

# PROGRAMMAZIONE WEB

Andrea De Lorenzo, University of Trieste

# ORARIO LEZIONI

Giorno	Orario	Aula
Lunedì	16:00 - 17:30	Aula F - Edif. G
Martedì	10:00 - 11:30	Aula Idraulica A - Edif. C2
Venerdì	9:15 - 10:45	Aula 4D - H2 bis

<http://delorenzo.inginf.units.it/>

# **MODALITÀ LEZIONI**

- Lezioni in **presenza**, e registrate
- Registrazioni disponibili per circa 6 mesi nel Team del corso
- Accessi al Team del corso tramite codice

**V7MK7KF**

# **COMUNICAZIONI**

Canale Telegram

[https://t.me/+\\_P07aSmDmf0yNDg0](https://t.me/+_P07aSmDmf0yNDg0)

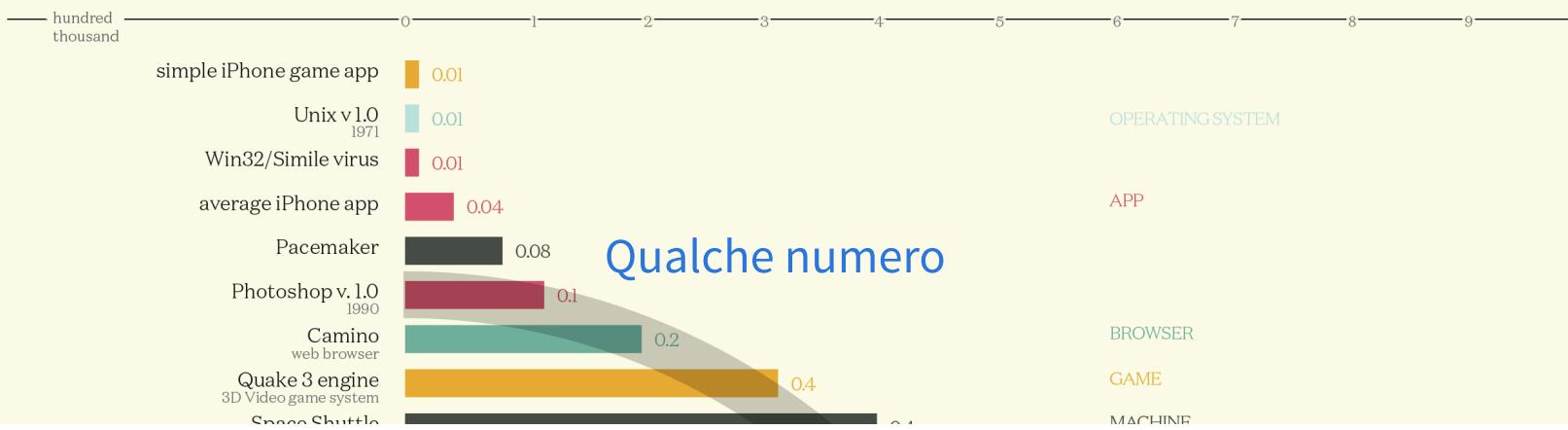
# PREREQUISITI

- HTTP (Reti di calcolatori)
- ~~Java~~
- Basi di Dati

# PROGRAMMARE OGGI

## Codebases

Millions of lines of code



# PROGRAMMARE OGGI

Qualche numero:

1. JavaScript
2. Python
3. Java
4. PHP
5. C#
6. CSS
6. TypeScript
8. C++
9. Ruby
10. C

# **PROGRAMMA DEL CORSO**

## **PROGRAMMAZIONE WEB**

Principi, metodi e tecniche di programmazione del web

**MA COSA SIGNIFICA?**

# PROGRAMMA DEL CORSO

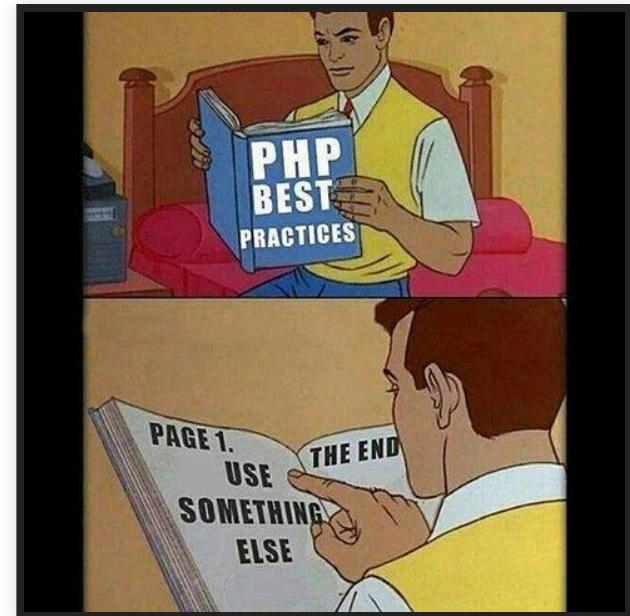
## MA COSA SIGNIFICA?

- ≠ Installare o configurare
  - Google Sites
  - Wordpress
  - Joomla
  - Drupal
  - etc.
- ∞ piattaforme e strumenti

# PROGRAMMA DEL CORSO

Corso pratico, molte tecnologie:

- lato client:
  - HTML, CSS, JavaScript
  - Ajax, VueJS
- lato server:
  - Node.js
  - Web Services



In pratica: come creare un'applicazione  
web

# SOFTWARE

Useremo solo prodotti disponibili online, gratis:

- Atom o [Visual Studio Code](#)
- Netbeans o [IntelliJ IDEA Community](#)

# MATERIALE DIDATTICO

- Queste slides disponibili sul [sito del docente](#) (eventualmente in maniera incrementale)
- Eventuali rimandi di approfondimento nelle slides
  - opzionali
  - obbligatori (→ parte ufficiale del programma)
- Per chi volesse approfondire:
  - "JavaScript Patterns", di Stoyan Stefanov
  - "JavaScript: The Good Parts", di Douglas Crockford
  - "[You Don't Know JS Yet](#)", di Kyle Simpson

# MATERIALE DIDATTICO OPZIONALE

## Libri di Programmazione:

- "The Pragmatic Programmer: From Journeyman to Master", di Andrew Hunt
- "Clean Code - A Handbook of Agile Software Craftsmanship", di Robert C. Martin
- "Design Patterns", di Gamma, Helm, Johnson e Vlissides
- "Refactoring: Improving the Design of Existing Code", di Martin Fowler

# MATERIALE DIDATTICO OPZIONALE

## Libri di Design:

- "Design of everyday things", di Don Norman
- "Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Guidelines", di Jeff Johnson
- "Don't make me think. Un approccio di buon senso all'usabilità web e mobile", di Steve Krug
- [Seminario](#) di Mark Miller sul design delle UI

# MODALITÀ D'ESAME

1. Progetto pratico finale ("tesina"), da consegnare  
*tre giorni prima*
2. Esame orale che inizia con la spiegazione della  
tesina

# MODALITÀ D'ESAME

Alcuni dettagli:

- svolgimento individuale
- specifiche fornite dal docente
- le specifiche cambiano con l'inizio di un nuovo corso

# PROGRAMMAZIONE WEB

## ATTENZIONE!!

Le tecnologie che vedremo

- sono un sottoinsieme microscopico rispetto a quanto disponibile
- **invecchiano rapidamente**

**Scopo finale:** insegnare a pescare, piuttosto che dare il pesce.

# **SEM BRA FACILE...**

# I heard you want to be a web developer



## Here are a few devices to test your site

# PROGRAMMAZIONE WEB

Valutate sempre:

- Quanti utenti?
- Affidabilità richiesta?
- Esiste già qualcosa che risolva il problema?
- Dove risiederà lato server?
- Soluzione monolitica?
- Soluzione modulare?

# HTML

# HTML

## HYPertext Markup Language (HTML)

Linguaggio usato per descrivere il **contenuto e la struttura** dell'informazione di un documento web

Non descrive la presentazione dell'informazione!

**Nota:** *página web = documento web*

# BREVE STORIA

- Preistoria: CERN
  - 1991 → HTML (bozza) **Nota:** *prima descrizione disponibile pubblicamente*
  - 1993 → HTML+ (bozza)
  - 1995 → HTML 2.0
- Ieri: W3C
  - 1/1997 → HTML 3.2
  - 12/1997 → HTML 4.0
  - 2000-2001 → XHTML
- Oggi: W3C + WHATWG
  - 2008-10/2014 → **HTML 5** **Nota:** *stable W3C Recommendation da ottobre 2014*
  - 12/2017 → HTML 5.2

**Nota:** *molto più complicata, [in realtà](#)*

# HTML 5 VS. HTML 4

*Importanti novità:*

- un solo standard, due possibili sintassi: HTML e XHTML (HTML → `<!DOCTYPE html>`)
- nuovi elementi utili per rendere più esplicita la semantica del documento: `<article>` , `<section>` , `<nav>` , `<aside>`
- nuovi elementi utili per integrare contenuti multimediali: `<audio>` , `<video>`

Approfondimento [ufficiale](#) sulle differenze

**Nota:** Per garantire la retro compatibilità, alcune cose possono esseremesse agli interpreti HTML, ma non agli autori

# **HTML**

## **LE BASI**

# SEMPLICISSIMO DOCUMENTO HTML

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Hello HTML</title>
5   </head>
6   <body>
7     <p>
8       Ciao <a href="http://www.units.it">Units</a>!<br/>
9       Stiamo imparando!
10    </p>
11  </body>
12 </html>
```

# ELEMENTI, TAG, ATTRIBUTI

- <title>Hello HTML</title> → **elemento**
  - <title> → **start tag**
  - Hello HTML → **contenuto**
  - </title> → **end tag**
- Lo start tag può contenere **attributi**:  
`<a href="http://www.units.it">UniTs</a>`
  - href="http://www.units.it" → **attributo**
  - href → **nome attributo**
  - http://www.units.it → **valore attributo**

# NOTE

- Alcuni elementi non hanno contenuto:
  - 
  - <br/>
  - c'è solo lo start tag
- Alcuni attributi non hanno un valore:
  - <input type="checkbox" checked/>
- I commenti si inseriscono con <!-- . . . -->:
  - <!--questo non lo vede nessuno-->

# HEADER E BODY

- <head> → *header*, informazioni aggiuntive sul documento
- <body> → contenuto informativo documento

# INFORMAZIONI AGGIUNTIVE

- <title> → titolo del documento
- <meta name="..." content="...>
  - name="author" → autore del documento
  - name="description" → descrizione del documento
  - name="keywords" → parole chiave del documento ( "....,....,...." )

# DAL DOCUMENTO HTML ALLO SCHERMO

Browser: HTML → DOM tree → rappresentazione su schermo

## DOCUMENT OBJECT MODEL (DOM)

Rappresentazione ad albero del documento, **in memoria**

Alternative allo schermo:

- sintesi vocale
- stampante
- ...

# **HTML**

## **REGOLE**

# REGOLE

Diversi livelli di correttezza:

- sintattica (HTML) → chiare, ma "funziona lo stesso"
- stilistica (HTML) → poche e vaghe
- stilistica (rappresentazione) → molte e vaghe
- semantica (HTML) → tante, abbastanza chiare

# REGOLE SINTATTICHE (HTML): ANNIDAMENTO

Gli elementi possono essere annidati ma non sovrapposti

- <p>Esempio <**strong**>corretto</**strong**></p> → corretto
- <p>Esempio <**strong**>corretto</p></**strong**> → errato

## REGOLE SINTATTICHE (HTML): VALORE ATTRIBUTI

Valore degli attributi tra virgolette se contengono spazi

-  → corretto
- <img src=cane-bianco.png alt="Cane bianco"/> → corretto
-  → **errato**

Per non sbagliare, mettiamo sempre le virgolette!

# REGOLE SINTATTICHE (HTML): *NAMED CHARACTER REFERENCES*

Caratteri speciali con un nome:

- `&darr;` → *down arrow* ( $\downarrow$ )
- `&rationals;` → razionali ( $\mathbb{Q}$ )
- tantissimi altri

Regole:

- nel testo non si può usare `&s` se s è il nome di uno dei caratteri
- tutte le *named character references* vanno terminate con il punto e virgola

# REGOLE SINTATTICHE (HTML): TUTTO QUI?

Sono **molte** di più!

Come fare?

- vedere specifica
- usare un validatore

# LA SPECIFICA

La specifica = **IL manuale** = la bibbia:

**HTML5 W3C Recommendation,**

<https://html.spec.whatwg.org/multipage/>

Cosa contiene?

- quali elementi esistono
- per ogni elemento, regole di indentazione
- per ogni elemento, quali attributi
- per ogni attributo, significato e quali valori validi
- ...

Anche come [introduzione all'HTML](#)

# HTML5 W3C RECOMMENDATION: MANUALE DETTAGLIATISSIMO

Esempio: start tag? end tag? /> ?

*The start and end tags of certain normal elements can be omitted, as described later*

Optional tags

*Nota: complicato, meglio mettere sempre l'end tag*

# HTML5 W3C RECOMMENDATION: MANUALE DETTLIATISSIMO

Altro esempio: <em>



The `em` element

# VALIDATORE

## W3C Validator

The screenshot shows the W3C Validator interface displaying five validation errors. Each error is listed with its type (Error or Warning), a brief description, the line number and column where the error occurred, and a snippet of the XML code with the problematic line highlighted.

- Error** Element `<div>` is missing one or more of the following attributes: `[role]`.  
From line 68, column 6; to line 68, column 94  
`<div class="collapse navbar-collapse" id="navbar-service-collapse" aria-expanded="false">`  
Attributes for element `<div>`:  
[Global attributes](#)
- Error** Saw `<!--` within a comment. Probable cause: Nested comment (not allowed).  
At line 104, column 40  
`17 << fine <!--/#header-conte`
- Warning** The document is not mappable to XML 1.0 due to two consecutive hyphens in a comment.  
At line 104, column 40  
`17 << fine <!--/#header-conte`
- Error** Element `<div>` is missing one or more of the following attributes: `[role]`.  
From line 162, column 9; to line 162, column 89  
`><div class="collapse navbar-collapse" id="navbar-collapse" aria-expanded="false">`  
Attributes for element `<div>`:  
[Global attributes](#)
- Error** The `[for]` attribute of the `<label>` element must refer to a non-hidden form control.  
From line 124, column 3; to line 124, column 66  
`<label class="element-invisible" for="edit-custom-search-paths">Custom`

W3C Validator su [www.units.it](http://www.units.it)

**Nota: ... il docente conosce il W3C Validator**

# REGOLE STILISTICHE (HTML)

- Indentazione!
- ...
- Attributi: mettiamo sempre le virgolette!
- Chiudere i tag opzionali

# **REGOLE STILISTICHE (RAPPRESENTAZIONE)**

Stile → *de gustibus*?

Usabilità

# REGOLE STILISTICHE (RAPPRESENTAZIONE)

## USABILITÀ



10 usability crimes

**Nota:** non riguardano solo l'HTML ( $HTML \neq rappresentazione$ )

# **HTML**

## **ELEMENTI HTML**

# ELEMENTI HTML

- `<!-- -->` → defines a comment
- `<!DOCTYPE>` → defines the document type
- `<a>` → defines a hyperlink
- `<abbr>` → defines an abbreviation
- `<address>` → defines an address element
- ...

Elenco completo:

- il [manuale](#)
- w3cschools: [www.w3schools.com](http://www.w3schools.com)

# ELEMENTI HTML: ELENCO

Tag	Description
<!--...-->	Defines a comment
<!DOCTYPE>	Defines the document type
<a>	Defines a hyperlink
<abbr>	Defines an abbreviation or an acronym
<acronym>	Not supported in HTML5. Use <abbr> instead. Defines an acronym
<address>	Defines contact information for the author/owner of a document

# SEZIONI

- article → un pezzo di contenuto indipendente; se annidiati, quello interno è legato all'esterno (es: post di un blog e suoi commenti).
- section → un raggruppamento di contenuti parte di un contenuto più ampio, tipicamente con un titolo
- nav → sezione con i link di navigazione
- aside → un pezzo di contenuto ortogonale al suo contesto (es: note, messaggi twitter, ecc.)

# SEZIONI

- h1 ,..., h6 → titoli (*headings*) (⚠ non usare per sottotitoli o tagline!)
- <header> → contenuto introduttivo (es: titolo e navigazione)
- footer → per informazioni legate alla sezione (es: autore, link aggiuntivi, copyright, ecc.)
- address → informazioni di contatto (⚠ non generici indirizzi)

# SEZIONI: DOMANDE

- Il font di `h1` è più grande di quello di `h2` ?
- Lo spazio verticale tra due `section` è più grande di quello dopo un `article` ?
- Come viene rappresentato il tag `address` ?

# SEZIONI: DOMANDE

Male, non dovreste chiedervelo!

## HYPertext Markup Language (HTML)

Linguaggio usato per descrivere **il contenuto e la struttura** dell'informazione di un documento web

Non la rappresentazione!

# RAGGRUPPARE IL CONTENUTO

```
1 <h2>Le mie memorie</h2>
2 <p>
3   Non me le ricordo più...
4 </p>
```

- p → paragrafo
- hr → cambio di argomento (non necessario tra sezioni)
- pre → testo preformatto
- blockquote → citazione esterna
- main → contenuto principale della pagina, non ripetuto nel sito. ⚡ uno per pagina!

# RAGGRUPPARE IL CONTENUTO

```
<h3>Ingredienti</h3>
<ul>
  <li>pesto alla siciliana</li>
  <li>fusilli</li>
  <li>formaggio grattugiato</li>
</ul>
<h3>Preparazione</h3>
<ol>
  <li>cucinare la pasta</li>
  <li>aggiungere il pesto</li>
  <li>guarnire con formaggio a piacere</li>
</ol>
```

- ul → *unordered list*
- ol → *ordered list*
- li → *list item*

⚠ non vanno inseriti nei paragrafi!

# A CAPO

- CR+LF ≠ nuova linea
- br → *line break*, fa parte del contenuto

*br elements must be used only for line breaks that are actually part of the content, as in poems or addresses.*

“must be used”? → correttezza semantica (HTML)

# SEMANTICA TESTUALE

- em → il contenuto dell'elemento merita enfasi
- strong → il contenuto è importante
- mark → testo evidenziato per riferimento (es: testo cercato)
- s → il contenuto è un pezzo non più accurato o rilevante
- sub e sup → testo in pedice e apice

# SEMANTICA TESTUALE

- i → lettura diversa (es: altra lingua, termine tecnico, ecc.)
- u → testo con annotazione non testuale (es: volutamente errato)

*Authors are encouraged to avoid using the u element where it could be confused for a hyperlink.*

- b → richiamare l'attenzione

*The b element should be used as a last resort when no other element is more appropriate.*

# RAPPRESENTARE CODICE

- code → il contenuto è un pezzo di codice sorgente
- samp → output prodotto dal codice
- var → una variabile
- kbd → input da tastiera... circa
  - inserito nel samp → il tasto è stato premuto dall'utente e mostrato a schermo
  - contiene un tag samp → è stato selezionato un menù
  - contiene altri kbd → combinazione di tasti

# ESEMPIO **kbd**

```
1 <p>Please press  
2   <bdo>  
3     <bdo>Ctrl</bdo> + <bdo>Shift</bdo> + <bdo>R</bdo>  
4   </bdo>  
5   to re-render a page.  
6 </p>
```

Please press Ctrl + Shift + R to re-render a page.

# ESEMPIO samp IN kbd

```
1 <p>To create a new file, choose the menu option  
2   <kbd>  
3     <kbd><samp>File</samp></kbd>=><kbd><samp>New Document</samp></kbd>  
4   </kbd>.  
5 </p>  
6 <p>Don't forget to click the <kbd><samp>OK</samp></kbd> button  
7 to confirm once you've entered the name of the new file.</p>
```

To create a new file, choose the menu option  
File⇒New Document.

Don't forget to click the OK button to confirm once  
you've entered the name of the new file.

# HYPERLINK

Vado all'<**a** href="http://www.units.it">università</**a**>

- **a** → link (*anchor*) ad un altro documento
- **href=""** → ubicazione dell'altro documento
  - stesso sito (**url relativo**) → href="udine.html"
  - altro sito (**url assoluto**) → href="http://www.units.it"
  - stesso documento → href="#persone"
  - indirizzo email → href="mailto:andrea.delorenzo@units.it"

**Nota:** *link* è un'altra cosa

# IMMAGINI

```

```

- src → ubicazione della risorsa immagine
- alt → *fallback content*: “*content that is to be used when the external resource cannot be used*”

# IMMAGINI: ATTRIBUTO alt

## alt è obbligatorio?

*Except where otherwise specified, the `alt` attribute must be specified and its value must not be empty; the value must be an appropriate replacement for the image.*

*One way to think of alternative text is to think about how you would read the page containing the image to someone over the phone, without mentioning that there is an image present.*

*In some cases, the icon is supplemental to a text label conveying the same meaning. In those cases, the `alt` attribute must be present but must be empty.*

*[...] In such cases, the `alt` attribute may be omitted, but one of the following conditions must be met [...]*

"Quasi" obbligatorio metterlo, ma può essere `alt=""`

# TABELLE

```
1 <table>
2   <caption>Orari della Linea 36</caption>
3   <thead>
4     <tr><th>Ora</th><th>Minuto</th></tr>
5   </thead>
6   <tbody>
7     <tr><td>8</td><td>00 15 30 45</td></tr>
8     <tr><td>9</td><td>15 45</td></tr>
9   </tbody>
10 </table>
```

- `caption` → titolo, intestazione
- `thead ; tfoot` → etichette delle colonne; totali, ecc... (*header, footer*)
- `tbody` → corpo (con i valori)
- `tr` → riga (*table row*)
- `td, th` → cella (*table data/header cell*)

# TABELLE

**Non si usano per formattare!**

*Tables must not be used as layout aids. Historically, some Web authors have misused tables in HTML as a way to control their page layout. This usage is non-conforming, because tools attempting to extract tabular data from such documents would obtain very confusing results. In particular, users of accessibility tools like screen readers are likely to find it very difficult to navigate pages with tables used for layout.*

*There are a variety of alternatives to using HTML tables for layout, primarily using CSS positioning and the CSS table model.*

# div E span

*The `div` element has no special meaning at all. It represents its children. It can be used with the `class`, `lang`, and `title` attributes to mark up semantics that is common to a group of consecutive elements.*

```
<div class = " "> ... </div>
```

→ a livello di BLOCCO

x impaginare => grafica

*The `span` element doesn't mean anything on its own, but can be useful when used together with the global attributes, e.g. `class`, `lang`, or `dir`. It represents its children.*

→ all'interno di una linea di testo

x raggrupp e dare un significato particolare

# div: ESEMPIO

```
<article lang="en-US">
  <h1>My use of language and my cats</h1>
  <p>My cat's behavior hasn't changed much since her absence, except
  that she plays her new physique to the neighbors regularly, in an
  attempt to get pets.</p>
  <div lang="en-GB">
    <p>My other cat, coloured black and white, is a sweetie. He followed
    us to the pool today, walking down the pavement with us. Yesterday
    he apparently visited our neighbours. I wonder if he recognises that
    their flat is a mirror image of ours.</p>
    <p>Hm, I just noticed that in the last paragraph I used British
    English. But I'm supposed to write in American English. So I
    shouldn't say "pavement" or "flat" or "colour"...</p>
  </div>
  <p>I should say "sidewalk" and "apartment" and "color"!</p>
</article>
```

# span : ESEMPIO

