di risposta contenente l'oggetto json corrispondente. La risposta varierà a seconda di come è impostata l'API.

Esecuzione asincrona

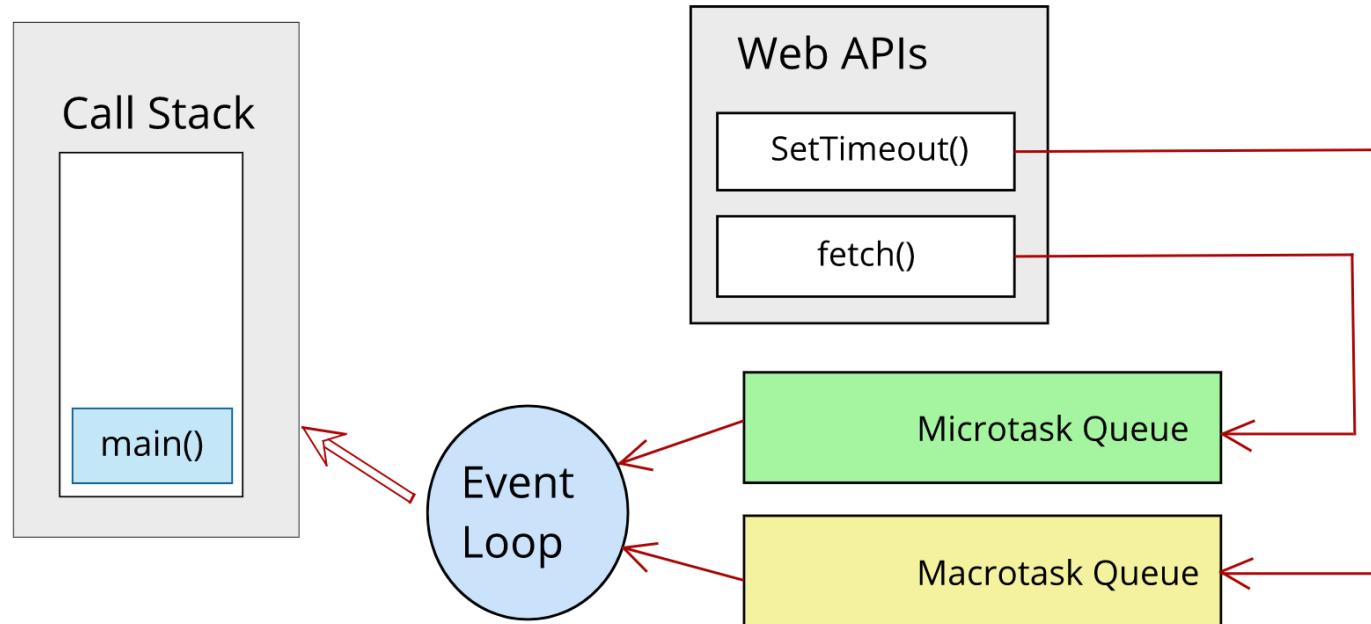
- ▶ Mentre nei linguaggi di programmazione che supportano la concorrenza una porzione di codice di un thread può essere interrotta per mandare avanti l'esecuzione di un altro thread, in JavaScript tutto avviene in un unico thread. Il modello di concorrenza in base al quale abbiamo l'illusione che più thread siano in esecuzione è quello dell'**event loop**: ogni evento inserisce un messaggio in una coda che viene elaborata sequenzialmente dal runtime di JavaScript in un ciclo infinito.
- ▶ In pratica, un motore JavaScript non fa altro che verificare la presenza di messaggi nella coda ed eseguire il codice dell'eventuale gestore per passare poi al messaggio successivo. Il codice eseguito tra un messaggio ed il successivo viene eseguito senza interruzioni. Qualsiasi evento che si verifica durante l'esecuzione di un ciclo dell'event loop non può interromperlo.

Priorità esecuzione asincrona

- ▶ Non tutte le operazioni hanno la stessa priorità nell'ambiente di esecuzione. Il browser quando deve eseguire del codice asincrono, lo collocherà in una delle due code in base al tipo di lavoro che riceve.
 - ▶ Le **promise** vengono inserite nella coda del microtask e hanno una priorità più alta.
 - ▶ **API del browser/API Web Eventi o funzioni** come setTimeout vengono inserite nella coda delle macrotask e hanno una priorità inferiore.
- ▶ Solo quando il motore JavaScript ha finito di eseguire tutto il suo codice sincrono nella call stack, il ciclo di eventi (event loop) inizierà a prelevare dalle code secondo la loro priorità e a inserirle per la loro esecuzione nella call stack.

Priorità di fetch

- ▶ Ricordandoci che una fetch restituisce una promise, la sua risposta verrà messa nella coda dei microtask



Priorità di fetch

```
for (i = 0; i < 10000; i++)  
    window.setTimeout(() => console.log("A"), 0)  
for (i = 0; i < 10000; i++)  
    window.setTimeout(() => console.log("B"), 0)  
fetch("https://www.thecocktaildb.com/api/json/v1/1/random.php")  
.then(response => response.json())  
.then(data => console.log(data))
```

A console i dati della fetch compaiono appena sono disponibili perché hanno una priorità maggiore rispetto alla callback del setTimeout. Si può ottenere (dipende dalla velocità di esecuzione e dalla connessione):

A //10000 A
B // un po' di B
dati dall'api
B //i rimanenti

Event Loop

Event loop : call stack

- ▶ L'Event Loop è il meccanismo con cui JavaScript gestisce l'asincronia e il non-blocco dell'esecuzione del codice in un ambiente mono-thread. L'event loop si occupa di:

I. Gestione della Pila di Esecuzione o **CALL STACK**:

L'Event Loop monitora costantemente la call stack, che tiene traccia delle funzioni in esecuzione. Se una funzione è sincrona e viene richiamata, allora viene inserita nella call stack e, quando è il suo turno di esecuzione, viene eseguita e il controllo rimane all'interno di quella funzione fino a quando non viene completata. Quando una funzione viene completata, viene rimossa dalla Pila di Esecuzione.

Event loop: coda di task

2. Gestione della Coda dei Task di Callback

(Macrotask): Le callback sono funzioni che vengono passate come argomenti ad altre funzioni e vengono eseguite in seguito di un evento o di un'operazione asincrona. Le API browser/web utilizzano delle funzioni callback per completare le attività che richiedono operazioni asincrone come i timer o richieste HTTP. Le callback vengono invocate quando l'evento o l'operazione asincrona è completata e non vengono immediatamente inserite nella call stack, ma vengono inserite nella coda di callback o coda dei (macro)task. Questa coda contiene i gestori degli eventi che sono pronti per essere elaborati, nell'ordine con cui si verificano, ma non vengono eseguiti immediatamente.

Event loop: coda dei microtask

3. **Gestione delle Promises (Microtask):** Quando una Promise viene risolta o rigettata, la sua callback (executor) viene inserita nella Coda dei microtask (Job Queue) che ha una priorità maggiore rispetto alla coda dei Task. Questa coda viene elaborata finché non diventa vuota.
4. **Esecuzione delle funzioni :** Quando la Pila di Esecuzione è vuota, l'Event Loop preleva il prossimo Task (Microtask o Macrotask) dalla rispettiva coda e lo inserisce nella call stack per l'esecuzione, tenendo conto delle priorità: i macrotask saranno prelevati solo se è vuota le job queue