```
<code class="lang-c"><span class="keyword">for</span> (<span class="ident">j</span> =  
  <span class="ident">i_t3</span> = (<span class="ident">i_t3</span> & 0xffff) | (<sp  
  <span class="ident">i_t6</span> = (((((<span class="ident">i_t3</span> >> 3) ^ <sp  
  <span class="keyword">if</span> (<span class="ident">i_t6</span> == <span class="ide  
    <span class="keyword">break</span>;  
></code>
```

```
for (j = 0; j < 256; j++) {  
    i_t3 = (i_t3 & 0xffff) | (j << 17);  
    i_t6 = (((((i_t3 >> 3) ^ i_t3) >> 1) ^ i_t3) >> 8) ^ i_t3) >> 5) & 0xff;  
    if (i_t6 == i_t1)  
        break;  
}
```

# **HTML**

## **ATTRIBUTI GLOBALI**

# ATTRIBUTO `id`

`id` → identificatore univoco

- unico in tutto il documento
- senza spazi: `<section id="abstract">`

Può essere usato per:

- link interni `<a href="#abstract">`
- ... vedremo

*Identifiers are opaque strings. Particular meanings should not be derived from the value of the `id` attribute.*

# ATTRIBUTO title

title → indicazione descrittiva

- <a href="udine.html" title="Scheda della città dei vicini">Udine</a>
- <abbr title="Cascading Style Sheet">CSS</abbr>

*Advisory information for the element, such as would be appropriate for a tooltip.*

⚠ Attenzione...

- Non considerateli tooltip (es: tablet?)
- Ereditarietà
- new line nell'HTML → new line per davvero!

# ATTRIBUTI lang E translate

lang → lingua principale del contenuto

```
1 <p lang="it">L'inglese è difficile da pronunciare:  
2 ad es. la parola <span lang="en-UK">Wednesdey</span>. </p>  
3 <p lang="de"><span lang="en-UK">Wednesdey</span>  
4 ist ein schwieriges Wort auszusprechen. </p>
```

translate → localizzare o meno questo contenuto

Valori enumerati:

- yes
- no

Es: frammenti di codice, input da tastiera, menù, etc.

# ATTRIBUTO class

Valore: un set di nomi separati da spazi: <span  
class="keyword importante">if</span>

*Authors are encouraged to use values  
that describe the **nature of the  
content**, rather than values that  
describe the desired presentation of  
the content*

**Nota:** correttezza semantica (HTML)

# ATTRIBUTO **data-\***

```
1 <section data-dog-weight="12.8kg" data-dog-speed="75%>
2 <h1>Cane bianco</h1>
3 <p>Il cane bianco è tozzo ma veloce.</p>
4 </section>
```

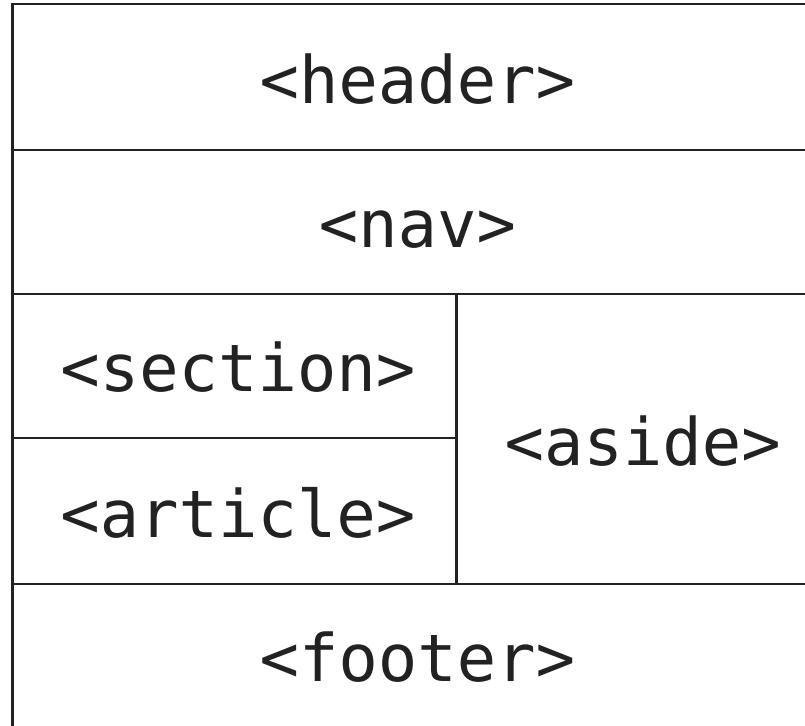
- quello che segue a `data-` è il *nome del custom attribute*
- dati **privati** per la pagina (o l'applicazione)
- CamelCase → **camel-case** (⚠ maiuscole non ammesse)

# **ATTRIBUTI `dir` E `style`**

`dir` → direzione del testo

`style` → aspetto dell'elemento, vedremo poi

# RIASSUNTO: ORGANIZZAZIONE ELEMENTI SEMANTICI



# article IN section O IL CONTRARIO?

The `article` element specifies independent, self-contained content.

The `section` element defines section in a document.

Possiamo usare la definizione per capire come annidare questi elementi? No.

In alcune pagine HTML `section` conterrà `article` e in altre `article` conterrà `section`

# ESERCIZIO HTML-1

Scrivere un documento (solo) HTML sulla **propria città o quartiere**.

Contenuto (*item*):

- i 3 personaggi/animali più famosi
- *oppure*, i 3 edifici/luoghi più belli

Valuteremo la correttezza sintattica, semantica e stilistica del documento HTML

# ESERCIZIO HTML-1

Regole:

- almeno 3 *item* con almeno 3 sezioni
- usare aside , i , em , strong , nav , lang , li
- ogni item con almeno 500 caratteri di testo
- tot. documento con almeno 5000 caratteri di testo
- almeno un link interno

# EDITOR DI TESTO

Scegliete voi, io suggerisco

- Atom
- Visual Studio Code

# EDITOR DI TESTO - ATOM

Pacchetti consigliati:

- [atom-live-server](#) → web-server integrato
- [atom-beautify](#) → formattazione testo
- [autoclose-html](#) → completamento: chiusura tag
- [highlight-selected](#) → evidenzia testo selezionato
- [minimap](#) → preview del codice completo a lato
- [atom-wrap-in-tag](#) → inserimento tag HTML
- [linter](#) → rileva possibili errori
  - [linter-htmlhint](#)
  - [linter-csslint](#)
  - [linter-jshint](#)

# EDITOR DI TESTO - VISUAL STUDIO CODE

Estensioni consigliate:

- [LiveServer](#) → web-server integrato
- [Prettier](#) → formattazione testo
- [ESLint](#) → rileva possibili errori JS
- [HTMLHint](#) → rileva possibili errori HTML
- [Code Runner](#) → esecuzione codice (snippet/file)
- [Vetur](#) → supporto a VueJS
- [Visual Studio IntelliCode](#) → autocompletamento smart
- [Debugger for Chrome](#) → Debug JS visto in Chrome

# RAPPRESENTAZIONE DELL'HTML

# RAPPRESENTARE UN DOCUMENTO HTML

## HYPertext Markup Language (HTML)

Linguaggio usato per descrivere il **contenuto e la struttura** dell'informazione di un documento web

Il browser *visualizza* il documento:  
come lo disegna sullo schermo?

# COME LO DISEGNA?

1. HTML
2. DOM tree  
(rappresentazione in memoria)
3. rappresentazione su schermo

Come?

- HTML → DOM tree: regole di trasformazione implicite
- DOM tree → schermo: ...

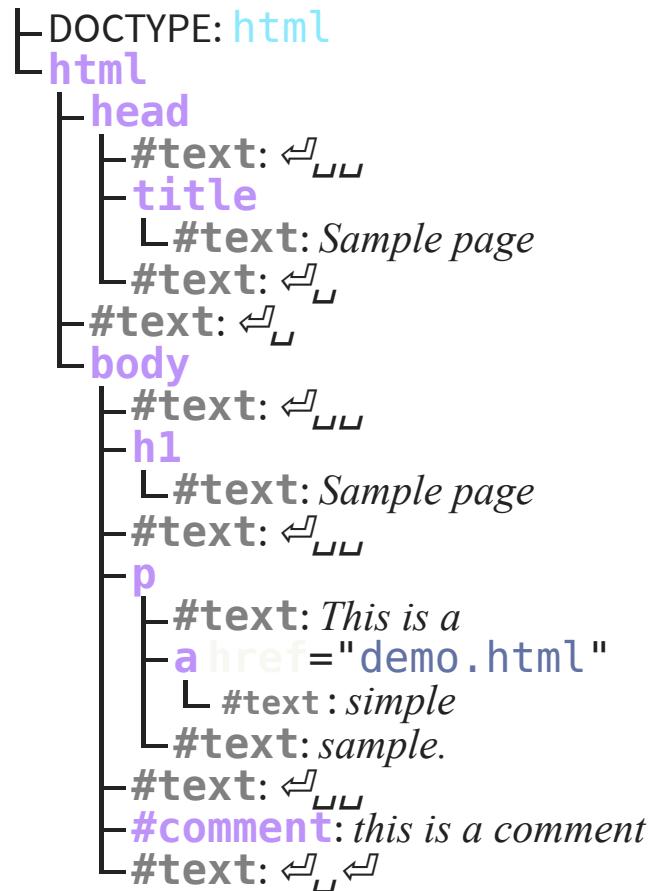
# HTML → DOM TREE

HTML

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Sample page</title>
5   </head>
6   <body>
7     <h1>Sample page</h1>
8     <p>This is a <a href="demo.html">simple</a> sample.</p>
9     <!-- this is a comment -->
10    </body>
11 </html>
```

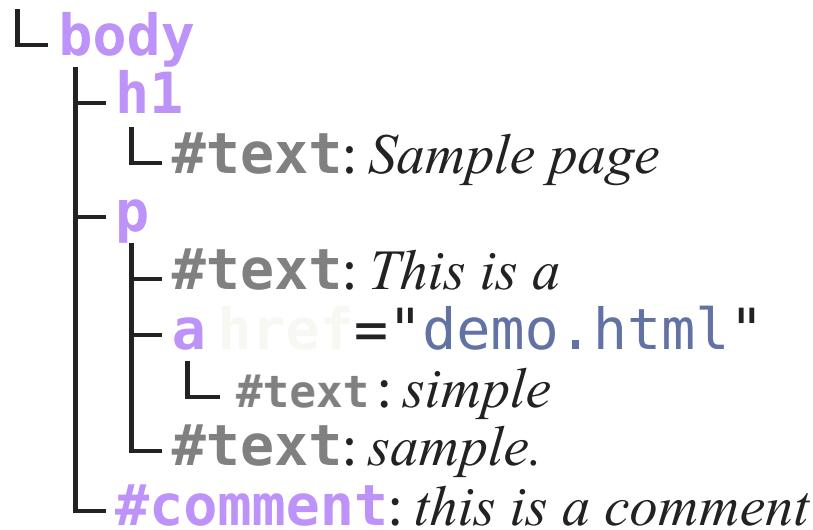
# HTML → DOM TREE

DOM tree:



# DOM TREE: NOMENCLATURA

Consideriamo il **sottoalbero** del nodo body (alcuni #text omessi):



- il nodo body è **parent** del nodo h1
- i nodi p , #comment ,... sono **sibling** del nodo h1
- i nodi h1 , p ,... sono **children** del nodo body
- i nodi a , h1 , p ,... sono **descendant** del nodo body
- i nodi tipo body , p , h1 , a sono **elementi**
- i nodi #text sono **testo** (non hanno children!)

# DOM TREE → SCHERMO: INFORMALMENTE

Per ogni nodo del DOM tree:

- cosa va disegnato?
- dove?
- come?

**Nota:** lettura del documento: "disegnato" → "pronunciato", "dove" → "quando"

Immaginate di essere il browser, o di dover scrivere il codice del programma browser...

# DOM TREE → SCHERMO: FORMALMENTE

**Obiettivo:** DOM tree → **boxes** tree

*Principali* regole della trasformazione:

- ogni elemento  $E$  del DOM tree può generare **zero o più** box
- un box  $B_E$  dell'elemento  $E$  può essere child di un altro box  $B'_E$  di  $E$  o child di un box  $B_{E_p}$  di  $E_p$  parent di  $E$

# COSA VA DISEGNATO?

Un elemento viene disegnato se e solo se sono valide tutte le condizioni:

- il parent è stato disegnato (o non ha parent)
- non è esplicitamente indicato come "da non disegnare"

# DOVE VA DISEGNATO?

Premessa: tipi di box

- **block-level box** → come un paragrafo
- **line box** → come una riga di testo
- **inline-level box** → come una parola

# TIPI DI BOX: CONTENUTO

Cosa possono contenere:

- block-level box
  - block-level boxes
  - line boxes
- line box
  - inline-level boxes
- inline-level box
  - testo
  - inline-level boxes
  - block-level boxes (es: una lista dentro un elemento `strong`)

# DOVE VA DISEGNATO?

A grandissime linee:

- block-level box consecutivi vengono messi **uno sotto l'altro** dentro il parent block-level box

Cerca di riempire tutto lo spazio orizzontale disponibile

- line box consecutivi vengono messi uno sotto l'altro dentro il parent block-level box
- inline-level consecutivi vengono messi **uno accanto all'altro** dentro il parent line-level box

Occupava solo lo spazio strettamente necessario

**Nota:** "consecutivi" → si segue l'ordine del DOM tree

# BLOCK-LEVEL BOXES

Lista non completa:

- p
- table
- nav
- div
- form
- main, article,  
section
- h1, h2, ...
- ...ecc.

# INLINE-LEVEL BOXES

Lista non completa:

- a
- span
- em
- strong
- input , button ,  
textarea
- img
- ...ecc.

# LINE BOXES

E che elementi generano Line Boxes?

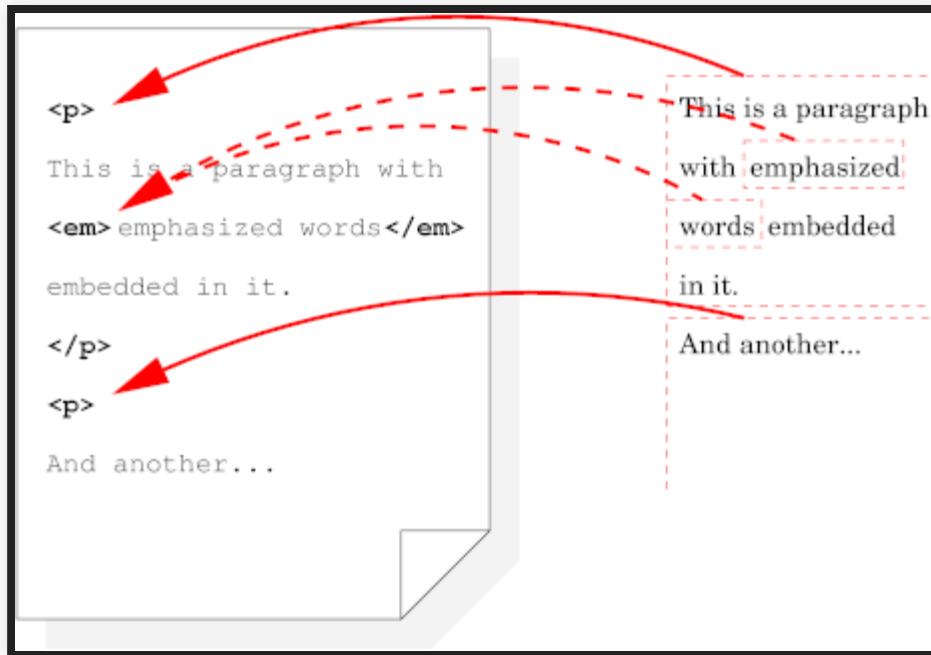
Sono generati in automatico!

*The rectangular area that contains the boxes that form a line is called a line box.*

*When several inline-level boxes cannot fit horizontally within a single line box, they are distributed among two or more vertically-stacked line boxes. Thus, a paragraph is a vertical stack of line boxes.*

*When an inline box exceeds the width of a line box, it is split into several boxes and these boxes are distributed across several line boxes. If an inline box cannot be split (e.g., if the inline box contains a single character, or language specific word breaking rules disallow a break within the inline box, or if the inline box is affected by a white-space value of nowrap or pre), then the inline box overflows the line box.*

# ESEMPIO



*Semplificazione del passaggio HTML → boxes*

# COME VIENE DISEGNATO?

Ci manca qualcosa:

- dato un elemento, che tipo di box genera?
- dato un elemento, quanto margin ha?
- ...
- "come viene disegnato?"

"tipo di box", "margin", ... → **style properties dei box**

# PROPERTIES E CSS

## CASCADING STYLE SHEET (CSS)

Un documento che specifica quali valori dare alle proprietà di quali elementi

- *sheet* → foglio, documento
- *style* → proprietà inerenti la rappresentazione stilistica
- *cascading* → vedremo...

# CSS

## CASCADING STYLE SHEET (CSS)

Un documento che specifica quali valori dare alle proprietà di quali elementi

"all'elemento di tipo `p` deve corrispondere un box di tipo block-level con un margine di 1cm"

# HTML SENZA CSS?

Ma il browser visualizza comunque la mia pagina,  
senza che io gli abbia fornito un CSS!



regole implicite

*The CSS rules given in these subsections are, except where otherwise specified, expected to be used as part of the **user-agent level style sheet** defaults for all documents that contain HTML elements.*

Nella specifica, segue un CSS

# CSS

# CSS: BREVE STORIA

- 1996 → CSS 1
- 1998 → CSS 2
- 2011 → CSS 2.1
- (2005) → **CSS 3** (ancora in sviluppo)

*You don't need to be a programmer or a CS major to understand the CSS specifications. You don't need to be over 18 or have a Bachelor's degree. You just need to be very pedantic, very persistent, and very thorough.*

# CSS: PRINCIPI BASE

- Compatibilità: forward e [backward](#)
- Complementare ai documenti strutturati (HTML e XML)
- Slegato da Vendor/Platform/Device (circa, v. stampanti)
- Manutenzione facile
- Semplice: un solo modo per ottenere l'effetto
- Performance di rete (es: non mando immagini ma testo elaborato)
- Flessibile: posso definirlo in più punti
- Compatibile con altri linguaggi (es: JS)
- Accessibilità

# CSS 3: LA "BIBBIA"?

Meno semplice rispetto all'HTML: la specifica è modulare

*As the popularity of CSS grows, so does interest in making additions to the specification. Rather than attempting to shove **dozens of updates into a single monolithic specification**, it will be much easier and more efficient to be able to update individual pieces of the specification. Modules will enable CSS to be **updated in a more timely and precise fashion**, thus allowing for a more flexible and timely evolution of the specification as a whole.*

*For resource constrained devices, it may be impractical to support all of CSS. For example, an aural browser may be concerned only with aural styles, whereas a visual browser may care nothing for aural styles. In such cases, a user agent may implement a subset of CSS. Subsets of CSS are limited to combining selected CSS modules, and once a module has been chosen, all of its features must be supported.*

# CSS: SPECIFICA MODULARE



Stato di sviluppo dei moduli della specifica

# CSS: DEFINIZIONI

## CASCADING STYLE SHEET (CSS)

Una lista ordinata di **regole** (*rules*)

### REGOLA

- **selector** → a quali elementi si rivolge la regola
- zero o più **declaration** → cosa si applica a quegli elementi

### DECLARATION

- nome **proprietà**
- valore proprietà

**Nota:** *non si applica solo ai documenti HTML*

# SINTASSI DI UN CSS

```
selector {  
    property-name: property-value;  
    property-name: property-value;  
    ...  
}  
  
selector { /* commento */  
    property-name: property-value;  
    property-name: property-value;  
    ...  
}  
/* commento */
```

La specifica dice:

- qual è la **sintassi** del selector
- quali sono i nomi delle proprietà
- per ogni proprietà, quali sono i possibili valori (**sintassi**)

# @ RULE

Vi sono delle regole particolari, che iniziano con @ :

- @charset
- @import
- @namespace
- @media
- @viewport
- ...

Ma ne parleremo poi...

# SINTASSI DI UN CSS

**Attenzione:** se la sintassi di una regola non è corretta,  
il browser **deve** ignorare tutta la regola!

**Nota:** "*deve ignorare*" = "*la specifica raccomanda che il browser ignori*" → *il browser potrebbe non ignorare...*

**Nota:** *diverso da quanto accade per l'HTML*

# SINTASSI DI UN CSS

Tre livelli di correttezza:

- sintattica (CSS) → altrimenti viene ignorato
- stilistica (CSS) → indentazione, *please!*
- stilistica (rappresentazione: HTML+CSS)

**Nota:** esiste il *validatore per i CSS*

# UBICAZIONE DI UN CSS

Chi stabilisce lo stile?

- Author
- User
- User agent

# UBICAZIONE DI UN CSS: AUTHOR

Dove specifico che voglio usare certe regole per un certo documento HTML *d*? Tre metodi:

- come documento di testo **esterno** a *d*:

```
<link rel="stylesheet" type="text/css" href="...">>
```

- **embedded** in *d*, come contenuto di un elemento **style**:

```
<style>...</style>
```

- dentro (**inline**) *d*: con la/le **sola/e declaration** come valore di un attributo **style** dell'elemento a cui voglio applicare la regola (il selector non serve!):

```
<p style="...">>
```

# UBICAZIONE DI UN CSS: @import

La regola `@import` permette di importare in un codice CSS altri fogli di stile

- Deve essere la **prima riga** del file CSS
  - prima ci può essere solo `@charset`
- Lo *user-agent* incorporerà il file come se fosse scritto in quel punto
- È possibile inserire condizioni (es: tipo di dispositivo, risoluzione, ecc.)

Es: importazione di [Google Web Font](#)

```
<link href="https://fonts.googleapis.com/css?family=Roboto" rel="stylesheet" >  
  
font-family: 'Roboto', sans-serif;
```

# CSS

## CSS CASCADING

Fonte

# TANTI METODI...

Si possono usare tutti i metodi assieme → *vengono applicate tutte!*

- Più documenti esterni (più <link ...> )  
**Nota:** *link serve anche per altre cose*
- E se più regole sono in **conflitto**? (regole che specificano valori diversi per la stessa proprietà di uno o più elementi)
- E se qualche proprietà non è stata definita?

# EREDITARIETÀ (*INHERITANCE*)

Se una proprietà  $P$  non è definita per un elemento  $E$ ,  
l'elemento **eredita il valore** di  $P$  del suo parent (parent di  $E$ )

- Per *multi* elementi e *molte* proprietà, si usa il  
valore specificato nello user-agent level style  
sheet

Inheritance → **cascading** style sheet

# EREDITARIETÀ: VALORI DELLE PROPRIETÀ

Come calcolo il valore di ogni proprietà?

Sei valori per ogni proprietà...

1. **Dichiarato:** esplicitamente nel codice
2. **Ereditato (*cascaded*):** dopo aver applicato l'ereditarietà
3. **Specificato:** prende il default se non dichiarato e non ereditato
4. **Calcolato:** "2 volte l'altezza del font"
5. **Usato:** dopo l'elaborazione del parent (es: "10% larghezza")
6. **Finale:** richieste impossibili per l>User Agent → 10.9 diventa 11

# EREDITARIETÀ: VALORI DELLE PROPRIETÀ

Esempi:

Dichiarato	Ereditato	Specificato	Calcolato	Usato	Finale
text-align: left	left	left	left	left	left
width: (none)	(none)	auto	auto	120px	120px
font-size: 1.2em	1.2em	1.2em	14.1px	14.1px	14px
width: 80%	80%	80%	80%	354.2px	354px

Nota: *em* è la dimensione del font →  $2em = 2$  volte la dimensione del font.

# EREDITARIETÀ: CONFLITTO

Se più regole sono in **conflitto**?