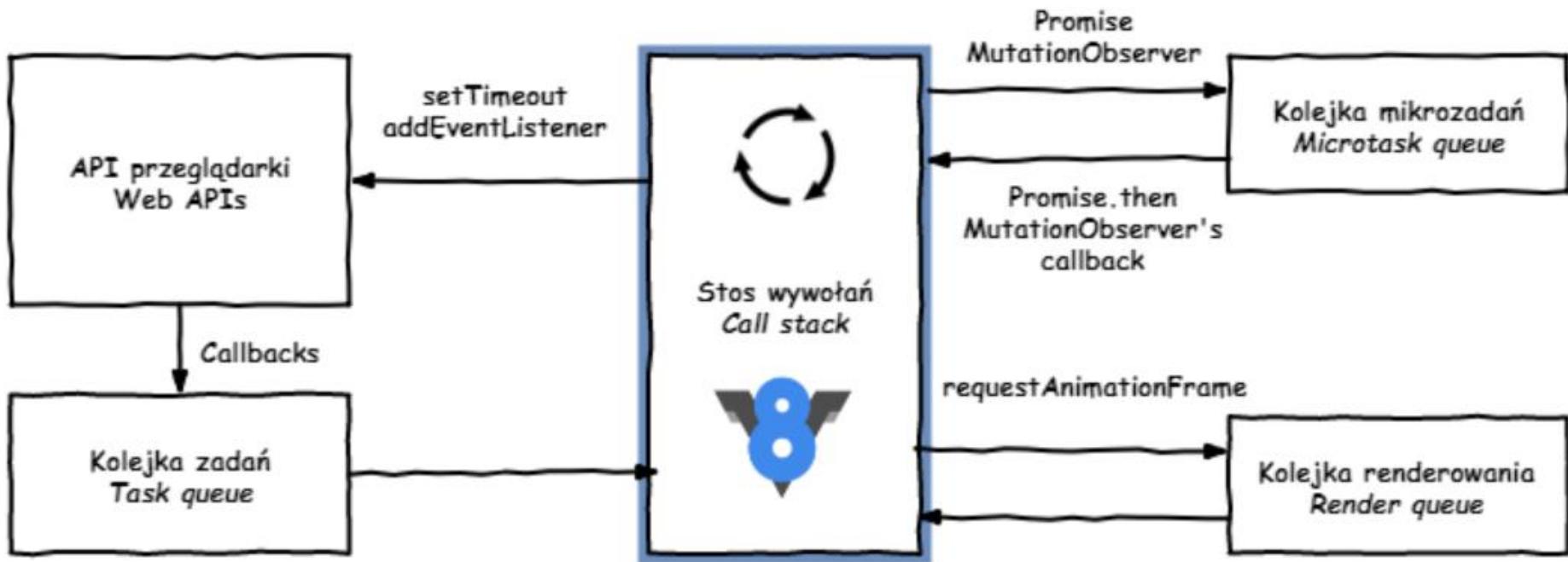
L'Event Loop, con il meccanismo descritto, assicura che le funzioni vengano eseguite in modo non bloccante, consentendo al codice di gestire gli eventi, le operazioni asincrone e le Promise in modo efficiente.

Event Loop: coda di rendering

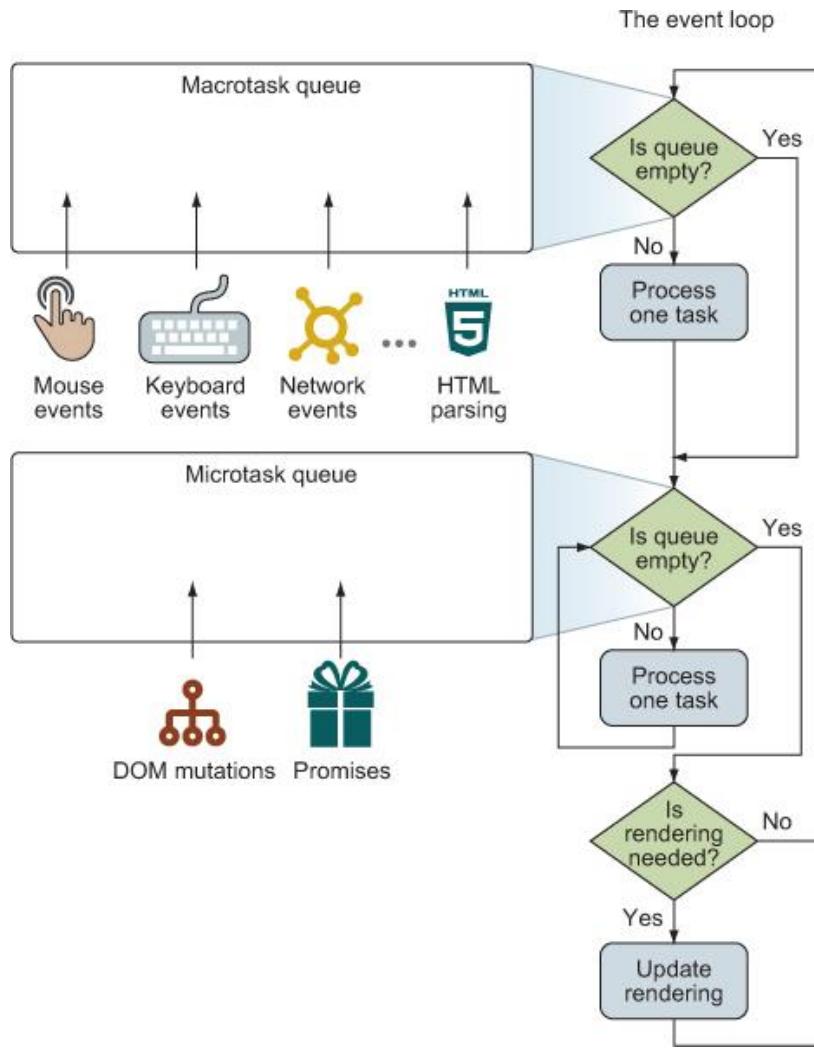
Per offrire un'esperienza fluida, il browser deve aggiornare le pagine Web circa 60 volte al secondo. L'event loop deve anche gestire una **coda di rendering**.