1. metodo (origine)
  1. inline style sheet ( `style="..."` , più importante)
  2. embedded style sheet ( `<style>` )
  3. documento esterno ( `<link>` )
  4. user level stle sheet
  5. user-agent level style sheet (meno importante)
2. specificità (vedremo)
3. ordine di definizione (vale l'ultima)

# EREDITARIETÀ: !important

*CSS attempts to create a balance of power between author and user style sheets. By default, rules in an author's style sheet override those in a user's style sheet.*

!important cambia la priorità:

1. User-agent
2. User
3. Author

Esempio:

```
{ background-color: lightgrey !important; }
```

# QUALCHE CASO D'USO LEGITTIMO PER !important

- State usando un Content Management System (CMS: Wordpress, Drupal, ...) e non potete modificare lo stile
- Volete dare uno stile uniforme ai bottoni del vostro sito, quindi definiti una regola generica. Poi aggiungete una regola con specificità maggiore che la sovrascrive (vedremo). In questo caso !important risolve il problema

# QUALCHE CASO D'USO LEGITTIMO PER !important

```
1 .button {  
2   background-color: #8c8c8c;  
3   color: white;  
4   padding: 5px;  
5   border: 1px solid black;  
6 }  
7  
8 #myDiv a {  
9   color: red;  
10  background-color: yellow;  
11 }
```

```
1 .button {  
2   background-color: #8c8c8c !important;  
3   color: white !important;  
4   padding: 5px !important;  
5   border: 1px solid black !important;  
6 }  
7  
8 #myDiv a {  
9   color: red;  
10  background-color: yellow;  
11 }
```

# VALORI PREDEFINITI

Alcune parole chiave permettono di forzare alcuni comportamenti:

- initial → valore predefinito
- inherit → ottenuto dagli ancestor
- ...ve ne sono altre ( unset , revert )

# CSS

## CSS SELECTORS

Fonte

# **SELECTOR**

## **CSS SELECTOR**

Espressione che applicata ad un elemento restituisce un booleano

$$f_{\text{selector}}(E) \in \{\text{true}, \text{false}\}$$

La specifica descrive tutte le possibili funzioni f

# SELECTORS PRINCIPALI

- $*$  → qualsiasi elemento ( $f_*(E) = \text{true}, \forall E$ )
- $e$  → gli elementi di tipo  $\langle e \rangle$
- $f \ e$  → gli elementi di tipo  $\langle e \rangle$  descendant di elementi di tipo  $\langle f \rangle$
- $e.\text{className}$  → gli elementi di tipo  $\langle e \rangle$  di classe "className"
- $e\#\text{idName}$  → gli elementi di tipo  $\langle e \rangle$  che hanno il valore dell'attributo id uguale a "idName"

**Nota:** *descendant ≠ child*

**Nota:** `` è selezionato sia da  
`img.photo` che da `img.old`

# COMBINAZIONE DI SELECTORS

I selectors si possono "combinare":

- e.c1 f → un <f> descendant di un <e> con classe "c1"
- e f g → un <g> descendant di un <f> descendant di un <e>
- e \* → tutti i descendant di un <e>

Il selettore universale \* si può omettere in **certi casi**:

- \*.c1 = .c1 → un elemento qualsiasi con classe "c1"
- \*#id1 = #id1 → un elemento qualsiasi con id "id1"

# SELECTORS SULLA PARENTELA

- $e > f \rightarrow$  un  $<f>$  child di un  $<e>$
- $e + f \rightarrow$  un  $<f>$  inserito *subito* dopo un suo sibling  $<e>$
- $e \sim f \rightarrow$  un  $<f>$  inserito dopo un suo sibling  $<e>$

# SELECTORS SUGLI ATTRIBUTI

- `e[a]` → un `<e>` con un attributo `a`
- `e[a="val"]` → un `<e>` con un attributo `a` uguale "val"
- `e[a~="val"]` → un `<e>` con `a="v1 v2 v3 ... val ..."` (valori separati da spazi)
- `e[a^="val"]` → un `<e>` con `a="valtuttoilresto"`
- `e[a$="val"]` → un `<e>` con `a="tuttoilrestoval"`
- `e[a*="val"]` → un `<e>` con `a="qualcosa val qualcosa"`
- `e[a|="val"]` → un `<e>` con `a="val-v2-v3"` (valori separati da "-")

`f [lang|="en"] ( <p lang="en-US"> ) = f [lang|="en"] ( <p lang="en-UK"> ) =`

# SELECTORS DI PSEUDOCLASSI STRUTTURALI

Pseudoclassi: classi non esplicitamente definite, ma corrispondenti a elementi presenti nell'HTML

- `e:nth-child(n)` , `e:nth-last-child(n)` → un `<e>` n-simo (n-esimo dal basso) figlio del suo parent
- `e:nth-of-type(n)` , `e:nth-last-of-type(n)` → un `<e>` n-simo (n-ultimo) figlio del suo parent tra i figli di tipo `<e>`
- `e:first-child` , `e:last-child` , `e:first-of-type` , `e:last-of-type`
- `e:only-child` , `e:only-of-type` → un `<e>` figlio unico (senza sibling di tipo e)
- `e:root` → un `<e>` root del documento
- `e:empty` → un `<e>` senza figli, nemmeno testo

# **nth-child : DETTAGLIO**

e:nth-child(n) : n può essere un'expressione un po'  
più complessa:

- e:nth-child(3) → il terzo figlio
- e:nth-child(even) = e:nth-child(2n) → figli pari
- e:nth-child(odd) = e:nth-child(2n+1) → figli dispari
- e:nth-child(5n+2) → il 2°, 7°, 12°, ... figlio

# SELECTORS DI ALTRE PSEUDOCLASSI

- `e:visited` , `e:link` → un `<e>` che è un'ancora ad un documento visitato (non visitato)
- `e:active` , `e:hover` , `e:focus` → un `<e>` in **certi stati** relativi all'interazione con l'utente
- `e:enabled` , `e:disabled` , `e:checked` → un `<e>` che fa parte della UI che trova in certi stati
- `e:lang(en)` → un `<e>` con contenuto in lingua inglese
- `e:target` → un `<e>` che è stato selezionato con un anchor (URL#anchor)

`http://units.it/bandi.html#scaduti` → `<section id="scaduti">` è `:target`

# SELECTORS DI PSEUDOELEMENTI

Pseudoelementi: elementi non esplicitamente definiti,  
quindi generati

- `e::first-line` → un **elemento fittizio** contenente la prima linea di testo di un `<e>`
- `e::first-letter` → un elemento fittizio contenente la prima lettera di un `<e>`
- `e::before` , `e::after` → un elemento fittizio che precede (succede) a un `<e>` suo sibling
  - molto utile se si usa la property `content`

# PSEUDOELEMENTI: ESEMPIO

```
<p>Nel mezzo di cammin di nostra vita</p>
```



```
<p><span>::first-letter>N</span>el mezzo di cammin di nostra vita</p>
```

# SELECTOR DI NEGAZIONE

- `e:not(selector)` → un `<e>` a cui **non** si applica il selector (solo selectors non combinati, `div > div` non va bene).

`p:not(:first-child)` → tutti i `p` che non siano i primi figli del loro parent

# SELECTOR DI PARENTELA

## :has()

The :has() CSS pseudo-class represents an element if any of the selectors passed as parameters match at least one element.

- e:has(selector) → un <e> che sia parent di un elemento a cui si applica il selector
- si può concatenare: e:has(f):has(g) oppure e:has(f, g)

**Nota:** Il supporto è limitato a *alcuni browser*

# SELECTOR E SPECIFICITÀ

Ricordate? È uno degli elementi per risolvere l'ereditarietà.

Si genera un "numero" composto da tre cifre, ottenute contando nel selettore:

- attributi ID
- altri attributi o pseudo-classi
- nomi di elementi
- si ignorano gli pseudo-elementi

Più grande il numero, più grande la specificità.

Ottimo esempio basato su Star Wars.

# SELECTOR: ESEMPIO

```
1 tr:nth-child(even) {  
2   background-color: #8be9fd; /* vedremo... */  
3 }
```

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20

# SELECTOR: ESEMPIO

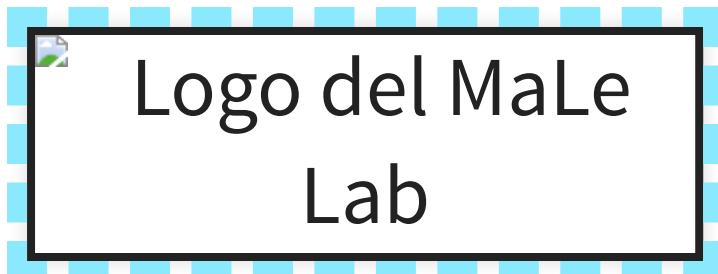
```
1 p::first-letter {  
2   font-size: 200%; /* vedremo ... */  
3   color: #8be9fd;  /* pure questo ... */  
4 }
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc tristique velit eget neque ornare, vitae luctus magna mollis. Donec arcu ligula, tristique et porttitor nec, feugiat et tortor. Suspendisse vel nisi a mauris consectetur hendrerit. Fusce congue leo est, et suscipit massa venenatis vitae. Sed nec molestie nibh. Sed purus tortor, molestie sed justo non, iaculis ultricies orci. Nullam quis quam justo. Nunc finibus aliquet tincidunt. Nunc mattis metus at arcu tristique semper. Vivamus iaculis lacus porttitor, pretium leo tincidunt, euismod nisi. Morbi sodales pharetra ante in congue. Aenean sed erat dui. Aenean eget elementum metus.

# SELECTOR: ESEMPIO

```
1 figure:has(figcaption) img {  
2     outline: 10px dashed #8be9fd;  
3 }
```

CON FIGCAPTION



Logo del MaLe Lab

SENZA FIGCAPTION



# CSS

## CSS PROPERTIES

Ne vedremo solo alcune...

Una lista completa

# COLORI

- color → colore del testo
- background-color → colore dello sfondo
- border-color → colore del bordo

## Valori:

- color: red; → per nome (eventualmente "esotico")
- color: #ff0000; → rgb esadecimale
- color: rgb(255,0,0) , color: rgb(100%,0%,0%) → rgb decimal
- color: rgba(255,0,0,0.33) , color: rgba(100%,0%,0%,0.33) -  
rgb decimale con canale alpha
- color: transparent; → trasparente

# IMMAGINE DI SFONDO

- background-color → colore dello sfondo
- background-image → immagine di sfondo
- background-repeat → come ripetere l'immagine
- background-attachment → come muovere l'immagine in caso di scrolling
- background-position → dove mettere l'immagine
- oppure background → **tutto insieme** (c.d. *Shorthand*)

Esempio:

```
1 body {  
2   background-image: url('img_tree.png');  
3   background-repeat: no-repeat;  
4   background-position: right top;  
5 }
```

# FONT

- font-family → quale font
- font-size → dimensione
- font-weight → peso (più o meno "grosso")
- font-variant → variante (normale, maiuscololetto)
- font-style → stile (normale, italico, obliquo)
- oppure font → **tutto insieme**

Esempio:

```
1 h1 {  
2   font-family: Helvetica, Verdana, sans-serif;  
3   font-weight: bold;  
4   font-size: x-large;  
5 }
```

font-family: **Helvetica, Verdana, sans-serif** → viene scelto il primo disponibile

# font-family E @font-face

L'autore vuole usare un font specifico, ma non si sa se l'utente ce l'ha:  
font-family: "Font Fighetto di Design", sans-serif;



@font-face definisce un font

```
1 @font-face {  
2   font-family: 'Ubuntu';  
3   src: local('Ubuntu'),  
4       local('Ubuntu-Regular'),  
5       url('http://themes.googleusercontent.com/font?kit=2Q-  
6   }  
7  
8 h1 { font: "Ubuntu" 10px; }
```

**Nota:** @font-face è una *at-rule*

**Nota:** *Google Web Font*

# font-size

font-size: absolute-size | relative-size | length | percentage | inherit

Esempi:

```
1 font-size: medium;
2 font-size: large;
3 font-size: 115%;
4 font-size: larger;
5 font-size: 10px;
6 font-size: 0.75cm;
```

# UNITÀ DI MISURA NEI CSS

Alcune proprietà accettano un valore di tipo [length](#)

Unità di misura **assolute**:

*in* → inches; 1 inch is equal to 2.54 centimeters

*cm* → centimeters

*mm* → millimeters

*pt* → points; 1pt is equal to 1/72 inch

*pc* → picas; 1 pica is equal to 12 points

Come fa il browser a sapere [quanti pixel fanno un cm?](#)

# UNITÀ DI MISURA NEI CSS

## Unità di misura relative:

*em → the font size of the element (or, to the parent element's font size if set on the 'font-size' property)*

*ex → the x-height of the element's font*

*px → viewing device*

*gd → the grid defined by 'layout-grid' described in the CSS3 Text module*

*rem → the font size of the root element*

*vw → the viewport's width*

*vh → the viewport's height*

*vm → the viewport's height or width, whichever is smaller of the two*

*ch → The width of the "0" glyph found in the font for the font size used to render*

# TESTO

- line-height
- letter-spacing
- text-align
- text-decoration
- text-indent
- text-transform
- vertical-align

Esempio (di più):

```
1 .draft span.note {  
2   text-transform: uppercase;  
3 }  
4 .draft span.old-version {  
5   text-decoration: overline;  
6 }
```

# CONTATORI

Si comportano come variabili

- definiti con la property counter-reset: nomeContatore;
- incrementati usando la property counter-increment: nomeContatore;
- Usabili nelle property content

Esempio:

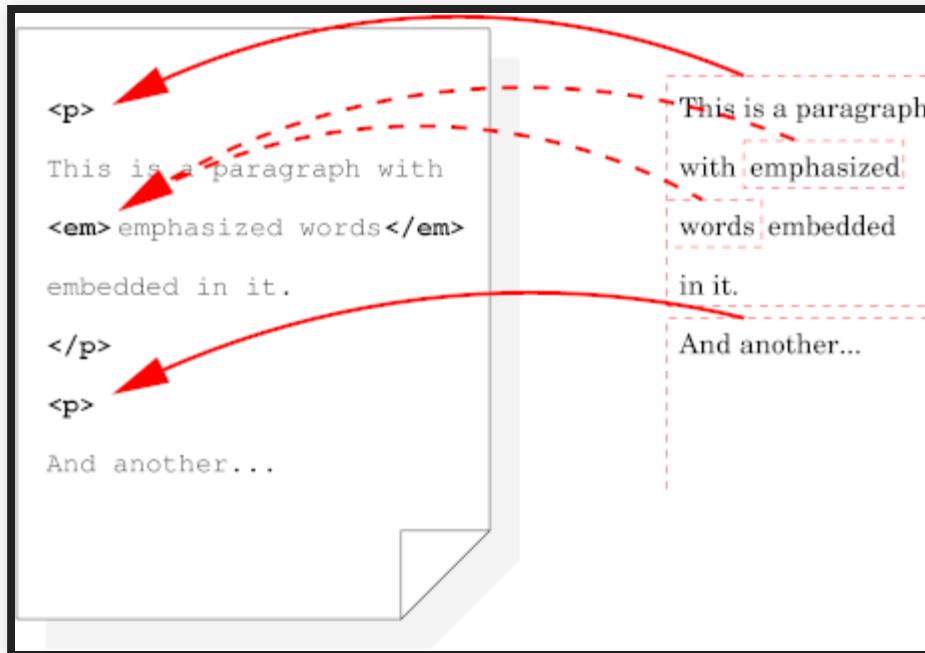
```
1 body {  
2   counter-reset: contaSezione;  
3 }  
4 h2::before {  
5   counter-increment: contaSezione;  
6   content: "Sezione " counter(contaSezione) ":";  
7 }
```

# CSS

## CSS LAYOUT

Fonte

# RICORDATE?



*Semplificazione del passaggio HTML → boxes*

# DISPLAY

Definisce come verrà realizzato il box

- **inner display type** → che contesto di formattazione crea, e come i suoi *descendant* saranno posizionati
- **outer display type** → come il box partecipa alla formattazione del suo *parent*

# OUTER DISPLAY TYPE

Ci sono tre valori possibili

- **block**
- **inline**
- **run-in** → fa il merge con il blocco che lo segue, inserendosi al suo inizio (Es: definizione nel dizionario).

# INNER DISPLAY TYPE

Varie opzioni:

- **flow** → dipende dall'*Outer Display Type*:
  - inline o run-in: genera una inline box
  - block: genera un blocco
- **flow-root** → crea sempre un contenitore di tipo Block
- ... e molti altri (table, flex, grid, ruby)

# PROPRIETÀ `display`

Riassume in un unico valore come impostare Inner/Outer Display Type

In sintesi, determina il **tipo di box** che verrà generato

- `display: inline` → inline flow → inline-level box
- `display: block` → block flow → block-level box
- `display: none` → **non** genera box
- `display: inline-block` → inline flow-root → vedremo poi
- `display: flex` → block flex → vedremo poi
- **molti altri**

`display` influisce **pesantemente** sulla rappresentazione di un documento!

# NASCONDERE UN BOX

- visibility: visible → disegna
- visibility: hidden → non  
disegnare

**Attenzione:**

- visibility: hidden → il box occupa lo spazio previsto ma non si vede
- display: none → il box non c'è

# DIMENSIONI

Si usano `width` e `height`:

- non si applicano agli inline-level box (in prima approssimazione)
- ... e se le dimensioni impostate superano le dimensioni della finestra?
  - appaiono le barre di scorrimento → brutto!
  - usiamo `max-width`

# DIMENSIONI

Basta sapere width e height per calcolare lo spazio occupato?

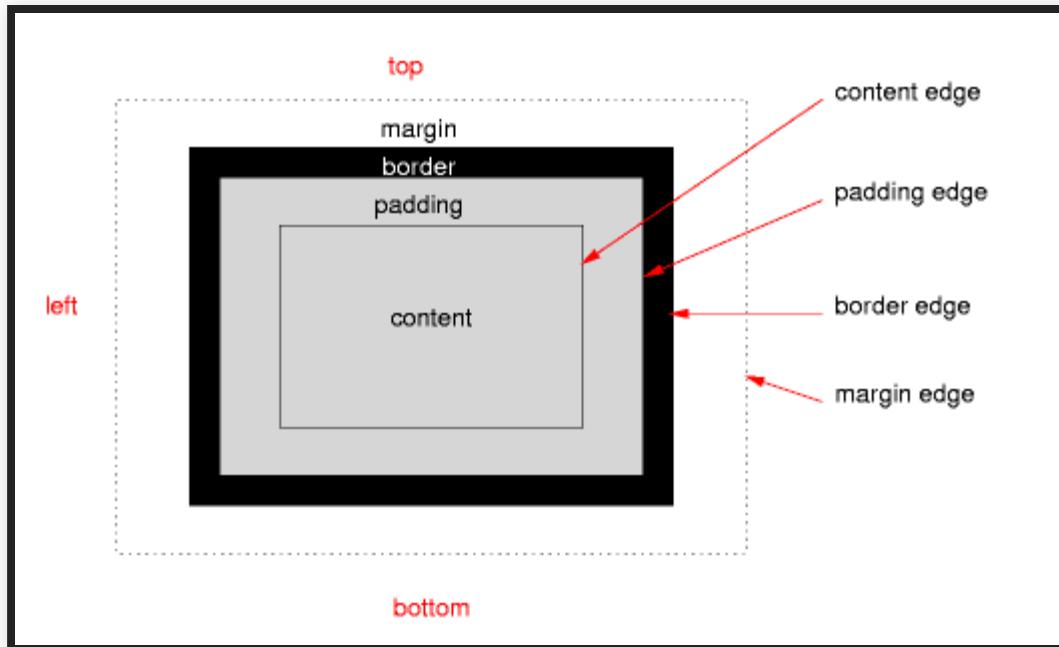
NO!

Fanno riferimento al contenuto, occorre conoscere margin e padding

# DIMENSIONI

Ma posso imbrogliare:

box-sizing: border-box; → padding e bordi rientrano nella misura



*Hint: valutate se inserirlo in un blocco \* { }*

# POSIZIONI

Come posso spostare il box?

Proprietà `position`:

- `static` → default, segue il normale flusso; non è affetto da `top/bottom/right/left` e formalmente non è posizionato
- `relative` → vanno usate proprietà extra (`top, right, bottom o left`)
  - *relative to itself*
  - gli altri contenuti non si infileranno negli spazi lasciati liberi
  - Es: `position: relative; top: -20px; left: 20px;` fa sovrapporre il secondo blocco al primo



# POSIZIONI

Come posso spostare il box?

Proprietà position :

- **fixed** → posizionato in base alla finestra del browser
  - gestito con top, right, bottom o left
  - lo spazio liberato viene riciclato
  - gestione traballante nei dispositivi mobile
- **absolute** → si comporta come fixed ma riferito al più vicino elemento di livello superiore **posizionato** quindi non static
  - Altrimenti, fa riferimento a <HTML>

# POSIZIONI

Esempio:

```
nav {  
    position: absolute;  
    left: 0px;  
    width: 200px;  
}  
  
section {  
    /* position is static by default */  
    margin-left: 200px;  
}  
  
footer {  
    position: fixed;  
    bottom: 0;  
    left: 0;  
    height: 70px;  
    background-color: white;  
    width: 100%;  
}
```

# FLOATING BOX

Un floating box viene spostato nella linea a destra o a sinistra finché non sbatte contro il bordo del suo parent box

Il resto del contenuto scorre a fianco; i line box vengono accorciati

- float: left , float: right → è float
- float: none → non è float (*default*)

Es: style="float: left; width: 200px; height: 100px; margin: 1em;"

Galleggio!

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus imperdiet, nulla et dictum interdum, nisi lorem egestas odio, vitae scelerisque enim ligula venenatis dolor. Maecenas nisl est, ultrices nec congue eget, auctor vitae massa. Fusce luctus vestibulum augue ut aliquet. Mauris ante ligula, facilisis sed ornare eu, lobortis in odio. Praesent convallis urna a lacus interdum ut hendrerit risus congue. Nunc sagittis dictum nisi, sed ullamcorper ipsum dignissim ac. In at libero sed nunc venenatis imperdiet sed ornare turpis. Donec vitae

dui eget tellus gravida venenatis. Integer fringilla congue eros non fermentum. Sed dapibus pulvinar nibh tempor porta. Cras ac leo purus. Mauris quis diam velit.

# FLOATING BOX

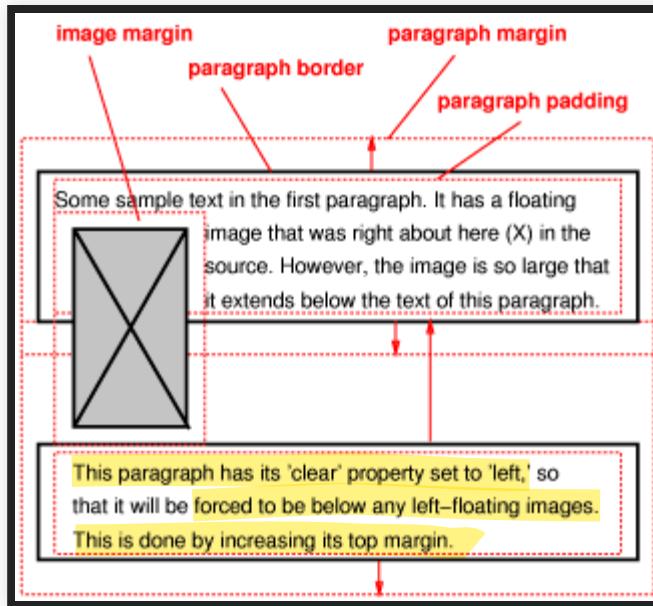
Un floating box può essere più alto del containing box; anche le righe del successivo block-level box vengono accorciate se necessario...  
come evitarlo?

- Soluzione 1: lo impedisco con `clear`
  - `clear: left` , `clear: right` , `clear: both` → non affianca a floating box a sinistra e/o destra
  - `clear: none` → permetti di affiancare
- Soluzione 2: estendo il box con un `clearfix`

```
1 .clearfix {  
2   overflow: auto;  
3   zoom: 1; /* per IE6 */  
4 }
```

# float E clear: ESEMPIO

```
1 p {clear: left;}  
2 img {float: left;}
```



Esempio di float e clear

In che punto del documento HTML è inserita l'immagine?

# INLINE-BLOCK BOX

Voglio creare una griglia di box che riempia tutta la larghezza del browser:  
posso farlo con

- `float: left` → scomodo, devo applicare `clear` al blocco successivo
- `display: inline-block` (qualche problema con IE6 e IE7)

Sono un  
blocco in-line!

Ed io non necessito di alcuna pulizia!

# INCOLONNARE

column mi permette di incolonnare il testo

```
1 .three-column {  
2   padding: 1em;  
3   column-count: 3;  
4   column-gap: 1em;  
5 }
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit.  
 Phasellus imperdiet, nulla et dictum interdum, nisi  
 lorem egestas odio, vitae scelerisque enim ligula  
 venenatis dolor. Maecenas nisl est, ultrices nec congue  
 eget, auctor vitae massa. Fusce luctus vestibulum

augue ut aliquet. Mauris ante ligula, facilisis sed ornare  
 eu, lobortis in odio. Praesent convallis urna a lacus  
 interdum ut hendrerit risus congue. Nunc sagittis  
 dictum nisi, sed ullamcorper ipsum dignissim ac. In at  
 libero sed nunc venenatis imperdiet sed ornare turpis.

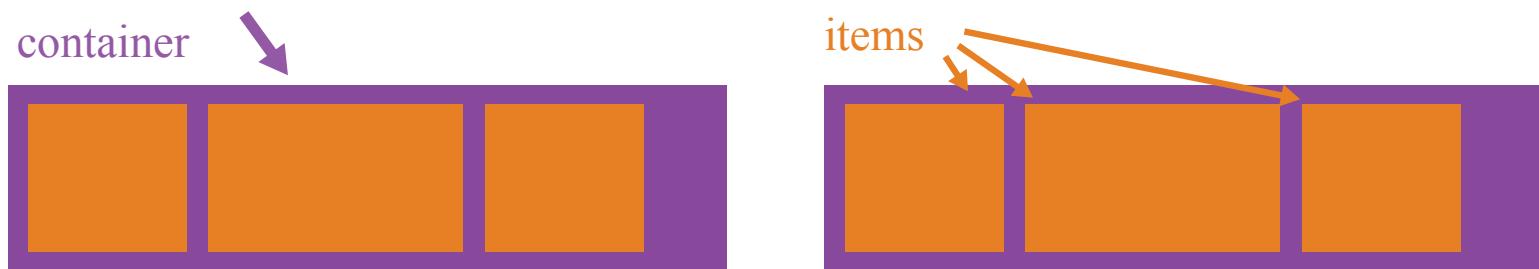
Donec vitae dui eget tellus gravida venenatis. Integer  
 fringilla congue eros non fermentum. Sed dapibus  
 pulvinar nibh tempor porta. Cras ac leo purus. Mauris  
 quis diam velit.

# FLEXBOX

`display: flex;` arriva con CSS3 → non tutti i browser si comportano bene

Ha svariate proprietà, raggruppabili per applicabilità:

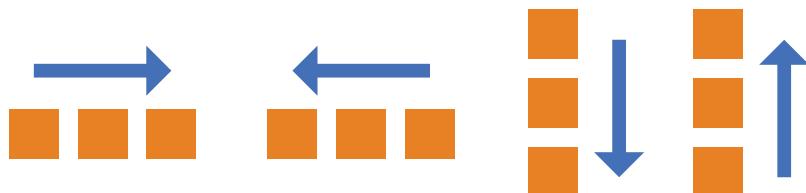
- parent (container)
- children (flex items)



Vi lascio una [guida visuale](#)

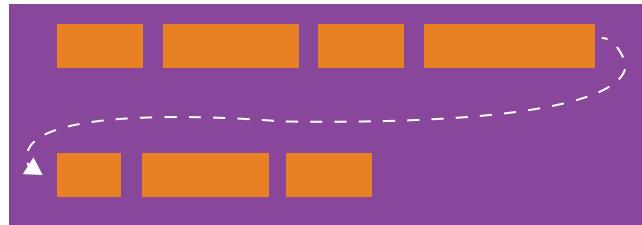
# FLEXBOX - CONTAINER

- `flex-direction : row | row-reverse | column | column-reverse`



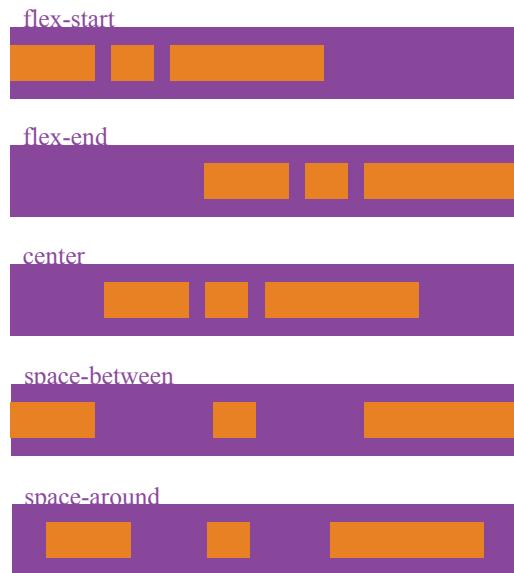
# FLEXBOX - CONTAINER

- `flex-direction : row | row-reverse | column | column-reverse`
- `flex-wrap : nowrap (default) | wrap | wrap-reverse (bottom-top)`



# FLEXBOX - CONTAINER

- flex-direction : row | row-reverse | column | column-reverse
- flex-wrap : nowrap (default) | wrap | wrap-reverse (bottom-top)
- justify-content : flex-start | flex-end | center | space-between | space-around | space-evenly



# FLEXBOX - CONTAINER

- `flex-direction` : row | row-reverse | column | column-reverse
- `flex-wrap` : nowrap (default) | wrap | wrap-reverse (bottom-top)
- `justify-content` : flex-start | flex-end | center | space-between | space-around | space-evenly
- `align-items` : flex-start | flex-end | center | baseline | stretch (allinea una riga del flex)
- `align-content` : flex-start | flex-end | center | space-between | space-around | stretch (allinea multi righe)

# FLEXBOX - CHILDREN

- `order : <integer>` → riordina il contenuto
- `flex : <integer>` → come spartirsi lo spazio
  - box con `flex = 1`
  - box con `flex = 2`
- `align-self` : sovrascrive quello del container

# RESPONSIVE WEB DESIGN

Banalmente: facciamo in modo che il nostro sito funzioni bene su ogni dispositivo.

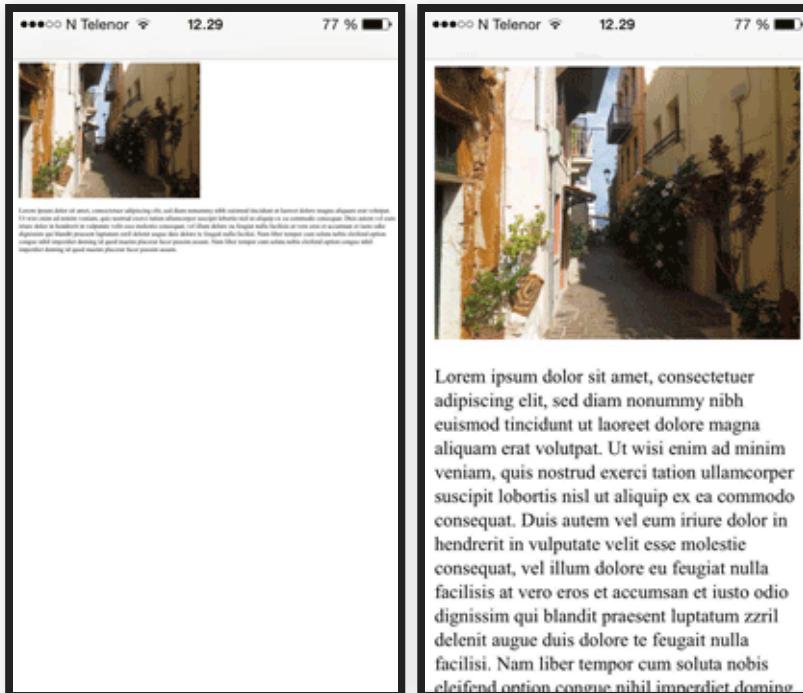
- ridisponendo il contenuto
- magari usando solo HTML e CSS
- non serve necessariamente JavaScript
- spesso disegnati immaginando una griglia di 12 colonne

# VIEWPORT

La ViewPort è l'area della pagina visibile all'utente.

I browser di smartphone & tablet rimpiccioliscono in automatico la pagina per farla stare nello schermo: evitiamo

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```



# DIFFERENZIAZIONE PER TIPO DI MEDIA

Si può fare in modo che alcune regole siano applicate solo a certi media (o media con certe caratteristiche):

- usando link :

```
<link ... media="screen" href="css1.css">
<link ... media="print,tty" href="css2.css">
```

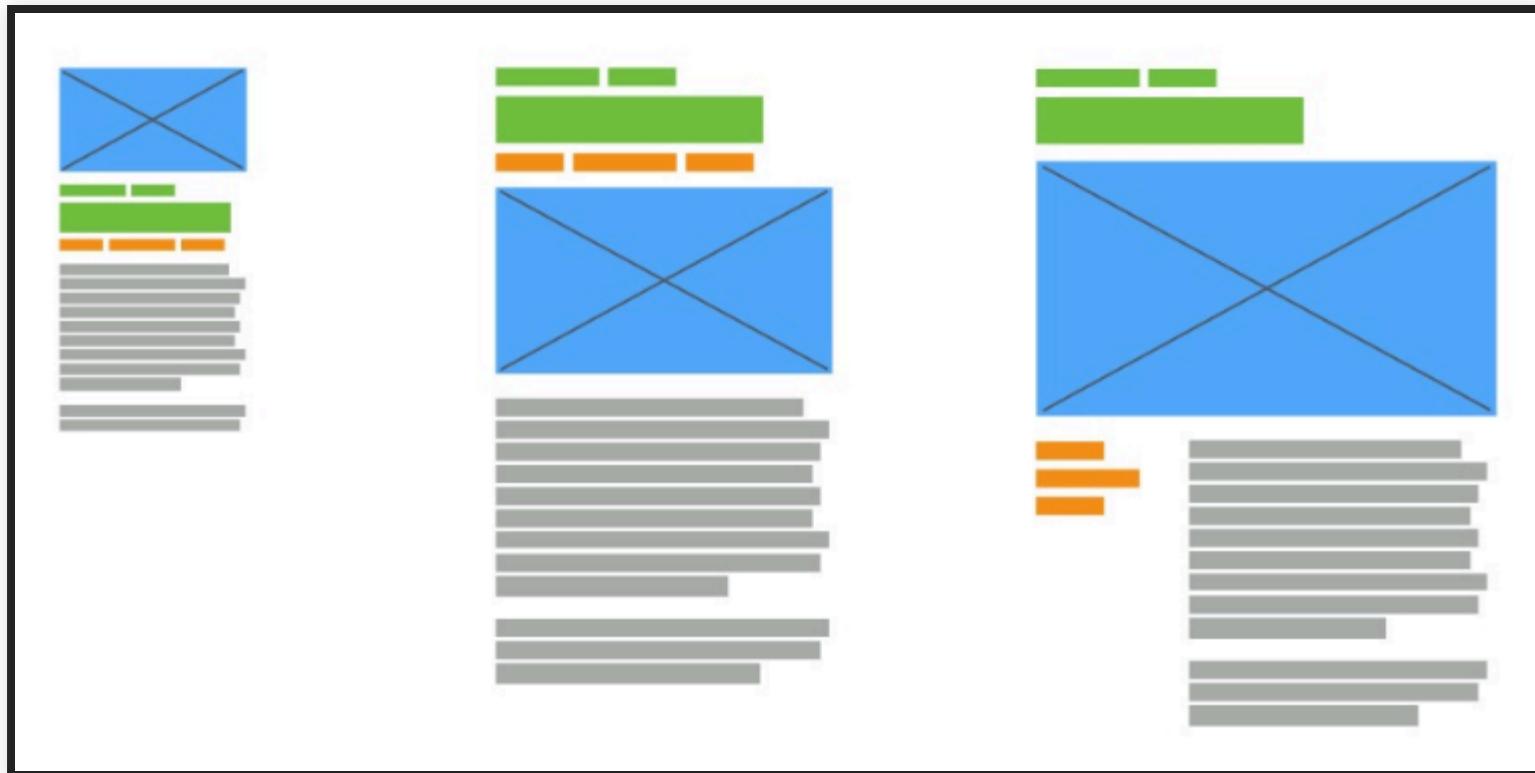
- usando @media :

```
@media print {
    /* style sheet for print goes here */
}
```

Alcuni media: screen, print, handheld, braille, speech, tty, ...

# GRID LAYOUT

Vogliamo un layout di questo tipo:



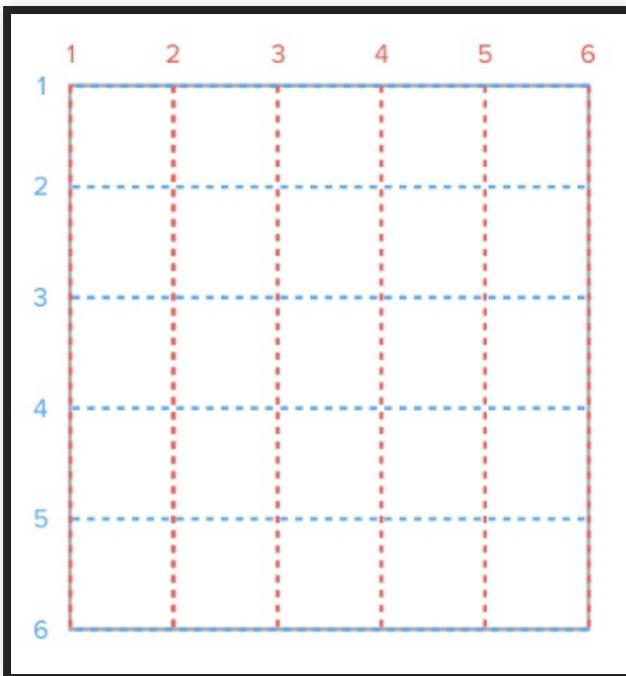
Come posso passare tra i tre layout o fare il terzo?

**Nota:** Slides ispirate dalla presentazione di [Morten Rand-Hendriksen](#)

# GRID LAYOUT

## Terminologia

- Grid **container**: l'elemento che contiene una grid, si imposta `display: grid;`
- Grid **item**: *descendant* del grid container
- Grid **line**: linea orizzontale o verticale che delimita la griglia
  - sono numerate (doppia numerazione: 1...N e -N...-1)



# GRID LAYOUT

## Terminologia

- Grid **container**: l'elemento che contiene una grid, si imposta `display: grid;`
- Grid **item**: *descendant* del grid container
- Grid **line**: linea orizzontale o verticale che delimita la griglia
  - sono numerate (doppia numerazione: 1...N e -N...-1)
- Grid **cell**: come la cella di una tabella
- Grid **area**: area rettangolare tra quattro *lines* → una o più celle
- Grid **track**: spazio tra due *lines* adiacenti (righe e colonne)
- Grid **gap**: spazio vuoto tra due *tracks*

# GRID LAYOUT

Come si usa?

1. Definire una griglia
2. Posizionare gli elementi nella griglia

That's it!

# GRID LAYOUT

Definire la griglia:

- `display: grid;`
- `grid-template-columns` e `grid-template-rows`: disegnano le righe; ricevono una lista di valori che identificano le distanze tra le righe.

Unità di misura: em, px, %, fr (frazioni), auto (si adatta al contenuto).

```
1 main{  
2   display: grid;  
3   grid-template-columns: 2fr 1fr 1fr;  
4 }
```

Finito! I grid items si dispongono all'interno (top-bottom, left-right).

# GRID LAYOUT

Prendere più celle

- `grid-column: x/y;` → dalla riga verticale X alla Y
- `grid-row: x/y;` → dalla riga orizzontale X alla Y

**Problema:** è difficile lavorare con i numeri, si può fare confusione (anche se in realtà potrei dare dei nomi alle linee)

# GRID TEMPLATES

Uso delle **descrizioni testuali** per dare dei **nomi** alle  
celle del grid container...

```
1 main{  
2   display: grid;  
3   grid-template-columns: 2fr 1fr 1fr;  
4   grid-template-rows: 1fr 1fr 1fr;  
5   grid-template-areas:  
6     "titolo titolo titolo"  
7     "principale testata testata"  
8     "principale laterale laterale";  
9 }
```

...e nei grid item specifico il nome

```
1 h1{  
2   grid-area: titolo;  
3 }
```

# GRID TEMPLATES

E la magia avviene con le **media query**...

```
1 @media screen and (max-width: 700px) {  
2   main {  
3     grid-template-areas:  
4       "titolo titolo titolo"  
5       "testata testata testata"  
6       "principale principale laterale";  
7   }  
8 }
```

# GRID LAYOUT

Alcuni dettagli:

- le griglie si possono annidare senza problemi;
- **buon supporto** nei diversi browsers
- ...se avete paura, esiste @supports (grid-area: auto){...}

[applico la regola se supportata](#)

# GRID LAYOUT

Operativamente:

1. disegnate per i cellulari (e funge da fall-back)
2. estendere con grid-templates per le altre risoluzioni

Alcuni riferimenti utili:

- Grid by Example
- CSS Tricks

# CSS

## CSS NEI VARI BROWSER

# FLEXBOX

Ma funziona sempre?

NO!

- CSS3 è ancora in scrittura
- diverse modifiche nel tempo
- **il nome stesso della property cambia**

E non solo lui, il problema si applica anche ad altro (es:  
column )

**Soluzione:** prefissi

# PREFISSI

Per specifica, le property non inizieranno mai con

- "\_"
- "\_\_"

```
1 .foo {  
2   display: -webkit-box;  
3   display: -moz-box;  
4   display: -ms-flexbox;  
5   display: -webkit-flex;  
6   display: flex;  
7   -webkit-box-orient: horizontal;  
8   -moz-box-orient: horizontal;  
9   -webkit-box-direction: normal;  
10  -moz-box-direction: normal;  
11  -webkit-flex-direction: row;  
12  -ms-flex-direction: row;  
13  flex-direction: row;  
14 }  
15 }
```

# STRUMENTI

Ma devo sempre fare tutto a mano?

No! Posso farmi aiutare

- Strumenti integrati nell'editor (es: [Emmet](#))
- Strumenti on-line (es: [AutoPrefixer](#))
- Strumenti usabili off-line (es: [Less](#))
- Framework che risolva il problema per me (es: [Bootstrap](#))

# BOOTSTRAP

Una riga per domarli, e dal buio liberarli...

```
<link rel="stylesheet"
```

```
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/[...]" >
```

Opzionale: un po' di JavaScript

- JQuery
- Popper (popup a lato di elementi)
- BootstrapJS

Conviene leggerne la guida ufficiale

# LESS

Strumento di sviluppo per CSS scritto in JavaScript  
usabile

- off-line (linea di comando)
- in codice JS
- client-side (basse performance, comodo per lo sviluppo)

```
1 <link rel="stylesheet/less" type="text/css" href="styles.less" />
2 <script src="less.js" type="text/javascript"></script>
```

# LESS

- Variabili:

```
@rosso: #FF0000;  
a {color: @rosso}
```

→ a {color: #ff0000}

- Nidificazione:

```
div{  
  width: 200px;  
  p{  
    font-size: 20px;}}
```

→ div{ width: 200px; }

→ div p{ font-size: 20px; }

# LESS

- **Mixin (macro):**      gruppo di regole che voglio usare insieme

```
.mixin{  
  color: red;  
  font-weight:bold;  
}  
  
p{  
  font-size:18px;  
  .mixin;  
}
```

→ .mixin{color: red; font-weight:bold;}

→ p{font-size: 18px; color: red; font-weight:bold;}

# LESS

- Mixin (macro) - esclusi dall'output:

```
.mixin(){  
  color: red;  
  font-weight:bold;  
}  
  
p{  
  font-size:18px;  
  .mixin;  
}
```

→ p{font-size: 18px; color: red; font-weight:bold;}

# LESS

- Mixin (macro) - **parametrici:**

```
.mixin(@param){  
  color: @param;  
  font-weight:bold;  
}  
  
p{  
  font-size:18px;  
  .mixin(red);  
}
```

# LESS

- Funzioni:
  - darken(colore, percentuale)
  - greyscale(colore)
  - ...molte altre

# ESERCIZIO CSS-1

Usando il documento HTML dell'esercizio HTML-1, scrivere un CSS che faccia rappresentare il documento come un codice miniato medioevale.



Esempi di rappresentazione

- due colonne, bordi larghi
- la prima lettera del primo testo della prima sezione deve essere enorme (titoli esclusi, il resto del testo circonda la lettera grande )

# ESERCIZIO CSS-1 → DETTAGLI

- Nessuno/pochi cambiamenti all'HTML
- Usare i Google Web Fonts
- Barra di navigazione in alto, allineata a destra
- Se la risoluzione è inferiore a 800px, gli elementi della barra vanno centrati
- Se la risoluzione è inferiore a 600px,
  - deve avere una colonna sola
  - la barra di navigazione dev essere su una sola colonna
- Se stampato, deve avere margine laterale di 3cm

# ESERCIZIO CSS-2

Riprendere il documento HTML: il contenuto di ogni section deve essere invisibile di default e diventare visibile in qualche condizione legata alle azioni dell'utente.

Ad esempio:

- un checkbox per articolo selezionato
- se si passa sopra all'elemento riferito con la navbar
- ...

# ESERCIZIO CSS-3

Ancora qualche modifica:

- Inserire dei contatti con degli indirizzi email e dei link, quindi
  - se il link è di una mail, anteporci il carattere ✉ (\2709)
  - ogni indirizzo appaia per esteso dopo il testo link, tra parentesi
- Se mostrato a schermo con risoluzione di 600px o più, l'articolo deve essere largo 21 cm, avere sfondo bianco ed apparire in rilievo sopra ad uno sfondo grigetto.
- Se stampato, deve avere la dimensione di un A5 con un margine laterale di 3cm e creare un nuovo foglio per ogni `section` → [@page](#) ; qualche aiuto

**Nota:** solo *HTML* e *CSS*

# **PROGRAMMAZIONE**

# **WEB *BROWSER-SIDE***

# CODICE COMPILATTO (O QUASI)

Posso chiedere al browser di eseguire del codice compilato (o di una VM)

- ActiveX → obsoleto, solo per Windows
- Flash → obsoleto
- Java Applet → uso residuale (es: firma elettronica con smart card)
- WebAssembly → ancora sperimentale

# RAPPRESENTARE UN DOCUMENTO HTML

## HYPertext Markup Language (HTML)

Linguaggio usato per descrivere il contenuto e la struttura dell'informazione di un documento web

In realtà l'obiettivo di HTML è esplicitamente più ampio:

*This specification is limited to providing a semantic-level markup language and associated semantic-level scripting APIs for authoring accessible pages on the Web ranging from static documents to dynamic applications.*

# SCRIPTING

## SCRIPTING WEB

Esecuzione di **codice** dell'autore nel **contesto** del documento web

Codice? → **JavaScript**

... ma non solo

- SVG scripting
- TypeScript
- GO
- ...