- ▶ La coda di rendering contiene attività che verranno eseguite prima che si verifichi il successivo rendering della pagina.
- ▶ Se si vuole inserire del codice in questa coda (ad esempio per ricalcolare la posizione di un oggetto) si deve usare la funzione `requestAnimationFrame()`.
- ▶ La coda di rendering viene elaborata finché non diventa vuota, come la coda dei microtask.

Event loop: code e call stack



Event Loop: ciclo completo



1. Se c'è qualcosa nella coda dei task, scegli il primo elemento e mettilo nella call stack .
2. Elabora la stack di chiamate finché non diventa vuota.
3. Elabora la coda dei microtask finché non diventa vuota.
4. Elabora le istruzioni provenienti da requestAnimationFrame, ricalcola gli stili, esegui il rendering della pagina.



Approfondimento

Linguaggi WEB lato **CLIENT**

388

388

Fetch()

- ▶ L'utilizzo di XMLHttpRequest per la gestione di chiamate HTTP da JavaScript risulta abbastanza prolioso e scomodo. È chiaramente questa una delle principali ragioni del successo di metodi alternativi, come ad esempio quello offerto da jQuery tramite `$.ajax()`. Per questo ed altri motivi, il gruppo di lavoro WHATWG ha definito recentemente una alternativa a XMLHttpRequest: l'API **fetch()**.
- ▶ Rispetto a XMLHttpRequest, fetch() ha una sintassi più semplice e meglio integrata nel modello ad oggetti di JavaScript. L'API prevede una gestione delle chiamate asincrone basata sulle *promise* ed è pensata per essere estesa ed utilizzabile in diversi contesti, non solo all'interno del browser

Fetch()

```
fetch("http://www.html.it")
  .then(response => { console.log(response); })
  .catch(error => console.log("Si è verificato un
    errore!"))
```

abbiamo specificato l'URL su cui effettuare la richiesta HTTP ed abbiamo gestito la risposta come una *promise*. In caso di successo la *promise* verrà risolta ed entreremo nel ramo `then()`, in cui ci verrà fornita la risposta del server sotto forma di oggetto di tipo `Response`, che possiede i seguenti attributi:

- ▶ **status**: int = codice di stato HTTP inviato dal server(200 = 'OK')
- ▶ **statusText**: string, descrive testualmente il significato. ("OK")
- ▶ **ok** :bool indica se la risposta del server è stata positiva, cioè se il codice di stato restituito è compreso tra 200 e 299, estremi inclusi

Fetch()

```
fetch("http://www.html.it").then(response => {
  if (response.ok) {
    console.log("Contenuto ricevuto"); }
  if (response.status >= 100 && response.status < 200) {
    console.log("Informazioni per il client"); }
  if (response.status >= 300 && response.status < 399) {
    console.log("Redirezione"); }
  if (response.status >= 400 && response.status < 499) {
    console.log("Richiesta errata"); }
  if (response.status >= 500 && response.status < 599) {
    console.log("Errore sul server"); }
}).catch(error => console.log("Errore!"))
```

Anche la condizione d'errore sul server (codici di stato compresi tra 500 e 599) determina la risoluzione positiva della *promise* generata da `fetch()`

Response

- ▶ L'oggetto response ha i seguenti metodi
 - ▶ **text()** Restituisce il contenuto sotto forma di testo
 - ▶ **json()** Effettua il parsing del contenuto e lo restituisce sotto forma di oggetto
 - ▶ **blob()** Restituisce il contenuto sotto forma di dati non strutturati (*blob*)
 - ▶ **arrayBuffer()** Restituisce il contenuto strutturato in un *arrayBuffer*

```
fetch("https://www.html.it/api/articoli/123")
  .then(response => {
    if (response.ok) {return response.json();}
  }).then(articolo =>
  console.log(articolo.titolo)).catch(error =>
  console.log("Si è verificato un errore!"))
```

▶ 392

Gestione di funzioni asincrone tramite approccio sincrono

Gli approcci generalmente più utilizzati per l'esecuzione di operazioni asincrone in JavaScript sono:

- ▶ le funzioni di **callback**, cioè funzioni passate come parametri di altre funzioni da eseguire al termine di una operazione asincrona;
- ▶ le **Promise**, cioè oggetti il cui stato rappresenta lo stato di esecuzione di una attività asincrona. (più semolice rispetto callback) Dalle specifiche di ECMAScript 2015, le promise sono diventate **un componente nativo del linguaggio**. Questa importante novità offre sia il vantaggio di non dipendere da librerie esterne, sia la comodità di stabilire un'API standard per la gestione di codice asincrono basato sul *Promise Pattern*.



Promise Pattern

- ▶ Secondo questo pattern, una **promise** è un oggetto che rappresenta il risultato pendente di un'operazione asincrona. Questo oggetto può essere usato per definire le attività da eseguire dopo che l'esecuzione asincrona è terminata.
- ▶ I metodi **done()** e **fail()** di una promise consentono di specificare cosa fare dopo aver ottenuto il risultato positivo o negativo dell'elaborazione asincrona. Questi stessi metodi restituiscono delle promise, così possiamo concatenare più **done()** per mettere in sequenza operazioni asincrone in maniera più leggibile delle callback annidate.
- ▶ Q libreria che implementa le promise, i framework **jQuery** o **AngularJS**, forniscono un supporto integrato.

Promise Pattern

```
var promise = getMessaggio();
promise.done(sendMessaggio).done(function() {
    console.log("Messaggio inoltrato");
});
promise.fail(function(err) {
    console.log("Errore: " + err.message);
});
```

sia la `getMessaggio()` che la `sendMessaggio()` restituiscono una promise, cioè un oggetto che rappresenta l'esito di un'operazione asincrona.

Promise: stato

Il suo stato può essere

- ▶ **resolved** (risolta) Una promise è risolta quando il valore che rappresenta diviene disponibile, cioè quando l'attività asincrona restituisce un valore
- ▶ **rejected** (rigettata) Una promise è rigettata quando l'attività asincrona associata non restituisce un valore o perché si è verificata un'eccezione oppure perché il valore restituito non è considerato valido
- ▶ **pending** (pendente) Una promise è pendente quando non è né risolta né rigettata, cioè la richiesta di esecuzione di un'attività asincrona è partita ma non abbiamo ancora ricevuto un risultato

Una promise può passare dallo stato pendente ad uno solo degli altri due stati. Non sono previsti altri passaggi di stato.

Promise: creazione

```
var promise = new Promise(handler);
```

- ▶ Il **costruttore Promise()** prevede un parametro rappresentato da una funzione (*promise handler*) che ha il compito di gestire la risoluzione o il rigetto della *promise* stessa. In generale, un *promise handler* riceve due funzioni come parametri:

```
var promise = new Promise(function(resolve, reject) {  
    if (condizione) {  
        resolve(valore);  
    } else {  
        reject(motivo);  
    }  
});
```

Promise: esempio

```
function httpGet(url) {  
    return new Promise(function(resolve, reject) {  
        var httpReq = new XMLHttpRequest();  
        httpReq.onreadystatechange = function() {  
            var data;  
            if (httpReq.readyState == 4) {  
                if (httpReq.status == 200) {  
                    data = JSON.parse(httpReq.responseText);  
                    resolve(data);  
                } else {  
                    reject(new Error(httpReq.statusText));  
                }  
            }  
        };  
        httpReq.open("GET", url, true);  
        httpReq.send();  
    });  
}
```