# CONTESTO

Durante l'esecuzione, il codice può accedere al **DOM tree**, quindi **all'informazione dentro al documento**

- in lettura → ottenere elementi, attributi, ... del DOM tree
- in scrittura → cambiare attributi, testo, struttura, ... del DOM tree

Ma anche a risorse esterne (la finestra, lo storage, ...)

# ACCESSO IN SCRITTURA DEL DOM

Documento HTML → DOM tree → rappresentazione

Manipolare il DOM implica cambiare la  
rappresentazione, in tempo reale!

# DOM TREE E SCRIPTING

La specifica descrive anche l'interfaccia dei vari elementi:

*DOM interface:*

```
[NamedConstructor=Image(),
 NamedConstructor=Image(in unsigned long width),
 NamedConstructor=Image(in unsigned long width, in unsigned long height)]
interface HTMLImageElement : HTMLElement {
    attribute DOMString alt;
    attribute DOMString src;
    attribute DOMString useMap;
    attribute boolean isMap;
    attribute unsigned long width;
    attribute unsigned long height;
    readonly attribute unsigned long naturalWidth;
    readonly attribute unsigned long naturalHeight;
    readonly attribute boolean complete;
};
```

È la documentazione dell'interfaccia DOM  
dell'elemento `img`

# QUANDO VIENE ESEGUITO IL CODICE?

Nota:

- il documento HTML è l'informazione
  - il documento CSS dice come rappresentare l'informazione
- non hanno un quando...

# "QUANDO" VIENE ESEGUITO IL CODICE?

Viene eseguito quando viene *trovato*  
Il browser legge (fa il parsing) il documento HTML,  
quando trova lo script lo esegue

# COME SI INCLUDE LO SCRIPT?

Due metodi principali:

- **embedded nel documento:**

```
<script type="text/javascript">  
    /* fai qualcosa */  
</script>
```

- **come documento di testo esterno:**

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

# src E INIZIO DELL'ESECUZIONE

Per gli script esterni, si può specificare l'attributo **defer** :

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

→ l'esecuzione viene ritardata a quando il browser è arrivato in fondo al documento

**Nota:** per gli embedded script, vedere [\*onload\*](#)

# **defer O NON defer ?**

Qual è la differenza?

Il codice può accedere al DOM...



HTML letto a metà → DOM completo a metà

# PROGRAMMAZIONE AD EVENTI

## HANDLER

Associazione di un **pezzo di codice** (funzione) ad un **evento**

c'è un click su questo bottone → fai queste cose

il browser è arrivato in fondo al documento → fai queste cose

L'associazione si può definire:

- nel documento HTML

```
<p onclick="/* pezzo di codice */">click me!</p>
```

- nel codice: vedremo

# LINGUAGGIO JAVASCRIPT

# PREMESSA

Cosa impariamo?

- JavaScript: come linguaggio e come linguaggio nel web
- Un'altra istanza del problema "linguaggio di programmazione":
  - come dare istruzioni ad una macchina priva di buon senso?
  - la risposta è data da uomini: può essere imperfetta

# BREVE STORIA

- 1995: inizialmente chiamato Mocha e rilasciato come LiveScript da Netscape
- 1996: Microsoft ne fa un port in JScript (poco compatibili)
- 1997: inizia la standardizzazione da parte di Ecma, come ECMAScript
- 2011: standard ISO (versione 5.1)
- 2015: ECMAScript 6 (nome ufficiale ECMAScript 2015)
- 2017: versione 8

Nel frattempo:

- popolarità sempre crescente
- utilizzo fuori dal browser sempre più reale

# JAVASCRIPT ENGINES

Engine = coppia (VM, dialetto)

Ce ne sono tanti:

- V8 (in Chrome e Node.js)
- Spidermonkey (Firefox, ...)
- Nitro/JavaScriptCode in Safari
- ...

Sofisticati: ad esempio, JIT

# JAVASCRIPT VS JAVA

- Sintassi simile (non uguale), più rilassata
- Loose typing
- Le funzioni sono oggetti!
- Eccezioni "sileziose"
- Imperfetto, data la sua storia

# SINTASSI

Non la vediamo nel dettaglio

Alcune peculiarità

- parole chiave: boolean , break , delete , extends , static ,..., molte non utilizzate
- punto e virgola: non è necessario, ma **raccomandato**
- tolleranza per gli "spazi"

Vedremo, man mano, altro

# TIPI

5 tipi di base immutabili:

- numeri
- stringhe
- booleani
- null
- undefined

Tutti gli altri valori sono **oggetti** (mutabili): array, funzioni, regex e oggetti.

# TIPI E `typeof`

`typeof` `x` è una stringa che descrive il tipo di `x`

```
typeof 1          // → "number"
typeof "a"        // → "string"
typeof true       // → "boolean"
typeof {a: 1}     // → "object"
typeof undefined  // → "undefined"
typeof null       // → ???
```

# TIPI E `typeof`: `null`

```
typeof null           // → "object" !!!
```

Ma perché???

In memoria: 32 bit per valore

- 3 bit per il tipo (circa)
  - 000 → object [ `Null` : dato = 0x00]
  - 001 → signed int
  - 010 → double
  - 100 → string
  - 110 → boolean
- resto per il dato

# TIPI E `typeof`: null

`typeof null` // → "object"!!!

Decodifica valore (codice originale):

```
1 if (JSVAL_IS_UNDEFINED(v)) {
2     type = JSTYPE_UNDEFINED;
3 } else if (JSVAL_IS_OBJECT(v)) {
4     obj = JSVAL_TO_OBJECT(v);
5     if (obj [...] clasp == js_FunctionClass) {
6         type = JSTYPE_FUNCTION;
7     } else {
8         type = JSTYPE_OBJECT;
9     }
10 } else if (JSVAL_IS_NUMBER(v)) {
11     type = JSTYPE_NUMBER;
12 } else if (JSVAL_IS_STRING(v)) {
13     type = JSTYPE_STRING;
14 } else if (JSVAL_IS_BOOLEAN(v)) {
15     type = JSTYPE_BOOLEAN;
16 }
17 return type;
```

# TIPI E `typeof`

`typeof x` è una stringa che descrive il tipo di `x`

```
typeof 1          // → "number"
typeof "a"        // → "string"
typeof true       // → "boolean"
typeof {a: 1}     // → "object"
typeof undefined  // → "undefined"
typeof null       // → "object" !!!
```

`typeof typeof 44` ?

C'è una sesta possibilità: vedremo.

**Nota:** Sarebbe stato meglio se `typeof null` fosse `"null"`.

# **typeof E SINTASSI**

Sintassi rilassata...

Quali e perché funzionano?

1. `typeof 1?`
2. `typeof(1)?`
3. `typeof  
(1)?`

# typeof E SINTASSI

Sintassi rilassata...

Quali e perché funzionano?

1. `typeof 1?`
2. `typeof(1)?`
3. `typeof  
(1)?`

`typeof` è un **operatore**, e quanto messo tra parentesi  
è una espressione da valutare.

# TIPI IMMUTABILI

Numeri, stringhe e booleani sono object-like: è possibile accedere ai metodi

```
(22) .toExponential()    \\ → "2.2e+1"  
"job".charAt(2)          \\ → "b"  
true.toLocaleString()  \\ → "true"
```

ma sono immutabili!

# NUMERI

C'è un solo tipo di numero: **float** (IEEE 754 Double Precision)

Ricordate i problemi che comportano?

# NUMERI

C'è un solo tipo di numero: **float** (IEEE 754 Double Precision)

- operazioni con decimali si comportano da float:

```
0.1 + 0.2 == 0.3 /* → false */
```

- operazioni con interi sono esatte (sicuri?)

```
999999999999999 == 1000000000000000 /* → true */
```

# null VS undefined

Un identificatore riferisce:

- `undefined` se non gli è stato associato un valore (diverso da `undefined`)
- `null` solo se è stato associato a `null`

Perché esistono entrambi? C'è un perché?

- ragioni "storiche" (le eccezioni sono nate dopo)
- una proprietà che esiste, ma senza valore, si può distinguere da una proprietà che non esiste

# OGGETTI

Un oggetto è una **collezione mutabile di proprietà**.

- Ogni proprietà ha un nome (stringa) ed un valore (qualsiasi tipo)
- Mutabile:
  - Si possono modificare i valori delle proprietà
  - Si possono aggiungere o rimuovere proprietà

# OBJECT LITERAL

Collezione di coppie chiave-valore separate da virgole.

```
var simba = {  
    name: "simba",  
    animalType: "dog",  
    legs: 4,  
    "sound": "bark",  
    collar: {  
        material: "leather",  
        size: "M"  
    }  
};
```

"animalType" , "dog" è una proprietà: "animalType" è il nome (tipo stringa), "dog" è il valore (tipo stringa).

Il valore della proprietà con nome "collar" è di tipo oggetto.

# var STATEMENT

```
1 var simba = {  
2   name: "simba",  
3   animalType: "dog",  
4   legs: 4,  
5   "sound": "bark"  
6 }
```

1. Definisce l'identificatore simba visibile nello scope.
2. Lo associa al (gli fa riferire il) valore oggetto.

Qual è lo scope? Vedremo.

# ACCESSO AI VALORI DELLE PROPRIETÀ DI UN OGGETTO

1. simba.name
2. simba["name"]

Entrambe le espressioni accedono alla proprietà name dell'oggetto riferito da simba .

La seconda è necessaria se il nome della proprietà ha certe caratteristiche: es. simba["è bello"] .

Se simba riferisce undefined o null , allora TypeError exception.

**Nota:** . e [] sono operatori di refinement

# ACCESSO AI VALORI DELLE PROPRIETÀ: SINTASSI

Gli spazi vanno a piacere:

- simba.name
- simba["name"]
- simba .name
- simba. name
- simba . name
- simba ["name"]
- ...

Sono tutti equivalenti e sintatticamente corretti.

Ma stilisticamente no...

# ACCESSO AI VALORI DI UN OGGETTO

- Lettura proprietà:

```
var name = simba.name;
```

Se non è definito, vale `undefined`

- Aggiunta/modifica proprietà:

```
simba.nickname = "Cane bianco";
```

Se non è definito, aggiunge, altrimenti modifica.

- Rimozione proprietà:

```
delete simba.nickname;
```

# RIFERIMENTI

Gli oggetti sono passati sempre per riferimento.

```
var simba = {  
    name: "simba",  
    collar: {  
        material: "leather",  
        size: "M"  
    }  
};  
var whitedog = simba;  
var c = whitedog.collar;  
delete whitedog.collar;
```

whitedog.collar ?

simba.collar ?

c ? →

# RIFERIMENTI

Gli oggetti sono passati sempre per riferimento.

```
var simba = {  
    name: "simba",  
    collar: {  
        material: "leather",  
        size: "M"  
    }  
};  
var whitedog = simba;  
var c = whitedog.collar;  
delete whitedog.collar;
```

whitedog.collar ? → undefined

simba.collar ?

c ? →

# RIFERIMENTI

Gli oggetti sono passati sempre per riferimento.

```
var simba = {  
    name: "simba",  
    collar: {  
        material: "leather",  
        size: "M"  
    }  
};  
var whitedog = simba;  
var c = whitedog.collar;  
delete whitedog.collar;
```

whitedog.collar ? → undefined

simba.collar ? → undefined

c ? →

# RIFERIMENTI

Gli oggetti sono passati sempre per riferimento.

```
var simba = {  
    name: "simba",  
    collar: {  
        material: "leather",  
        size: "M"  
    }  
};  
  
var whitedog = simba;  
var c = whitedog.collar;  
delete whitedog.collar;
```

whitedog.collar ? → undefined

simba.collar ? → undefined

c ? → Object {material: "leather", size: "M"}

# ITERAZIONE SULLE PROPRIETÀ: "REFLECTION"

```
var simba = {  
  name: "simba",  
  legs: 4  
};  
for (var propName in simba) {  
  console.log("simba."+propName+" = "+simba[propName]+", of type "+(typeof simba[propName]));  
}
```

# FUNZIONI

Una funzione è un oggetto con due proprietà nascoste in più:

- il codice
- il contesto

`typeof f` restituisce "function" se `f` è una funzione.

Ecco il sesto elemento

# FUNCTION LITERAL

```
1 var sum = function(a, b) {  
2     var c = a+b;  
3     return c;  
4 }  
5 sum(2,3); /*→ 5*/
```

sum riferisce un oggetto funzione, di cui la sequenza dei due statement (linee 2 e 3) è il **body**.

# FUNCTION NAME

```
1 var sum = function(a, b) {  
2   return a+b;  
3 }  
4 sum.name /* → "" (o il nome della variabile) */
```

Funzione anonima, sum riferisce la funzione.

Questa è l'opzione raccomandata.

**Nota:** "Perché" c'è una proprietà name? Chi l'ha messa? Vedremo.

# FUNCTION NAME

```
1 var sum = function summation(a, b) {  
2     return a+b;  
3 }  
4 sum.name /* → "summation" */
```

Funzione con nome, sum riferisce la funzione,  
summation non è un identificatore definito.

Il nome è comunque utile:

- Debugging
- Funzioni ricorsive

# FUNCTION NAME

```
1 function summation(a, b) {  
2     return a+b;  
3 }  
4 summation.name /* → "summation" */
```

Funzione con nome, `summation` è definito e riferisce la funzione.

# OGGETTO (O) FUNZIONE?

Le funzioni *sono* oggetti, cioè collezioni mutabili di proprietà!

```
1 var sum = function(a, b) {  
2   return a+b;  
3 }  
4 sum.p = "ciao";
```

# SCOPE

Lo scope è l'ambito nel quale sono definiti (visibili) gli identificatori.

Lo scope in JavaScript è il body della funzione.

*Nota: nonostante la sintassi a blocchi (con le graffe), lo scope non è il blocco!*

# SCOPE

Lo scope in JavaScript è il body della funzione.

```
1 var maxSumProd = function(a, b) {  
2     var s = a+b;                      /* s is defined */  
3     var p = a*b;                      /* s, p are defined */  
4     for (var i = 0; i<4; i++) {  
5         var c = i;                    /* s, p, i, c are defined */  
6     }  
7     if (s > p) {                     /* s, p, i, c are defined */  
8         return s;  
9     } else {  
10        return p;  
11    }  
12};
```

Diverso da Java!

# SCOPE E GERARCHIA

Gli scope sono gerarchici: nello scope di una funzione sono visibili gli identificatori (tranne `this` di cui parleremo dopo e `arguments`) definiti nello scope in cui la funzione è definita.

```
1 var fun = function() {
2   var a = 1;
3   var b = 2;
4   var innerFun = function() {
5     var c = a+b;
6     return c;
7   };
8   return innerFun();
9 };
```

# SCOPE GLOBALE

`var` definisce una variabile: se omesso, si sottointende lo **scope globale**

- `var a;` → `a` è definito (ma non cambia il riferimento) nello scope "locale"
- `var a = 0;` → `a` è definito e riferisce 0 nello scope "locale"
- `a = 0;`
  - se `a` era definito in uno scope "padre", allora ora riferisce 0
  - altrimenti viene definito nello scope globale e riferisce 0

# SCOPE GLOBALE

Attenzione!

```
1 var calculateTopTenCosts = function() {  
2     var sum = 0;  
3     for(top = 0; top < 10; top++) {  
4         sum += getRankCosts(top);  
5     }  
6     return sum;  
7 };
```

# SCOPE GLOBALE

Attenzione!

```
1 var calculateTopTenCosts = function() {
2     var sum = 0;
3     for(top = 0; top < 10; top++) {
4         sum += getRankCosts(top);
5     }
6     return sum;
7 };
```

- manca `var` prima della definizione di `top` nel ciclo
- `top` è una variabile globale (esecuzione in browser): l'oggetto `window` più elevato
- il ciclo non viene eseguito ( `top` non è un numero)

# SCOPE DI BLOCCO

`var` inserisce la variabile nello scope della funzione.

E se ne volessi uno a livello di blocco?

Con EcmaScript 6 si può: `let`

# INVOCARE UNA FUNZIONE

Ci sono **quattro** *Function Invocation Patterns*:

1. Method Invocation
2. Function Invocation
3. Apply Invocation
4. Constructor Invocation

Cosa cambia?

- Oltre ai parametri dichiarati, ogni funzione ne riceve altri due:
  - `this`
  - `arguments`
- `this` varia in base all'*invocation pattern*

# 1. METHOD INVOCATION

Se e solo se:

- la funzione è il valore di una proprietà di un oggetto (*metodo*)
- viene invocata con la refinement di quell'oggetto (.)

allora:

- `this` è l'oggetto in cui è contenuta
- gli argomenti sono le valutazioni delle espressioni passate tra parentesi **Nota:** se sono di meno, i mancanti sono `undefined`

# 1. METHOD INVOCATION

```
1 var o = {  
2     n: 3,  
3     sum: function(m) {  
4         return this.n+m;  
5     },  
6     dec: function(m) {  
7         return n-m;  
8     }  
9 };
```

o.sum(2) ?

o.dec(1) ?

var f = o.sum; f(2) ?

var f = o.sum(); f(2) ?

# 1. METHOD INVOCATION

```
1 var o = {  
2     n: 3,  
3     sum: function(m) {  
4         return this.n+m;  
5     },  
6     dec: function(m) {  
7         return n-m;  
8     }  
9 };
```

o.sum(2) ? → 5

o.dec(1) ?

var f = o.sum; f(2) ?

var f = o.sum(); f(2) ?

# 1. METHOD INVOCATION

```
1 var o = {  
2     n: 3,  
3     sum: function(m) {  
4         return this.n+m;  
5     },  
6     dec: function(m) {  
7         return n-m;  
8     }  
9 };
```

o.sum(2) ? → 5

o.dec(1) ? → errore! “n undefined”

var f = o.sum; f(2) ?

var f = o.sum(); f(2) ?

# 1. METHOD INVOCATION

```
1 var o = {  
2     n: 3,  
3     sum: function(m) {  
4         return this.n+m;  
5     },  
6     dec: function(m) {  
7         return n-m;  
8     }  
9 };
```

o.sum(2) ? → 5

o.dec(1) ? → errore! “n undefined”

var f = o.sum; f(2) ? → NaN

var f = o.sum(); f(2) ?

# 1. METHOD INVOCATION

```
1 var o = {  
2   n: 3,  
3   sum: function(m) {  
4     return this.n+m;  
5   },  
6   dec: function(m) {  
7     return n-m;  
8   }  
9 };
```

o.sum(2) ? → 5

o.dec(1) ? → errore! “n undefined”

var f = o.sum; f(2) ? → NaN

var f = o.sum(); f(2) ? → “f is not a function”

# METHOD INVOCATION E `this`

`this` non è opzionale!

**Nota:** *a parità di semantica.*

# 2. FUNCTION INVOCATION

Quando la funzione non è la proprietà di un oggetto e viene quindi invocata **direttamente**:

```
var sum = add(3, 4); // sum is 7
```

- Gli argomenti: come prima.
- **this** è l'oggetto globale.
  - Cos'è l'oggetto globale? Dipende dal contesto.
  - In un browser, è l'oggetto `Window` del DOM
  - **Attenzione:** non è il `this` dell'eventuale outer function!

# 2. FUNCTION INVOCATION

Se fosse fatto bene, `this` sarebbe quello della funzione esterna...

```
1 var myObject = {  
2     value: 1,  
3     double: function() {  
4         var helper = function() {  
5             this.value = this.value * 2;  
6         };  
7         helper();  
8     }  
9 }  
10 console.log(myObject.value); /* → 1 */  
11 myObject.double(); /* Method Invocation */  
12 console.log(myObject.value); /* → ?? */
```

# 2. FUNCTION INVOCATION

Se fosse fatto bene, `this` sarebbe quello della funzione esterna...

```
1 var myObject = {  
2   value: 1,  
3   double: function() {  
4     var helper = function() {  
5       this.value = this.value * 2;  
6       helper(); /*Function Invocation*/  
7     }  
8   }  
9   console.log(myObject.value); /* → 1 */  
10  myObject.double(); /*Method Invocation*/  
11  console.log(myObject.value); /* → 1 */
```

# 2. FUNCTION INVOCATION

## Workaround

```
1 var myObject = {  
2   value: 1,  
3   double: function() {  
4     var that = this;  
5     var helper = function() {  
6       that.value = that.value * 2;  
7     };  
8     helper(); /*Function Invocation*/  
9   }  
10 }  
11 console.log(myObject.value); /* → 1 */  
12 myObject.double(); /*Method Invocation*/  
13 console.log(myObject.value); /* → 2 */
```

# 3. APPLY INVOCATION

Ogni funzione ha anche un metodo `apply` : quando invocato, il primo parametro sarà `this` e il secondo sarà `arguments`.

```
1 var sum = function(a, b) {  
2   return a+b;  
3 };  
4 sum.apply(undefined, [2, 3]); /* → 5*/
```

**Nota:** `[2, 3]` è un array literal, vedremo.

# 3. APPLY INVOCATION

```
1 var myObject = {  
2   value: 1,  
3   double: function() {  
4     var helper = function() {  
5       this.value = this.value * 2;  
6     };  
7     helper.apply(this); /*Apply Invocation*/  
8   }  
9 }  
10 console.log(myObject.value); /* → 1 */  
11 myObject.double(); /*Method Invocation*/  
12 console.log(myObject.value); /* → 2 */
```

# 4. CONSTRUCTOR INVOCATION ( new )

Se una funzione è invocata premettendo la keyword `new` allora:

- `this` è un nuovo oggetto
- se il valore di ritorno non è un oggetto, viene restituito il valore di `this`

Il nuovo oggetto ha un link nascosto al valore della proprietà `prototype` della funzione.

# 4. CONSTRUCTOR INVOCATION ( new )

Assomiglia ad un costruttore, ma si può usare anche a sproposito.

Per **convenzione** (e per evitare di usarle a sproposito) le funzioni pensate come costruttori hanno il nome che inizia con maiuscola.

# 4. CONSTRUCTOR INVOCATION

```
1 var Dog = function(name, size) {  
2     this.name = name;  
3     this.size = size;  
4     this.sound = "bark";  
5     this.makeSound = function() {  
6         return this.sound;  
7     }  
8 };  
9  
10 var whiteDog = new Dog("Simba", "M");  
11 var brownDog = new Dog("Gass", "M");
```

# 4. CONSTRUCTOR INVOCATION

```
1 var Dog = function(name, size) {  
2     this.name = name;  
3     this.size = size;  
4     this.sound = "bark";  
5     this.makeSound = function() {  
6         return this.sound;  
7     }  
8 };  
9  
10 var whiteDog = new Dog("Simba", "M");  
11 var brownDog = new Dog("Gass", "M");
```

- valore di `this` dopo la riga 3?
- valore di `this` dopo la riga 5?

# 4. CONSTRUCTOR INVOCATION

```
1 var Dog = function(name, size) {  
2     this.name = name;  
3     this.size = size;  
4     this.sound = "bark";  
5     this.makeSound = function() {  
6         return this.sound;  
7     }  
8 };  
9  
10 var whiteDog = new Dog("Simba", "M");  
11 var brownDog = new Dog("Gass", "M");
```

- valore di `this` dopo la riga 3? → “`Dog {name: "Gass", size: "M"}`”
- valore di `this` dopo la riga 5?

# 4. CONSTRUCTOR INVOCATION

```
1 var Dog = function(name, size) {  
2     this.name = name;  
3     this.size = size;  
4     this.sound = "bark";  
5     this.makeSound = function() {  
6         return this.sound;  
7     }  
8 };  
9  
10 var whiteDog = new Dog("Simba", "M");  
11 var brownDog = new Dog("Gass", "M");
```

- valore di `this` dopo la riga 3? → “`Dog {name: "Gass", size: "M"}`”
- valore di `this` dopo la riga 5? → “`Dog {name: "Gass", size: "M", sound: "bark", makeSound: function}`”

# 4. CONSTRUCTOR INVOCATION: A SPROPOSITO

```
1
2 var Dog = function(name, size) {
3     this.name = name;
4     this.size = size;
5     this.sound = "bark";
6     this.makeSound = function() {
7         return this.sound;
8     }
9 }
10
11 var weirdDog = Dog("Simba", "M");
```

- valore di weirdDog ?