Promise: then()

```
var myPromise = httpGet("/utente/123");

myPromise.then(
    function(utente) {
        console.log("Il server ha restituito " + utente.nome);
    },
    function(error) {
        console.log("Si è verificato " + error.message);
    }
);
```

I metodo **then()** prende due parametri: il primo parametro è la funzione che verrà eseguita nel caso in cui la promise venga risolta; il secondo parametro è la funzione che verrà eseguita se la promise viene rigettata. Queste due funzioni corrispondono in pratica alle due funzioni passate come parametri al *promise handler*. Se ne può passare solo uno ma è sconsigliato

Promise: then()

```
httpGet("/utente/123")
  .then(function(utente) {httpGet("/blog/" + utente.blogId)
  .then(function(blog) {displayPostList(blog.posts);});});
```

- ▶ possiamo riscrivere il codice precedente in maniera più leggibile sfruttando il fatto che il metodo *then()* restituisce sempre una *promise* e possiamo pertanto concatenare il tutto come mostrato di seguito:

```
function getUtente() {return httpGet("/utente/123");}
function getBlog(utente) {
    return httpGet("/blog/" + utente.blogId);}
function displayBlog(blog) {displayPostList(blog.posts);}
```

```
getUtente()
  .then(getBlog)
  .then(displayBlog);
```

Promise: catch()

- ▶ Se è sufficiente avere un solo gestore del rigetto delle promise, come nel nostro esempio, possiamo evitare di passare lo stesso gestore per ciascun metodo `then()` e utilizzare il metodo **catch()**, come mostrato di seguito:

```
getUtente()  
.then(getBlog)  
.then(displayBlog)  
.catch(gestisciErrore);
```

async/await

- ▶ **async** e **await** si basano sul meccanismo delle *Promise* e il loro risultato è compatibile con qualsiasi API che utilizza le *Promise*.
- ▶ In particolare, la parola chiave **async** consente di dichiarare una funzione come asincrona, cioè che contiene un'operazione asincrona, mentre la parola chiave **await** sospende l'esecuzione di una funzione in attesa che la *Promise* associata ad un'attività asincrona venga risolta o rigettata.
- ▶ la parola chiave **await** può essere usata soltanto all'interno di funzioni marcate con **async**.

fetch() e async

```
function getUtente(userId) {  
  fetch("/utente/" + userId)  
    .then(response => {  
      console.log(response);  
    }).catch(error => console.log("Errore!"));  
}
```

utilizza **fetch()** per effettuare una chiamata HTTP (e quindi una operazione asincrona) e visualizzare sulla console i dati di un utente oppure un messaggio d'errore.

fetch() e async

la stessa funzione può essere riscritta utilizzando la parola chiave **async**, che indica che verrà eseguita una operazione asincrona. Il corpo della funzione mantiene la struttura tipica di un normale codice sincrono.

```
async function getUtente(userId) {  
    try {  
        let response = await fetch("/utente/" + userId);  
        console.log(response);  
    } catch (e) {console.log("Errore!");}  
}
```

await davanti all'invocazione di `fetch()` fa in modo che l'esecuzione della funzione `getUtente()` venga sospesa all'avvio dell'operazione asincrona e venga poi automaticamente ripresa quando viene ottenuto un risultato, cioè quando la `Promise` associata a `fetch()` viene risolta o rigettata.

async/await in serie

```
async function getBlogAndPhoto(userId) {  
  try {  
    let utente = await fetch("/utente/" + userId);  
    let blog = await fetch("/blog/" + utente.blogId);  
    let foto = await fetch("/photo/" + utente.albumId);  
    return {utente, blog, foto};  
  } catch (e) {console.log("Errore!");}  
}
```

Le operazioni asincrone non avvengono in parallelo, avendo quindi un potenziale impatto sulle prestazioni dell'applicazione.

async/await in parallelo Promise.all

```
async function getBlogAndPhoto(userId) {  
    try {  
        let utente = await fetch("/utente/" + userId);  
        let result = await Promise.all([  
            fetch("/blog/" + utente.blogId),  
            fetch("/photo/" + utente.albumId)  
        ]);  
        return {utente, blog: result[0], foto: result[1]};  
    } catch (e) { console.log("Errore!")}  
}
```

In questo caso attendiamo il completamento del caricamento dei dati dell'utente, requisito essenziale per recuperare le altre informazioni, e quindi rimaniamo in attesa con `Promise.all()` del caricamento in parallelo dei dati del blog e delle foto.