# 4. CONSTRUCTOR INVOCATION: A SPROPOSITO

```
1
2 var Dog = function(name, size) {
3     this.name = name;
4     this.size = size;
5     this.sound = "bark";
6     this.makeSound = function() {
7         return this.sound;
8     }
9 }
10
11 var weirdDog = Dog("Simba", "M");
```

- valore di `weirdDog` ? → “*Undefined*”!

# PARAMETRI E arguments

- arguments è un (quasi) array
  - in realtà è un oggetto con un metodo length...
- arguments può contenere più parametri di quelli previsti dalla funzione stessa:

```
1 function myConcat(separator) {  
2     var result = '';  
3     var i;  
4     for (i = 1; i < arguments.length; i++) {  
5         result += arguments[i] + separator;  
6     }  
7     return result;  
8 }  
9 myConcat(', ', 'red', 'orange', 'blue');  
10 //→ "red, orange, blue, "
```

# PARAMETRI E arguments

I rispettivi valori sono legati:

```
1 var sum = function(a, b) {  
2   var c = a+b;  
3   return c;  
4 }
```

a e arguments[0] hanno sempre lo stesso valore.

# PARAMETRI E arguments

```
1 var surpriseMe = function (a) {  
2   console.log(a+" - "+arguments[0]);  
3   a = 3;  
4   console.log(a+" - "+arguments[0]);  
5 }
```

surpriseMe(2) ?

# PARAMETRI E arguments

```
1 var surpriseMe = function (a) {  
2   console.log(a + " - " + arguments[0]);  
3   a = 3;  
4   console.log(a + " - " + arguments[0]);  
5 }
```

surpriseMe(2) ?

```
2 - 2  
3 - 3
```

# PARAMETRI E arguments

*For non-strict mode functions the array index (defined in 15.4) named data properties of an arguments object whose numeric name values are less than the number of formal parameters of the corresponding function object initially share their values with the corresponding argument bindings in the function's execution context. This means that changing the property changes the corresponding value of the argument binding and vice-versa. This correspondence is broken if such a property is deleted and then redefined or if the property is changed into an accessor property. For strict mode functions, the values of the arguments object's properties are simply a copy of the arguments passed to the function and there is no dynamic linkage between the property values and the formal parameter values.*

## Stranezze di Javascript

**Nota:** ma, per fortuna, c'è [stackoverflow](#)

# DEFAULT PARAMETERS

Una volta si faceva a mano:

```
1 function multiply(a, b) {  
2   b = typeof b !== 'undefined' ? b : 1;  
3   return a * b;  
4 }
```

...ora si può fare nell'intestazione (ECMAScript 2015)

```
1 function multiply(a, b = 1) {  
2   return a * b;  
3 }
```

# REST PARAMETER

Aggrega più parametri in un array:

```
1 function myConcat(separator, ...values) {  
2     var result = '';  
3     var i;  
4     for (i = 0; i < values.length; i++) {  
5         result += values[i] + separator;  
6     }  
7     return result;  
8 }  
9  
10 myConcat(', ', 'red', 'orange', 'blue');  
11 //→ "red, orange, blue, "
```

# VALORE DI RITORNO

Le funzioni hanno sempre un valore di ritorno.

- Se il corpo contiene un return statement, allora (constructor invocation a parte) il valore di ritorno è il valore lì specificato.
- Altrimenti è undefined .

L'esecuzione (ovviamente) termina al raggiungimento del return statement.

# return E SPAZI

La sintassi del return statement è "poco tollerante"  
agli spazi

```
1 var f = function() {  
2   return  
3     {word: "ciao"};  
4 }
```

è equivalente a

```
1 var f = function() {  
2   return undefined;  
3   {word: "ciao"};  
4 }
```

...

# return E SPAZI

Non è equivalente a

```
1 var f = function() {  
2   return {word: "ciao"};  
3 }
```

che, a sua volta, è equivalente a

```
1 var f = function() {  
2   return {  
3     word: "ciao"  
4   };  
5 }
```

# CONVENZIONE SU SCOPE

Per convenzione: tutti i `var` statement vanno all'inizio del body:

```
1 var fun = function() {  
2     var a, b, s;  
3     a = 1;  
4     b = 2;  
5     var innerFun = function() {  
6         var c;  
7         c = a+b;  
8         return c;  
9     }  
10    s = innerFun();  
11    return s;  
12};
```

# SOLO UNA CONVENZIONE?

```
1 var fun = function() {
2   var a = 1;
3   var innerFun = function() {
4     console.log(a);
5   };
6   innerFun();
7 };
8
9 fun() /* stampa 1: è cosa buona e giusta*/
```

# SOLO UNA CONVENZIONE?

```
1 var fun = function() {  
2     var a = 1;  
3     var innerFun = function() {  
4         console.log(a);  
5         var a = 2;  
6     };  
7     innerFun();  
8 };  
9  
10 fun() /* stampa undefined!!! */
```

Perché?

# VARIABLE HOISTING

Internamente, JavaScript mette tutti i `var` all'inizio del body:

```
1 var fun = function() {
2   var a, innerFun;
3   a = 1;
4   innerFun = function() {
5     var a;
6     console.log(a);    /* a in scope locale, è undefined */
7     a = 2;
8   };
9   innerFun();
10 };
11
12 fun() /* stampa undefined*/
```

Perché?

# VARIABLE HOISTING

Perché?

Colpa dell'interprete, che compie due passi:

*For completeness, let's mention that actually at the implementation level things are a little more complex. There are two stages of the code handling, where variables, function declarations, and formal parameters are created at the first stage, which is the stage of parsing and entering the context. In the second stage, the stage of runtime code execution, function expressions and unqualified identifiers (undeclared variables) are created. But for practical purposes, we can adopt the concept of hoisting, which is actually not defined by ECMAScript standard but is commonly used to describe the behavior.*

In realtà le specifiche non dicono nulla!

**Attenzione:** non si applica alle variabili definite con  
let

# FUNCTION HOISTING

Si applica anche alle funzioni, se definite direttamente:

```
1 function esempioOK() {
2   console.log(foo); /* → codice della funzione */
3   foo(); /* → "bar" */
4   function foo() {
5     console.log('bar');
6   };
7 }
```

# FUNCTION HOISTING

Si applica anche alle funzioni, se definite direttamente:

```
1 function esempioNonOK() {  
2     console.log(foo); /* → undefined */  
3     foo(); /* → TypeError: foo is not a function */  
4     var foo = function() {  
5         console.log('bar2');  
6     };  
7 }
```

Perché?

1. Viene fatto l'hoisting della *variabile* `foo`
2. La variabile riceve un valore solo nella riga 4

# CLOSURE

Come facciamo a esporre delle funzionalità ma non le variabili interne?

Prendiamo un contatore:

```
1 var counter = 0;
2 function add() {
3   counter += 1;
4 }
5 add();
6 add();
7 add();
```

- Funziona? Sì
- Il Contatore viene incrementato? Sì
- **Problema:** chi può richiamare `add()` può modificare `counter`

# CLOSURE

E se mettessi la variabile nella funzione?

```
1 var counter = 0;
2 function add() {
3     var counter;
4     counter += 1;
5 }
6 add();
7 add();
8 add();
```

- Funziona? No, alla fine `counter` vale 0.
- Incrementiamo una variabile interna, ma mostriamo quella globale.

# CLOSURE

E se togliessi la variabile globale, ritornando il valore?

```
1 function add() {  
2   var counter;  
3   counter += 1;  
4   return counter;  
5 }  
6 add();  
7 add();  
8 add();
```

- Funziona? No, alla fine counter vale 1.
- La variabile viene ridefinita ad ogni chiamata.

# CLOSURE

Proviamo ad annidiare le funzioni:

```
1 function add() {  
2   var counter = 0;  
3   function plus() {counter += 1;}  
4   plus();  
5   return counter;  
6 }
```

- In JavaScript, tutte le funzioni hanno accesso allo scope che le contiene.
- La funzione `plus` ha accesso allo scopo di ciò che la contiene, quindi anche a `counter`.
- **Problema:** come faccio ad invocare `plus()` ?

# CLOSURE: SOLUZIONE

Funzione che si auto-invoca:

```
1 var add = (function () {
2   var counter = 0;
3   return function () {
4     counter += 1;
5     return counter;
6   }
7 })();
8 add(); /* 1 */
9 add(); /* 2 */
10 add(); /* 3 */
```

- Funziona? Sì, l'invocazione alla fine ritorna 3
- Posso accedere a counter ? No
- Ma come ho fatto??

# CLOSURE: SOLUZIONE

```
1 var add = (function () {
2     var counter = 0;
3     return function () {
4         counter += 1;
5         return counter;
6     }
7 })();
8 add(); add(); add();
```

- add contiene il risultato di una funzione che si auto-invoca
- la funzione che si auto-invoca viene eseguita una volta sola: imposta il contatore, ritorna una funzione e viene cestinata
- add è una funzione → ha accesso allo scope che la contiene → ha accesso a counter , che esiste ancora
- nessun altro ha accesso a counter

# CLOSURE

Si possono generare anche closure più evolute:

```
1 var counter = function() {  
2     var n;  
3     n = 0;  
4     return {  
5         inc: function() {  
6             n = n+1;  
7             return n;  
8         }  
9     };  
10 }();  
11 counter.inc();
```

- la funzione riferita da `inc` è una *closure*
- `n` è una *enclosed variable*

# OPERATORI + E TIPI

Alcuni operatori si applicano anche a tipi inattesi: è opportuno sapere come vanno le cose per evitare sorprese.

Operatore + :

- se entrambi gli operandi sono di tipo number, somma i valori
- se entrambi gli operandi sono stringhe, le concatena
- altrimenti, abbiamo già visto: per esempio, cerca di convertire in stringhe e concatenare gli operandi

**Nota:** spesso fonte di spiacevoli inconvenienti quando si legge un numero dal DOM

# TRUTHY AND FALSEY

Ogni valore in JS ha anche un valore booleano implicito:

## Falsy

- false
- 0 (attenzione: non "0"!)
- "" o '' (stringa vuota)
- null
- undefined
- NaN (teoricamente: è valore di tipo number)
- document.all (ringraziamo [Microsoft](#) per questo)

## Truthy

...in tutti gli altri casi

# TRUTHY AND FALSEY

Possiamo usare una funzione per testare:

```
1 function truthyOrFalsy(a) {  
2     return a ? "truthy" : "falsy";  
3 }
```

- `truthyOrFalsy(0)`
- `truthyOrFalsy(10 == 5)`
- `truthyOrFalsy(1)`
- `truthyOrFalsy(-1)`
- `truthyOrFalsy({})`

# TRUTHY AND FALSY

Possiamo usare una funzione per testare:

```
1 function truthyOrFalsy(a) {  
2     return a ? "truthy" : "falsy";  
3 }
```

- `truthyOrFalsy(0)` → falsy
- `truthyOrFalsy(10 == 5)`
- `truthyOrFalsy(1)`
- `truthyOrFalsy(-1)`
- `truthyOrFalsy({})`

# TRUTHY AND FALSY

Possiamo usare una funzione per testare:

```
1 function truthyOrFalsy(a) {  
2     return a ? "truthy" : "falsy";  
3 }
```

- `truthyOrFalsy(0)` → falsy
- `truthyOrFalsy(10 == 5)` → falsy
- `truthyOrFalsy(1)`
- `truthyOrFalsy(-1)`
- `truthyOrFalsy({})`

# TRUTHY AND FALSEY

Possiamo usare una funzione per testare:

```
1 function truthyOrFalsy(a) {  
2     return a ? "truthy" : "falsy";  
3 }
```

- `truthyOrFalsy(0)` → falsy
- `truthyOrFalsy(10 == 5)` → falsy
- `truthyOrFalsy(1)` → truthy
- `truthyOrFalsy(-1)`
- `truthyOrFalsy({})`

# TRUTHY AND FALSEY

Possiamo usare una funzione per testare:

```
1 function truthyOrFalsy(a) {  
2     return a ? "truthy" : "falsy";  
3 }
```

- `truthyOrFalsy(0)` → falsy
- `truthyOrFalsy(10 == 5)` → falsy
- `truthyOrFalsy(1)` → truthy
- `truthyOrFalsy(-1)` → truthy
- `truthyOrFalsy({})`

# TRUTHY AND FALSEY

Possiamo usare una funzione per testare:

```
1 function truthyOrFalsy(a) {  
2     return a ? "truthy" : "falsy";  
3 }
```

- `truthyOrFalsy(0)` → falsy
- `truthyOrFalsy(10 == 5)` → falsy
- `truthyOrFalsy(1)` → truthy
- `truthyOrFalsy(-1)` → truthy
- `truthyOrFalsy({})` → truthy

# MENTIROSO...

Ci sono **dei valori truthy** che sono `== false`

- "0" è truthy (stringa non vuota), ma "0" `== false` :
  1. `==` → converte la stringa in numero
  2. `0 == false`
  3. `"0" == 0`
  4. → `"0" == false`
- Non è un **caso isolato...**

# == VS ===

Operatore === :

- true, se gli operandi hanno uguale tipo e valore
- false, altrimenti

Operatore == : tenta la conversione

```
1 false == undefined /* false */
2 false == null /* false */
3 null == undefined /* true */
```

Addio transitività dell'uguaglianza: meglio === che == !

# ARRAY

Un array è un oggetto che:

- ha un costrutto literal caratteristico
- ha dei metodi predefiniti

# ARRAY

Gli array sono oggetti

```
1 var a = [2, 3, "pippo"];
2 var a = new Array(2, 3, "pippo"); /* → alternativo */
3 a[0];      /* → 2 */
4 a[2];      /* → "pippo" */
5 a["2"];    /* → "pippo" */
6 a.length;  /* → 3 */
7 a[3];      /* → undefined */
8
9 a.propertyAggiunta = "sono un oggetto";
10 a.length; /* → 3 */
```

Altri metodi interessanti:

- `slice()` → ne copia una fetta,
- `splice()` → rimuovere (e sostituire) elementi,
- ...

# ARRAY: LENGTH

`length` ≠ upper bound → posso superarlo!

```
1 var myArray = [];
2 myArray.length;      /* → 0 */
3 myArray[10000] = "pippo";
4 myArray.length;      /* → 10001 */
```

Aggiungere elementi:

```
1 numbers[numbers.length] = 'shi'; /* oppure... */
2 numbers.push('go');
```

# ARRAY: RIMUOVERE ELEMENTI

Assegnando un valore più piccolo a `length`, gli elementi in coda vengono eliminati.

Posso togliere un elemento qualsiasi, ma si lascia un buco:

```
1 var a = [2, 3, 79];
2 a.length;      /* → 3 */
3 delete a[1];
4 a.length;      /* → 3 */
5
6 typeof(a[1]); /* → "undefined" */
```

...meglio usare il metodo `splice`

# ARRAY: RICONOSCERLI

`typeof (array) → "object"`

Scriviamo una nostra funzione:

```
1 var is_array = function (value) {  
2   return value &&  
3     typeof value === 'object' &&  
4     typeof value.length === 'number' &&  
5     typeof value.splice === 'function' &&  
6     !(value.propertyIsEnumerable('length'));  
7 };
```

- è truthy? not {false, null, undefined, "", 0, NaN}
- è un oggetto?
- ha una proprietà `length` di tipo numerico?
- ha una funzione `splice` ?
- `length` apparirebbe in un `for ... in` ?

# ARRAY: ORDINARLI

Attenzione...

```
[5, 12, 9, 2, 18, 1, 25].sort(); → [1, 12, 18, 2, 25, 5, 9]
```

Il comportamento standard è di ordinare  
alfabeticamente!

Dobbiamo specificare una funzione di sorting:

```
1 [5, 12, 9, 2, 18, 1, 25].sort(function(a, b) {  
2   return a - b;  
3});
```

# ARRAY: ITERAZIONE

Attenzione...

```
1 var a = [2, 3, 79];
2 for (var name in a) {
3   console.log(name)
4 }
5 /*output: 0, 1, 2*/
```

Sto iterando le properties dell'oggetto, comprese quelle aggiunte da me (o da altri framework)!

# ARRAY: ITERAZIONE

Devo procedere alla vecchia:

```
1 var a = [2, 3, 79];
2 for (var i = 0; i < a.length; i += 1) {
3     console.log(a[i]);
4 }
5 /*output: 2, 3, 79*/
```

Oppure usando il metodo `forEach`:

```
1 a.forEach(function(valore) {
2     console.log(valore);
3 });
4 /*output: 2, 3, 79*/
```

# MEMOIZATION

A volte devo riusare calcoli già fatti...

```
1 var fibonacci = function (n)  {
2   if (n < 2) {
3     return n
4   } else {
5     return fibonacci(n - 1) + fibonacci(n - 2);
6   }
7 }
8
9 for (var i = 0; i <= 10; i += 1) {
10   console.log('// ' + i + ': ' + fibonacci(i));
11 }
```

Inefficiente: funzione richiamata 453 volte

# MEMOIZATION

Sfrutto la *closure* per memorizzare i risultati di una precedente elaborazione:

```
1 var fibonacci = (function() {
2     var memo = [0, 1];
3     var fib = function(n) {
4         var result = memo[n];
5         if (typeof result !== 'number') {
6             result = fib(n - 1) + fib(n - 2);
7             memo[n] = result;
8         }
9         return result;
10    };
11    return fib;
12 })();
```

Bene! fibonacci richiamata solo 18 volte

# EREDITARIETÀ CLASS-BASED

Tradizionalmente (Java, C++, C#) ci sono due entità: **classi** e **istanze**.

- **Classe**: definisce le proprietà ed i metodi dell'oggetto che descrive, ma non ne rappresenta mai uno. Es: *Docente*
- **Istanza**: materializzazione di una classe. Es: *Alberto Bartoli, Eric Medvet, Andrea De Lorenzo*

Inoltre:

- definisco separatamente le proprietà ed il costruttore;
- gerarchia definita con il *subclassing* di una classe esistente;
- casting esplicito non richiesto (*polimorfismo*).

# EREDITARIETÀ *PROTOTYPE*- *BASED*

JavaScript è *loosely-typed*: ogni oggetto deriva da un altro oggetto da cui eredita le proprietà, detto **prototype**.

Come si realizza?

# DELEGATION

- Accesso in aggiunta/modifica/rimozione di proprietà, usando il link nascosto
- Accesso in **lettura**:
  - se la proprietà esiste nell'oggetto, restituisce il valore
  - altrimenti, se esiste nel prototype, restituisce il valore
  - altrimenti, se esiste nel prototype del prototype, ...
  - ...
  - altrimenti, undefined
- Accesso in **scrittura**:
  - crea una nuova property nell'oggetto finale → il prototipo non viene modificato.

# ESEMPIO EREDITARIETÀ *PROTOTYPE-BASED*

```
1 public class Employee {  
2     public String name;  
3     public String dept;  
4     public Employee () {  
5         this("", "general");  
6     }  
7     public Employee (String name) {  
8         this(name, "general");  
9     }  
10    public Employee (String name, String dept) {  
11        this.name = name;  
12        this.dept = dept;  
13    }  
14 }
```

```
1 function Employee(name="", dept="general") {  
2     this.name = name;  
3     this.dept = dept;  
4 }
```

# ESEMPIO EREDITARIETÀ *PROTOTYPE-BASED*

```
1 public class Engineer extends Employee {  
2     public String machine;  
3     public Engineer () {  
4         this ("");  
5     }  
6     public Engineer (String mach) {  
7         dept = "engineering";  
8         machine = mach;  
9     }  
10 }
```

```
1 function Engineer(mach = "") {  
2     this.dept = 'engineering';  
3     this.machine = mach;  
4 }  
5 Engineer.prototype = new Employee;
```

# ESEMPIO EREDITARIETÀ *PROTOTYPE-BASED*

```
1 function Employee(name="", dept="general") {  
2     this.name = name;  
3     this.dept = dept;  
4 }  
5 function Engineer(mach = "") {  
6     this.dept = 'engineering';  
7     this.machine = mach;  
8 }  
9 Engineer.prototype = new Employee;  
10 var luca = new Engineer("computer");
```

Quanto valgono:

- luca.machine
- luca.dept
- luca.name

# ESEMPIO EREDITARIETÀ *PROTOTYPE-BASED*

```
1 function Employee(name="", dept="general") {  
2     this.name = name;  
3     this.dept = dept;  
4 }  
5 function Engineer(mach = "") {  
6     this.dept = 'engineering';  
7     this.machine = mach;  
8 }  
9 Engineer.prototype = new Employee;  
10 var luca = new Engineer("computer");
```

Quanto valgono:

- luca.machine → "computer"
- luca.dept
- luca.name

# ESEMPIO EREDITARIETÀ *PROTOTYPE-BASED*

```
1 function Employee(name="", dept="general") {  
2     this.name = name;  
3     this.dept = dept;  
4 }  
5 function Engineer(mach = "") {  
6     this.dept = 'engineering';  
7     this.machine = mach;  
8 }  
9 Engineer.prototype = new Employee;  
10 var luca = new Engineer("computer");
```

Quanto valgono:

- luca.machine → "computer"
- luca.dept → "engineering"
- luca.name

# ESEMPIO EREDITARIETÀ *PROTOTYPE-BASED*

```
1 function Employee(name="", dept="general") {  
2     this.name = name;  
3     this.dept = dept;  
4 }  
5 function Engineer(mach = "") {  
6     this.dept = 'engineering';  
7     this.machine = mach;  
8 }  
9 Engineer.prototype = new Employee;  
10 var luca = new Engineer("computer");
```

Quanto valgono:

- luca.machine → "computer"
- luca.dept → "engineering"
- luca.name → ""

# DYNAMIC INHERITANCE

Il prototipo di un oggetto è un oggetto, se

- aggiungo
- modifico
- rimuovo

una proprietà (d)al prototipo, ciò influisce su tutti gli oggetti che hanno quel prototipo (tramite *delegation*)

```
1 ...
2 Engineer.prototype = new Employee;
3 var luca = new Engineer("computer");
4 Employee.prototype.salary = 100;
5 luca.salary; /* → 100 */
```

# LA FUNZIONE CREATORE

- Creare un oggetto come literal è equivalente a crearlo con constructor invocation di Object
- Creare una funzione come literal è equivalente a crearla con constructor invocation di Function

# LA FUNZIONE CREATORE: OGGETTI

Creare un oggetto come literal è equivalente a crearlo con constructor invocation di Object

```
1 var o = new Object();
```

è equivalente a

```
1 var o = {};
```

e l'oggetto può essere riempito dopo.

Object.prototype è l'oggetto prototipo da cui entrambi derivano.

**Nota:** La creazione come literal è raccomandata

# LA FUNZIONE CREANTE: OGGETTI

Per esempio, posso aggiungere proprietà read-only:

```
1 var o = new Object();
2 Object.defineProperty(o, "prop", {
3     value: "test",
4     writable: false
5 });
6
7 o.prop; /* → "test"*/
8 o.prop = "Ciao";
9 o.prop; /* → "test"*/
```

# LA FUNZIONE CREATANTE: FUNZIONI

Creare una funzione come literal è equivalente a crearla con constructor invocation di Function

```
1 var f = new Function();
```

è equivalente a

```
1 var f = function() {};
```

ma la funzione non può essere riempita dopo (nel suo body).

Function.prototype è l'oggetto da cui entrambi derivano (il prototype).

**Nota:** *La creazione come literal è necessaria*

# INHERITANCE DINAMICA

```
1 var sum = function(a, b) {  
2     return a+b;  
3 };  
4 sum.creator; /* → undefined */  
5  
6 Function.prototype.creator = "Eric Medvet";  
7  
8 sum.creator; /* → Eric Medvet (delegation) */  
9 var prod = function(a, b) {  
10    return a*b;  
11 }  
12  
13 prod.creator; /* → Eric Medvet (delegation) */  
14 prod.creator = "Simba";  
15 prod.creator; /* → Simba (no delegation) */  
16 sum.creator; /* → Eric Medvet */
```

# IMPOSTARE IL PROTOTIPO

```
1 Object.create = function(o) {  
2   var F = function () {};  
3   F.prototype = o;  
4   return new F();  
5 };  
6  
7 var dog = {sound: "bark"};  
8 var coloredDog = Object.create(dog);  
9 coloredDog.sound; /* → bark*/
```

# IMPOSTARE IL PROTOTIPO

```
1 Object.create = function(o) {  
2   var F = function () {};  
3   F.prototype = o;  
4   return new F();  
5 };
```

1. crea una nuova funzione e la aggiunge come proprietà a Object
2. quando create è invocata con argomento o
  1. crea una nuova funzione F vuota
  2. ne setta il prototype a o
  3. restituisce il risultato di new F() (un nuovo oggetto con link nascosto alla proprietà prototype della funzione creante, cioè o )

# AUGMENTING

```
1 Object.create = function(o) {  
2   ...  
3 };
```

1. crea una nuova funzione e la aggiunge come proprietà a Object

*Augmenting di un tipo base ( Object )*

Si può fare anche con stringhe, numeri, ecc...

```
1 String.reverse = function() {  
2   ...  
3 };
```

**Nota:** *non abusatene*

# MONKEY PATCHING

Aggiungere funzionalità nuove alla libreria di sistema  
estendendola

Es: vogliamo aggiungere una funzione a `String` che  
renda maiuscola la prima lettera

```
1 String.prototype.capitalize = function() {  
2   return this[0].toUpperCase() + this.substr(1);  
3 }  
4 const capitalName = 'jacob'.capitalize(); /* → "Jacob" */
```

# MONKEY PATCHING

Sarebbe da evitare!

In caso ci fossero altre librerie che fanno la stessa cosa,  
cosa succederebbe?

In passato il problema si è già verificato: [smooshgate](#)  
Libreria MooTools, molto usata, aveva aggiunto la  
funzione flatten

```
1 Array.prototype.flatten = function() { . . . }
```

# MONKEY PATCHING

Sarebbe da evitare!

In caso ci fossero altre librerie che fanno la stessa cosa,  
cosa succederebbe?

In passato il problema si è già verificato: [smooshgate](#)  
Libreria MooTools, molto usata, aveva aggiunto la  
funzione flatten

```
1 Array.prototype.flatten = function() { . . . }
```

... ma nel 2019 è stato introdotto flatten in  
ECMAScript!

# INHERITANCE DINAMICA E PRIMITIVI

```
1 Object.prototype.version = "7.2.11";
2 "uh".version; /* → 7.2.11 */
```

Perché? "uh" non è un primitivo String?

# PRIMITIVES COERCION

```
1  typeof true; /*"boolean"*/
2  typeof Boolean(true); /*"boolean"*/
3  typeof new Boolean(true); /*"object"*/
4  typeof (new Boolean(true)).valueOf(); /*"boolean"*/
5
6  typeof "abc"; /*"string"*/
7  typeof String("abc"); /*"string"*/
8  typeof new String("abc"); /*"object"*/
9  typeof (new String("abc")).valueOf(); /*"string"*/
10
11 typeof 123; /*"number"*/
12 typeof Number(123); /*"number"*/
13 typeof new Number(123); /*"object"*/
14 typeof (new Number(123)).valueOf(); /*"number"*/
```

Per eseguire "uh".version viene generata un nuovo oggetto String (wrapper), usato solo per invocare la property e poi abbandonato al Garbage Collector.

# PRIMITIVES COERCION

Ma quindi posso aggiungere proprietà ai primitivi?

```
1 var primitive = "september";
2 primitive.vowels = 3;
3 primitive.vowels; /* → ?? */
```

# PRIMITIVES COERCION

Ma quindi posso aggiungere proprietà ai primitivi? **NO!**

```
1 var primitive = "september";
2 primitive.vowels = 3; /* → crea un nuovo wrapper:*/
3 (new String("september")).vowels = 3;
4 primitive.vowels;      /*→ creo un wrapper per
                           leggere la property:*/
5                               /* → undefined*/
6 (new String("september")).vowels; /* → undefined*/
```

# PRIMITIVES UN-WRAPPING

I wrapper tornano facilmente al tipo primitivo:

```
1 var Twelve = new Number(12);
2 var fifteen = Twelve + 3;
3 fifteen; /*15*/
4 typeof fifteen; /*"number" (primitive)*/
5 typeof Twelve; /*"object"; (still object)*/
```

Ma è pur sempre JavaScript:

```
1 if (new Boolean(false)) {
2   console.log("true");
3 } else{
4   console.log("false");
5 }
```

Output?

# PRIMITIVES UN-WRAPPING

I wrapper tornano facilmente al tipo primitivo:

```
1 var Twelve = new Number(12);
2 var fifteen = Twelve + 3;
3 fifteen; /*15*/
4 typeof fifteen; /*"number" (primitive)*/
5 typeof Twelve; /*"object" (still object)*/
```

Ma è pur sempre JavaScript:

```
1 if (new Boolean(false)) {
2   console.log("true");
3 } else{
4   console.log("false");
5 }
```

Output? → "true", perché gli oggetti sono truthy.

Per i booleani, usare il valore: `new Boolean(false).valueOf();`

# ESERCIZIO JS-WORDCOUNTER

Creare una funzione JavaScript `wordCounter` che restituisce una "mappa" <parola, frequenza> calcolata dalla stringa ricevuta in input.

```
1 wordCounter("la mamma di mio figlio è anche mamma di mia fig
```

**Nota:** vedere `String. split()`

**Nota:** E se ci fossero segni di punteggiatura? Vedere espressioni regolari e *literal corrispondente*.

# ESERCIZIO JS-INFINITE FUNCTION

Creare una funzione JavaScript tale per cui.

```
1
2 var f = ...;
3 f(1)                      /* → 1 */
4 f() (3)                    /* → 3 */
5 f() () () () ("a")        /* → "a" */
6 f() ... () ("x")          /* → "x" */
7
8 var g = f;
9 f = 33;
10 g() () () () (f)         /* → 33 */
```

# JAVASCRIPT

## JAVASCRIPT NEL CONTESTO WEB

# DOM

Ricordate?

## DOCUMENT OBJECT MODEL (DOM)

Rappresentazione ad albero del documento, **in memoria**

Interfaccia programmatica per:

- Accedere al tree
- Modificare la struttura
- Modificare lo stile
- Modificare il contenuto

*Essentially, it connects web pages to scripts or programming languages.*

# INTERFACCE DOM

- **Node** : quanto serve per la rappresentazione dell'albero
- **Document** : deriva da `node` , ma alcune proprietà non sono applicabili (es: un `Document` non ha attributi).
- Browser:
  - **Window** : finestra (tab) del browser contenente il documento

**Attenzione:** alcuni metodi sono sempre applicati alla finestra (es: `window.resizeTo` )

- **Navigator** : informazioni sul browser
- **History** : scorrere in avanti/indietro la history
- ...

# INTERFACCE DOM - HTML

- `HTMLDocument` : estende `Document`, ma con `HTML5` molti metodi/proprietà sono già presenti in esso.
- `HTMLElement` : di tutti i tipi
  - `HTMLAnchorElement`
  - `HTMLParagraphElement`
  - `HTMLFormElement`
  - ecc.

Non solo HTML! C'è anche tutto il mondo dell'SVG...

# ACCEDERE AGLI ELEMENTI

L'oggetto document contiene il DOM:

- `document.getElementById("...")` → restituisce l'elemento ( `HTMLElement` ) con l'id in argomento
- `document.getElementsByTagName("...")` → gli elementi con il nome tag in argomento ( "a" , "span" , ...); "\*" li restituisce tutti
- `document.getElementsByClassName("...")` → gli elementi con la classe (list di valori separati da spazio) in argomento
- `document.querySelectorAll("...")` → gli elementi che soddisfano il **sele~~ctor~~CSS** in argomento
- `document.querySelector("...")` → il primo elemento che soddisfa il **sele~~ctor~~CSS** in argomento

# CREARE ELEMENTI

- `document.createElement("...")` → crea un elemento del tipo richiesto
- `element.appendChild(child)` → aggiunge l'elemento come ultimo children del nodo element specificato
- `.innerHTML` → legge/imposta il codice HTML che scrive i descendants dell'elemento

```
1 var mainSection = document.getElementById("mainSection");
2 var newP = document.createElement("p");
3 newP.innerHTML = "questo è il <b>nuovo</b> paragrafo";
4 mainSection.appendChild(newP);
```

# STILE DEGLI ELEMENTI

- `element.style` → accede all'attributo `style` dell'elemento → ignora gli stili ereditati/esterni

```
1 element.style.color = "#ff3300";
2 element.style.marginTop = "30px";
```

- `window.getComputedStyle(element)` → restituisce lo stile computato per l'elemento

# DOM EVENT MODEL

Ci sono diversi metodi per legarsi agli **eventi**:

## 1. HTML Attribute

```
<button onclick="alert('Hello world!')">
```

**Brutto:**

- Markup più corposo → meno leggibile
- Contenuto/comportamento non separati → bug più difficili da trovare

# DOM EVENT MODEL

Ci sono diversi metodi per legarsi agli [eventi](#):

1. HTML Attribute
2. EventTarget.addEventListener ([specifiche](#))

```
1 /* Assumiamo che myButton sia un Button*/
2 myButton.addEventListener('click', function() {
3   alert('Hello world');
4 }, false);
```

Bello!

**Attenzione:** non supportato in Internet Explorer 6-8, in cui esiste  
EventTarget.attachEvent → conviene usare librerie per massima  
compatibilità

# DOM EVENT MODEL

Ci sono diversi metodi per legarsi agli **eventi**:

1. HTML Attribute
2. EventTarget.addEventListener (**specifiche**)
3. DOM element properties

```
1 /* Assumiamo che myButton sia un Button*/
2 myButton.onclick = function(event) {
3   alert('Hello world');
4 }
```

**Attenzione:** un solo *handler* per elemento e per evento.

# PROPAGAZIONE EVENTI

Ci sono **tre fasi**:

- 1. Capture:** l'evento si propaga dall'evento più ancestrale (Window) quello più dettagliato (es: td )
- 2. Target:** l'evento arriva a destinazione, ed in base al tipo di evento ferma o prosegue con il bubble
- 3. Bubble:** l'evento si propaga a ritroso nell'albero

Bloccare manualmente la propagazione bubble:

```
ev.stopPropagation();
```

Attenzione, non interrompe l'esecuzione della funzione.

# ESERCIZIO JS-ANAGRAMMI

Scrivere un documento HTML con una casella di input e un bottone: quando l'utente clicca sul bottone, la pagina mostra un elenco con le prime 20 permutazioni della parola scritta nella casella

Spunti:

- casella di input → `input`
- "quando l'utente" → evento

# **ECMASCRIPT 2015**

## **A.K.A. ES6**

# QUALCHE DETTAGLIO DI ES6

Alcune cose le abbiamo già viste:

- `const` , da usarsi come predefinito
- `let` , da usarsi se il valore può cambiare
- valore predefinito parametri `function a(par1 = 1){...}`

# CONCATENAZIONE

Old style:

```
1 var nome = "Luca";
2 console.log("Ciao " + nome + ", come va?");
```

ES6: backtick!

```
1 var nome = "Luca";
2 console.log(`Ciao ${nome}, come va?`)
```

# PROPRIETÀ IMPLICITE

Old style:

```
1 function creaProdotto(prezzo, quantita) {  
2     return {  
3         prezzo: prezzo,  
4         quantita: quantita  
5     };  
6 };
```

ES6: posso omettere i nomi

```
1 function creaProdotto(prezzo, quantita) {  
2     return {  
3         prezzo,  
4         quantita  
5     };  
6 };
```

# OBJECT DECONSTRUCTION

Old style:

```
1 var prodotto = {  
2     descr: "computer",  
3     prezzo: 10  
4 };  
5 var prezzo = prodotto.prezzo;  
6 var descr = prodotto.descr;
```

ES6: {}

```
1 const prodotto = {  
2     descr: "computer",  
3     prezzo: 10  
4 };  
5 const {prezzo, descr} = prodotto;
```

# ARROW FUNCTIONS

Funzioni anonime con alcuni vantaggi:

- scrittura più compatta
- non crea un proprio `this`

Old style:

```
1 var x = function(parametri){...}
```

ES6: =>

```
1 const x = (parametri) => { ... }
```

- `( )` sono opzionali se ho un solo parametro (non se ne ho 0 o più)
- Se esegue un solo statement, `{}` opzionali (return implicito)

# CLASSI

Sappiamo usare la constructor invocation...

```
1 function Persona(nome, anni) {  
2     this.nome = nome;  
3     this.anni = anni;  
4 };  
5  
6 var luca = new Persona("Luca", 21);
```

...e anche come funziona l'ereditarietà (scomoda)

# CLASSI

## Soluzione ES6: class

```
1 class XYZ{  
2     constructor(par1, ...) {  
3         this.par1 = par1;  
4     }  
5 }
```

Attenzione:

- in realtà è una funzione con un costrutto literal particolare
- NON subisce l'*hoisting*

Sintassi alternative:

```
1 var XYZ = class {...}  
2 var XYZ = class XYZ{...}
```

# CLASSI

## Costruttore

→ metodo speciale valido solo in `class` che crea una istanza della classe

- un costruttore non può essere getter, setter, async
- una classe può avere solo un costruttore!

# CLASSI

## Ereditarietà

```
1 class X extends Y{  
2     constructor(par1, par2, ...) {  
3         super(par1);  
4         ...  
5     }  
6 }
```

Voilà!!

# CLASSI

## Ereditarietà

Posso estendere qualsiasi funzione sia un costruttore e abbia la proprietà `prototype`

```
1 function AllaVecchia() {
2   this.someProperty = 1;
3 }
4 AllaVecchia.prototype.someMethod = function () { };
5
6 class Moderna extends AllaVecchia { }
7
8 class Moderna {
9   someProperty = 1;
10  someMethod() { }
11 }
12
13 class AltraClasse extends Moderna { }
```

# CLASSI

## Public fields

```
1 class ClasseConCampi {  
2     instanceField;  
3     instanceFieldWithInitializer = "instance field";  
4     static staticField;  
5     static staticFieldWithInitializer = "static field";  
6  
7     constructor(instanceField) {  
8         this.instanceField = instanceField;  
9     }  
10 }
```

# CLASSI

## Private fields

Vengono dichiarati usando il prefisso #

```
1 class ClasseConCampiPrivati {
2     #privateField;
3     #privateFieldWithInitializer = 42;
4
5     #privateMethod() { }
6
7     static #privateStaticField;
8     static #privateStaticFieldWithInitializer = 42;
9
10    constructor(x) {
11        this.#privateField = x;
12    }
13 }
```

# PROMISE

**PREMESSA: `setTimeout()`,  
`setInterval()`**

- Prendono due parametri, code e delay (opzionale)
- Eseguono code dopo delay millisecondi (0 se non specificato)
  - `setInterval()` ripete l'operazione

# PROMISE

## PREMESSA: OPERAZIONI ASINCRONE

```
1 setTimeout(function() {  
2   console.log("Delayed for 5 seconds.");  
3 }, 5000);  
4 console.log("Hello!");
```

- `setTimeout()` è asincrona

# PROMISE

## PREMESSA: OPERAZIONI ASINCRONE

```
1 setTimeout(function() {  
2   console.log("Delayed for 5 seconds.");  
3 }, 5000);  
4 console.log("Hello!");
```

- `setTimeout()` è asincrona
- Stampa prima "Hello!" e poi, dopo 5 secondi, "Delayed for 5 seconds."

# PROMISE

...ovvero, gestire operazioni asincrone.

*"Quando hai finito di inviare il messaggio dammi una conferma, ma intanto lasciami lavorare..."*

Prima: design pattern che risolva il problema

```
1 function inviaMessaggio(msg, callback) {  
2     function doAsync{ /*operazione asincrona*/  
3         boolean success = contactServerAndSendData(msg);  
4         callback(success);  
5     };  
6     doAsync();  
7 }  
8  
9 inviaMessaggio("Pippo", function(resultato){  
10    console.log(resultato);  
11 }) ;
```

# PROMISE

...ovvero, gestire operazioni asincrone.

*"Quando hai finito di inviare il messaggio dammi una conferma, ma intanto lasciami lavorare..."*

**Prima:** design pattern che risolva il problema

**ES6: Promise**, oggetto preconfezionato che rappresenta il completamento (o fallimento) di una operazione asincrona.

```
1 var promiseObj = new Promise( tetherFunction );
```

*TetherFunction:* funzione eseguita dal costruttore con due parametri

- *resolutionFunc* → funzione da eseguire se va tutto bene
- *rejectFunc* → funzione da eseguire se ci sono errori

Questa funzione esegue l'operazione asincrona.

# PROMISE

...ovvero, gestire operazioni asincrone.

*"Quando hai finito di inviare il messaggio dammi una conferma, ma intanto lasciami lavorare..."*

**Prima:** design pattern che risolva il problema

**ES6: Promise**, oggetto preconfezionato che rappresenta il completamento (o fallimento) di una operazione asincrona.

```
1 var promiseObj = new Promise( tetherFunction );
2 promiseObj.then(
3   /*funzione eseguita se tutto ok*/
4 ).catch(
5   /*funzione eseguita se errore*/
6 );
```

# PROMISE

- **Esecutore:** una funzione che fa delle operazioni che richiedono del tempo
- **Consumatore:** una funzione che necessita del risultato del produttore

# PROMISE: ESECUTORE

L'esecuotre (tetherFunction) prende due parametri, due callback chiamate in caso di successo o fallimento dell'operazione

- `resolve(value)` se la funzione termina correttamente, con risultato `value`
- `reject(error)` se si verifica un errore dove `error` è l'errore

```
1 let promise = new Promise(function(resolve, reject) {  
2   /* esecutore: operazione che impiegherà del tempo */  
3   resolve("Fatto!"); /* termina e restituisce "Fatto!" */  
4 });
```

# PROMISE: CONSUMATORE

L'esecutore non viene eseguito finchè non lo invoco esplicitamente. Per farlo devo usare un consumatore.

- `then()` : prende due parametri, due funzioni da eseguire se la promise ha successo oppure no
- `catch()` : prende un parametro, una funzione da eseguire per catturare gli errori

```
1 let promise = new Promise(function(resolve, reject) {  
2   /* esecutore: operazione che impiegherà del tempo */  
3   resolve("Fatto!"); /* termina e restituisce "Fatto!" */  
4 });  
5  
6 promise.then(  
7   function(result){ console.log(result) },  
8   function(error){ console.log(error) }/* opzionale */  
9 );
```

# PROMISE: CONSUMATORE

Possiamo riscrivere tutto con le arrow function

```
1 let promise = new Promise((resolve, reject) => {
2   /* esecutore: operazione che impiegherà del tempo */
3   resolve("Fatto!"); /* termina e restituisce "Fatto!" */
4 );
5
6 promise.then(
7   result => console.log(result),
8   error => console.log(error) /* opzionale */
9 );
```

# PROMISE: ASINCRONE

```
1 let promise = new Promise(function(resolve, reject) {  
2   setTimeout(() => resolve("Fatto!"), 2000)  
3 } );  
4  
5 promise.then(result => console.log(result));  
6 console.log("Ciao");
```

Stampa prima "Ciao" e poi "Fatto"

# PROMISE: CONCATENARE

```
1 function setTimeoutPromise(delay) {
2     return new Promise((resolve, reject) => {
3         if (delay < 0) return reject("No ritardo")
4         setTimeout(() => {
5             resolve(`Hai atteso ${delay} millisecondi`);
6         }, delay);
7     })
8 }
9 setTimeoutPromise(250).then(msg => {
10     console.log(msg);
11     return setTimeoutPromise(500);
12 }).then(msg => {
13     console.log(msg);
14 }) ;
```

# MODULI

Tanto tanto tempo fa, tutto funzionava bene:

- Pagine web essenziali (poco CSS)
- Ogni tanto un po' di JavaScript

# MODULI

Tanto tanto tempo fa, tutto funzionava bene:

- Pagine web essenziali (poco CSS)
- Ogni tanto un po' di JavaScript

Ora:

- JavaScript ovunque
- JavaScript corposo
- Ingestibile con un singolo file .JS (modifiche concorrenti?)

*"Sarebbe bello spezzare il codice in tanti file facili da mantenere..."*

# MODULI

Immagino abbiate capito come si chiama la soluzione



Il **supporto nei browsers** è discreto, se facciamo finta  
che IE non esista...

Due parole chiave:

- `export` → esporta qualcosa dal file JS
- `import` → importa qualcosa nel file JS

# MODULI - ESPORTARE

## Opzione 1: in fondo al file

- `export default cosa;` → esporta "cosa" come predefinito
- `export {pippo, pluto};` → esporta "pippo" e "pluto".

## Opzione 2: in linea (più bello)

```
1 export default class Cosa { ... }
2 export function pippo() { ... }
3 export function pluto() { ... }
```

**Nota:** Posso esportare solo una cosa come default per file

# MODULI - IMPORTARE

Importiamo l'export di default:

```
1 import Cosa from './modulo.js';
2 let x = new Cosa();
```

Importiamo anche il resto:

```
1 import Cosa, {pippo, pluto} from './modulo.js';
2 let x = new X();
```

Alias per quanto non di default:

```
1 import {pippo as goofy} from './modulo.js';
```

**Nota:** Con l'alias posso risolvere i conflitti di nomi tra moduli diversi

# TO MODULE OR NOT TO MODULE?

Quando includiamo uno script in HTML, possiamo usare degli **attributi**:

- defer → esegue dopo aver caricato il DOM
- nomodule → lo script viene ignorato se il browser supporta i moduli (ES6) → versione per browser tardi
- type="module" → eseguito solo se il browser supporta i moduli:
  - attiva il supporto ai moduli
  - defer implicito
  - se aggiungo l'attributo async viene eseguito subito

# ASYNCHRONOUS JAVASCRIPT

## JAVASCRIPT

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

- Single-thread?
- Asynchronous?