Attributi di script async defer

- ▶ Sono utili per ottimizzare il caricamento della pagina evitando stop nel rendering
- ▶ Il file HTML viene analizzato fino a quando non viene richiamata una dipendenza (uno `<script>`), a quel punto l'analisi si ferma ed il browser inizia a scaricare il file (se è esterno). Lo `<script>` verrà eseguito subito dopo il download, il rendering dell'HTML continuerà soltanto dopo che lo `<script>` viene eseguito. Per i server lenti e pesanti richiamare `<script>` in questo modo significa che la visualizzazione della pagina web sarà ritardata.

<script async>

- ▶ L'attributo **async** evita di interrompere il rendering dell'HTML durante il download dello <script>, che di fatto avviene in parallelo.
- ▶ Lo <script> viene eseguito subito dopo il download.
- ▶ Il parsing della pagina viene messo in pausa soltanto mentre lo <script> viene eseguito
- ▶ Se non importa quando lo <script> sarà disponibile, usare **async**, per esempio per gli <script> come Google Analytics

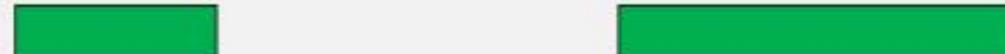
<script defer>

- ▶ L'attributo **defer** evita di interrompere il rendering dell'HTML durante il download dello <script>, che di fatto avviene in parallelo
- ▶ Lo <script> viene eseguito subito dopo il parsing della pagina HTML
- ▶ Il parsing della pagina non viene mai messo in pausa
- ▶ Il DOM sarà già pronto per lo <script>.

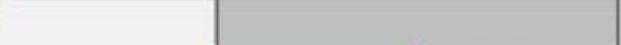
Confronto

<script>

HTML Parsing



HTML Parsing Paused



Script Download

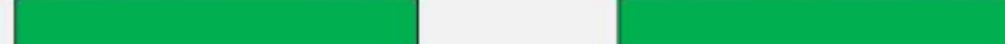


Script Execution



<script async>

HTML Parsing



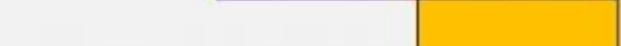
HTML Parsing Paused



Script Download



Script Execution



<script defer>

HTML Parsing



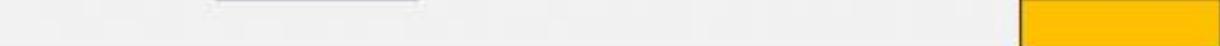
HTML Parsing Paused



Script Download



Script Execution



Quando usarli?

Ecco alcune regole generali da seguire:

- ▶ **Se lo <script> è modulare e non si basa su altri <script>, sarebbe meglio utilizzare async**
- ▶ **Se lo <script> si basa su altri <script> o viene invocato da un altro <script>, sarebbe meglio utilizzare defer**
- ▶ **Se lo <script> è di piccole dimensioni ed è invocato da uno <script> asincrono, sarebbe meglio utilizzare uno <script> in linea senza attributi posto sopra gli <script> asincroni**
- ▶ **Di solito la libreria jQuery non è un buon candidato per l'attributo async perché altri <script> potrebbero dipendere da quella libreria (molto utilizzata). Puoi usare async ma devi assicurarti che gli altri <script> non vengano eseguiti fino al caricamento di jQuery.**



Memorizzazione dei dati



Cookie

- ▶ **Un cookie è un insieme di informazioni salvate sul computer dell'utente, che possono essere riutilizzati in successivi accessi allo stesso sito. L'utente li deve autorizzare.**
- ▶ **La loro dimensione massima è di 4KB e uno stesso dominio non può utilizzarne più di 20**
- ▶ **Non si possono impostare cookie diversi per sessioni contemporanee sullo stesso browser dello stesso sito (due finestre aperte nello stesso dominio)**



WEB storage

- ▶ Le proprietà `localStorage` e `sessionStorage` consentono di salvare coppie chiave/valore in un browser web.
- ▶ L'oggetto **localStorage** archivia i dati senza data di scadenza. I dati non verranno eliminati alla chiusura del browser e saranno disponibili il giorno, la settimana o l'anno successivi. La proprietà `localStorage` è di sola lettura. La proprietà **sessionStorage** memorizza i dati per una sessione di una pagina: i dati vengono persi quando si chiude la scheda del browser.
- ▶ Se apri due browser non condividono i dati!

BOM

WEB storage

- ▶ Il webstorage è disponibile dall'HTML 5, per superare i limiti dei cookie. Possono anch'essi archiviare i dati localmente all'interno del browser dell'utente.
- ▶ È più sicuro (le informazioni non vengono mai trasferite al server) e il limite di archiviazione è molto più grande (almeno 5 MB).
- ▶ Tutte le pagine della stessa origine (dominio+protocollo), possono memorizzare e accedere agli stessi dati: quando siamo connessi ad un certo dominio sono visibili ed accessibili soltanto le variabili relative a quel dominio (con un max di 10MB per dominio)

BOM

WEB storage

- ▶ Permette di memorizzare i dati sotto forma di chiavi-valore (valore è una stringa), in due oggetti:
- **window.localStorage**: dati comuni a tutto il dominio (come i cookie) e non hanno "data di scadenza".
- **window.sessionStorage**: memorizza i dati per una sessione (i dati vengono persi quando la scheda del browser viene chiusa)
- ▶ Prima di utilizzarlo, controllare se è supportato dal browser

```
if (typeof(localStorage|sessionStorage)!== "undefined") {  
    if('localStorage' in window &&  
        window['localStorage'] !== null) {  
            // Code for LocalStorage/sessionStorage.  
    } else {    // Sorry! No Web Storage support..}
```

BOM

WEB storage

- ▶ I metodi per gestire le coppie chiave-valore:
 - **setItem("chiave", "valore")** solo stringhe
 - **getItem("chiave")**
 - **removeItem("chiave")** elimina la chiave
 - **clear()** elimina tutte le chiavi
- ▶ Al posto del set/getItem si può usare la dot notation

```
if (localStorage.getItem("chiave")) //se !undefined  
    localStorage.setItem("chiave",  
        Number(localStorage.getItem("chiave"))+1);
```

```
if (localStorage.chiave) //se !undefined  
    localStorage.chiave = Number(localStorage.chiave)+1;
```

Debugger



Come avviare il debugger in IE

- ▶ Per abilitare il debug di script in Internet Explorer
 - Scegliere Opzioni Internet dal menu Strumenti.
 - Nella finestra di dialogo Opzioni Internet scegliere la scheda Avanzate.
 - Nella categoria Esplorazione deselezionare la casella di controllo Disabilita debug degli script.
 - Fare clic su OK.
 - Chiudere e riavviare Internet Explorer.



Come avviare il debugger in Chrome

- ▶ Per abilitare il debug di script in Chrome
 - Premere la combinazione di tasti **Ctrl+Shift+i**

Come avviare il debugger in Firefox

There are three ways to open the debugger:

- select "Debugger" from the Web Developer submenu in the Firefox Menu (or Tools menu if you display the menu bar or are on Mac OS X)
- press the **Ctrl+Shift+S** (Command Option S on OSX) keyboard shortcut
- press the menu button (), press "Developer", then select "Debugger".



Selezionare quindi nella scheda 421 **SORGENTI** il file da debuggare.

Come avviare il debugger in Firefox



When the debugger is stopped at a breakpoint, you can step through it using four buttons in the toolbar:

- **Play:** run to the next breakpoint
- **Step over:** advance to the next line in the same function.
- **Step in:** advance to the next line in the function, unless on a function call, in which case enter the function being called
- **Step out:** run to the end of the current function

Si può aggiungere il nome della variabile da controllare nella scheda ESPRESSIONI