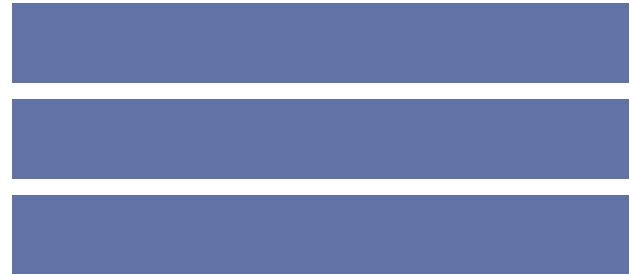
# SYNCHRONOUS JAVASCRIPT EXECUTION STACK

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

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

# FUNCTION EXECUTION STACK: ESEMPIO

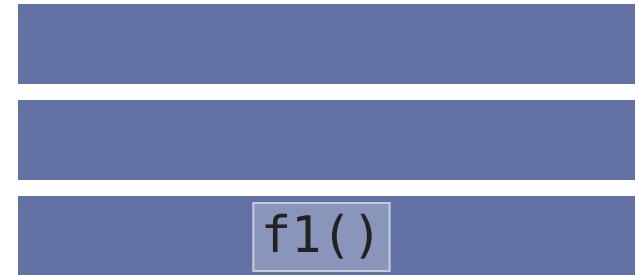
```
1 function f1 () {  
2     ...  
3 }  
4 function f2 () {  
5     ...  
6 }  
7 function f3 () {  
8     ...  
9 }  
10 f1 () ;  
11 f2 () ;  
12 f3 () ;
```



Function Execution  
Stack

# FUNCTION EXECUTION STACK: ESEMPIO

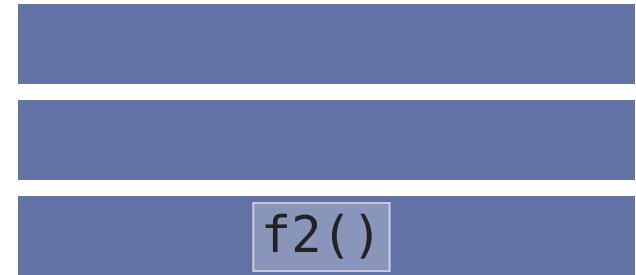
```
1 function f1 () {  
2   ...  
3 }  
4 function f2 () {  
5   ...  
6 }  
7 function f3 () {  
8   ...  
9 }  
10 f1 () ;  
11 f2 () ;  
12 f3 () ;
```



Function Execution  
Stack

# FUNCTION EXECUTION STACK: ESEMPIO

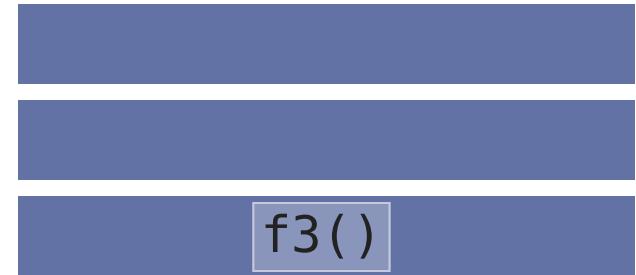
```
1 function f1 () {  
2   ...  
3 }  
4 function f2 () {  
5   ...  
6 }  
7 function f3 () {  
8   ...  
9 }  
10 f1 ();  
11 f2 ();  
12 f3 ();
```



Function Execution  
Stack

# FUNCTION EXECUTION STACK: ESEMPIO

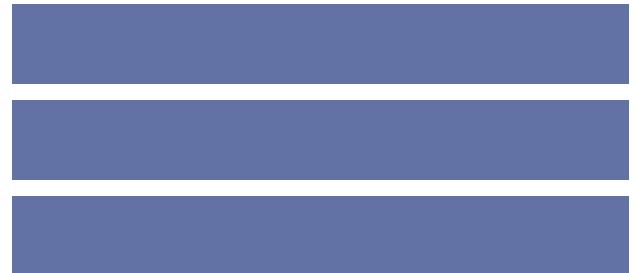
```
1 function f1 () {  
2   ...  
3 }  
4 function f2 () {  
5   ...  
6 }  
7 function f3 () {  
8   ...  
9 }  
10 f1 ();  
11 f2 ();  
12 f3 ();
```



Function Execution  
Stack

# FUNCTION EXECUTION STACK: ESEMPIO

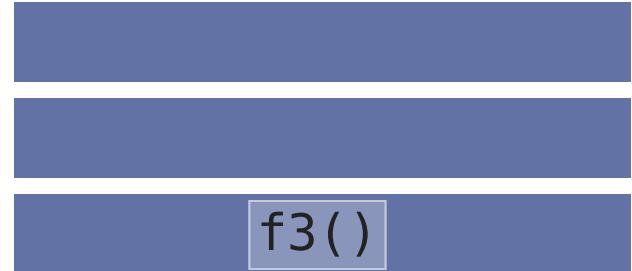
```
1 function f1 () {  
2     ...  
3 }  
4 function f2 () {  
5     f1 () ;  
6 }  
7 function f3 () {  
8     f2 () ;  
9 }  
10 f3 () ;
```



Function Execution  
Stack

# FUNCTION EXECUTION STACK: ESEMPIO

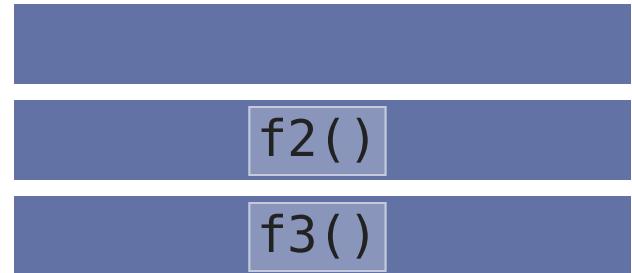
```
1 function f1 () {  
2     ...  
3 }  
4 function f2 () {  
5     f1 () ;  
6 }  
7 function f3 () {  
8     f2 () ;  
9 }  
10 f3 () ;
```



Function Execution  
Stack

# FUNCTION EXECUTION STACK: ESEMPIO

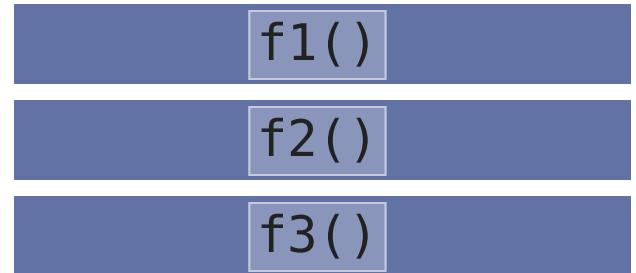
```
1 function f1 () {  
2     ...  
3 }  
4 function f2 () {  
5     f1 ();  
6 }  
7 function f3 () {  
8     f2 ();  
9 }  
10 f3 ();
```



Function Execution  
Stack

# FUNCTION EXECUTION STACK: ESEMPIO

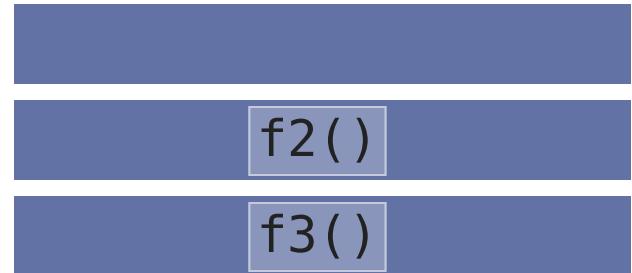
```
1 function f1 () {  
2     ...  
3 }  
4 function f2 () {  
5     f1 () ;  
6 }  
7 function f3 () {  
8     f2 () ;  
9 }  
10 f3 () ;
```



Function Execution  
Stack

# FUNCTION EXECUTION STACK: ESEMPIO

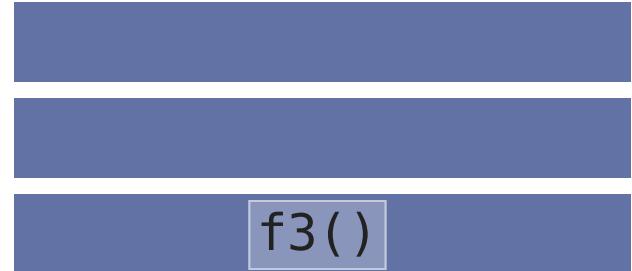
```
1 function f1 () {  
2     ...  
3 }  
4 function f2 () {  
5     f1 () ;  
6 }  
7 function f3 () {  
8     f2 () ;  
9 }  
10 f3 () ;
```



Function Execution  
Stack

# FUNCTION EXECUTION STACK: ESEMPIO

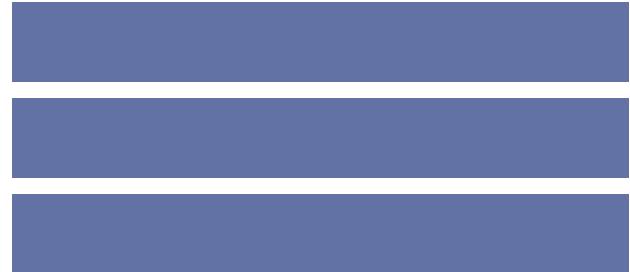
```
1 function f1 () {  
2     ...  
3 }  
4 function f2 () {  
5     f1 () ;  
6 }  
7 function f3 () {  
8     f2 () ;  
9 }  
10 f3 () ;
```



Function Execution  
Stack

# FUNCTION EXECUTION STACK: ESEMPIO

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



Function Execution  
Stack

# ASYNCHRONOUS JAVASCRIPT

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

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

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

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

# BROWSER API/WEB API

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

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

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

In che ordine vengono stampate le due frasi?

- Vengono attesi 2 secondi e poi esegue `stampami()` e quindi `test()` ?
- Oppure viene eseguita `test()` e solo 2 secondi `stampami()` ?

# BROWSER API/WEB API

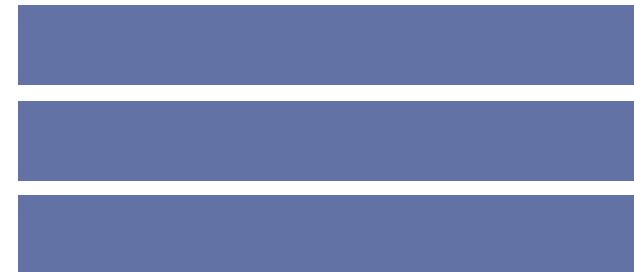
JavaScript ha una coda dedicata per le callback (**Callback Queue**)

Viene creato un ciclo che periodicamente controlla la coda e sposta le callback nello stack (**Event Loop**):

- Le funzioni nello stack vengono eseguite normalmente
- Se viene invocata una API del browser, si aggiunge una callback nella coda (**macrotask**)
- **Se lo stack è vuoto**, viene spostata la callback dalla coda allo stack
- Ricominica da capo

# BROWSER API/WEB API: ESEMPIO

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



Function Execution Stack



Browser API



Callback Queue

# BROWSER API/WEB API: ESEMPIO

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



Function Execution Stack



Browser API



Callback Queue

# BROWSER API/WEB API: ESEMPIO

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

console.log('main')

main()

Function Execution Stack

Browser API

Callback Queue

# BROWSER API/WEB API: ESEMPIO

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



Function Execution Stack



Browser API



Callback Queue

# BROWSER API/WEB API: ESEMPIO

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

setTimeout()

main()

Function Execution Stack

Browser API

Callback Queue

# BROWSER API/WEB API: ESEMPIO

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



Function Execution Stack



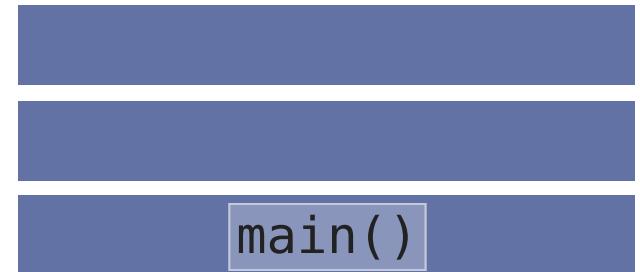
Browser API



Callback Queue

# BROWSER API/WEB API: ESEMPIO

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



Function Execution Stack



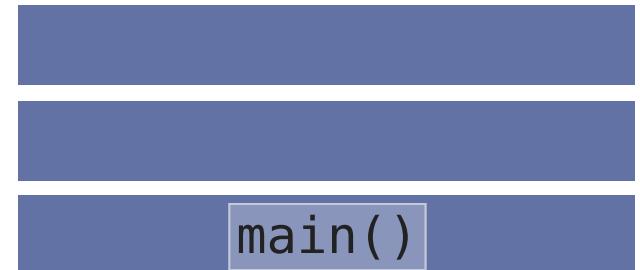
Browser API



Callback Queue

# BROWSER API/WEB API: ESEMPIO

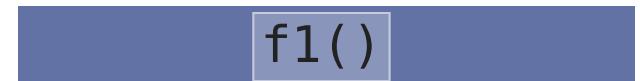
```
function f1() {  
    console.log('f1');  
}  
  
function f2() {  
    console.log('f2');  
}  
  
function main() {  
    console.log('main');  
    setTimeout(f1, 0);  
    f2();  
}  
main();
```



Function Execution Stack



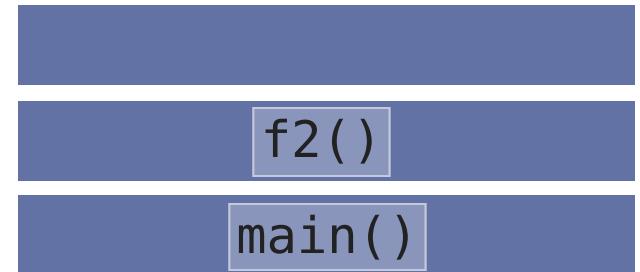
Browser API



Callback Queue

# BROWSER API/WEB API: ESEMPIO

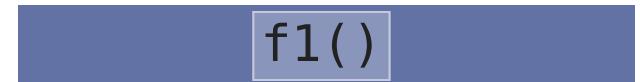
```
function f1() {  
    console.log('f1');  
}  
  
function f2() {  
    console.log('f2');  
}  
  
function main() {  
    console.log('main');  
    setTimeout(f1, 0);  
    f2();  
}  
main();
```



Function Execution Stack



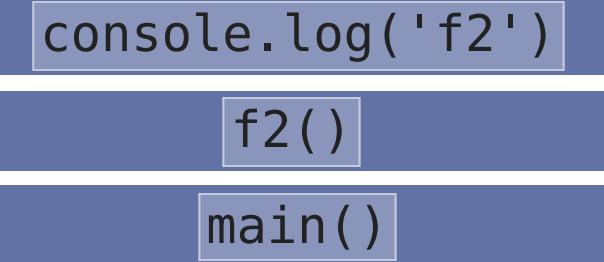
Browser API



Callback Queue

# BROWSER API/WEB API: ESEMPIO

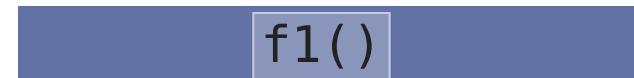
```
function f1() {  
    console.log('f1');  
}  
  
function f2() {  
    console.log('f2');  
}  
  
function main() {  
    console.log('main');  
    setTimeout(f1, 0);  
    f2();  
}  
main();
```



Function Execution Stack



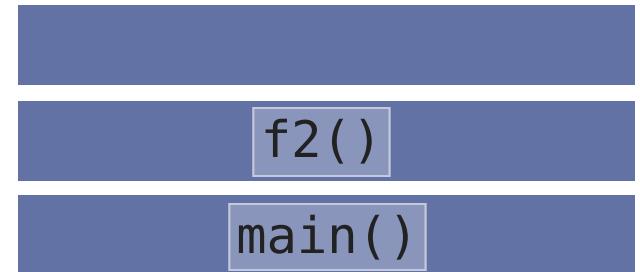
Browser API



Callback Queue

# BROWSER API/WEB API: ESEMPIO

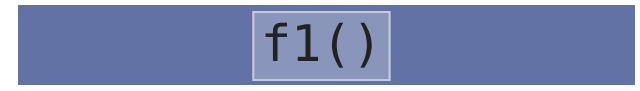
```
function f1() {  
    console.log('f1');  
}  
  
function f2() {  
    console.log('f2');  
}  
  
function main() {  
    console.log('main');  
    setTimeout(f1, 0);  
    f2();  
}  
main();
```



Function Execution Stack



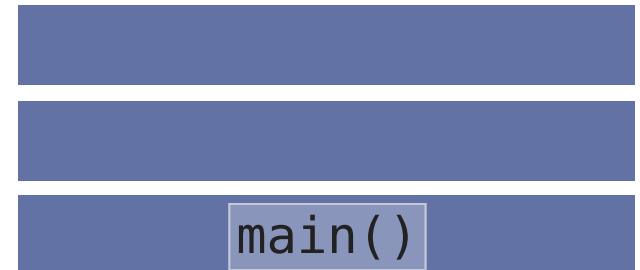
Browser API



Callback Queue

# BROWSER API/WEB API: ESEMPIO

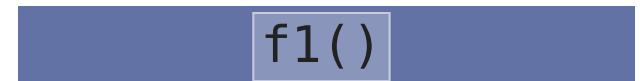
```
function f1() {  
    console.log('f1');  
}  
  
function f2() {  
    console.log('f2');  
}  
  
function main() {  
    console.log('main');  
    setTimeout(f1, 0);  
    f2();  
}  
main();
```



Function Execution Stack



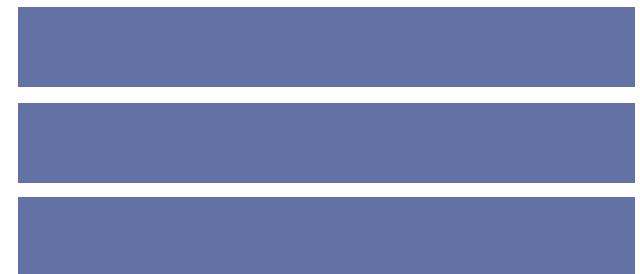
Browser API



Callback Queue

# BROWSER API/WEB API: ESEMPIO

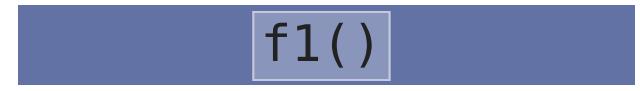
```
function f1() {  
    console.log('f1');  
}  
  
function f2() {  
    console.log('f2');  
}  
  
function main() {  
    console.log('main');  
    setTimeout(f1, 0);  
    f2();  
}  
main();
```



Function Execution Stack



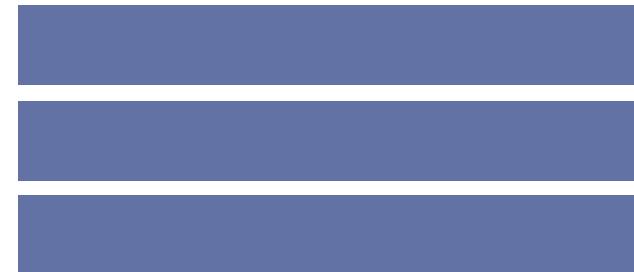
Browser API



Callback Queue

# BROWSER API/WEB API: ESEMPIO

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



Function Execution Stack



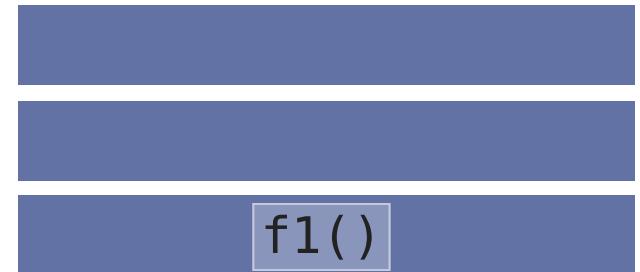
Browser API



Callback Queue

# BROWSER API/WEB API: ESEMPIO

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



Function Execution Stack



Browser API



Callback Queue

# BROWSER API/WEB API: ESEMPIO

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

console.log('f1')

f1()

Function Execution Stack

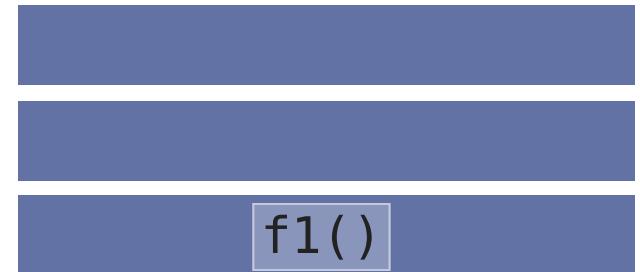
API => f1()

Browser API

Callback Queue

# BROWSER API/WEB API: ESEMPIO

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



Function Execution Stack



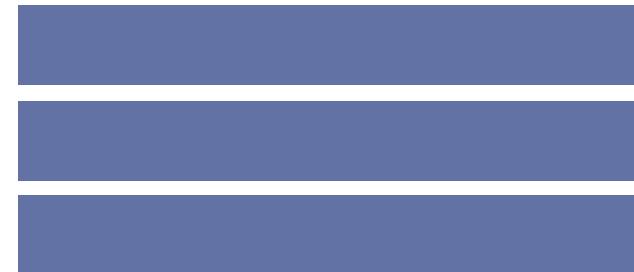
Browser API



Callback Queue

# BROWSER API/WEB API: ESEMPIO

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



Function Execution Stack



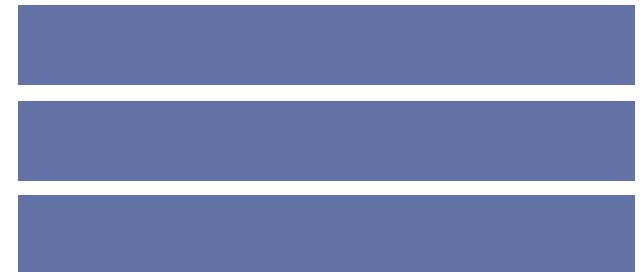
Browser API



Callback Queue

# BROWSER API/WEB API: ESEMPIO

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



Function Execution Stack



Browser API



Callback Queue

# PROMISE

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

# PROMISE: ESEMPIO

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



Function Execution Stack



Browser API



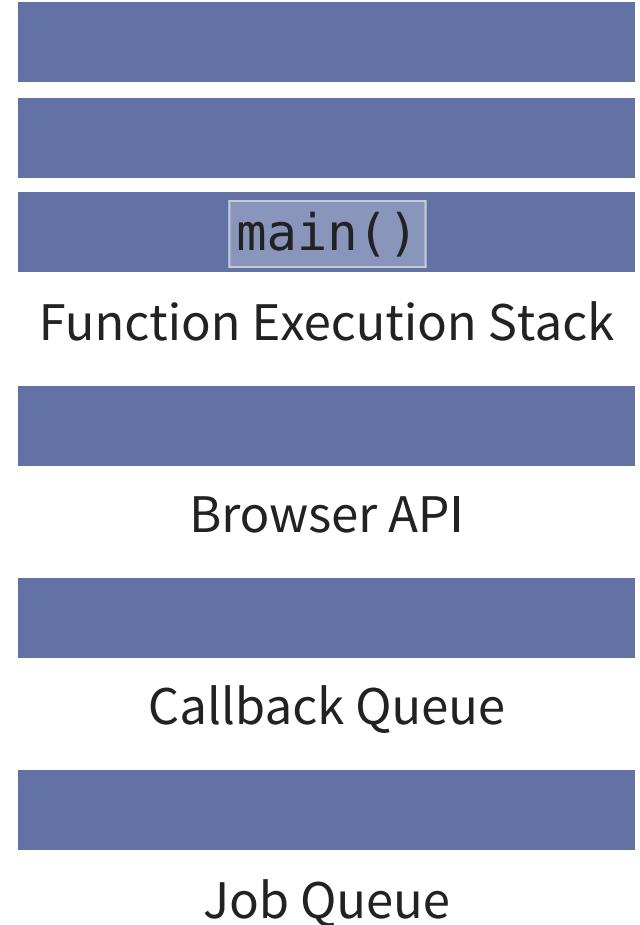
Callback Queue



Job Queue

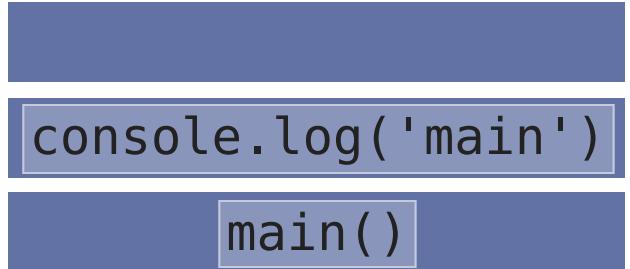
# PROMISE: ESEMPIO

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



# PROMISE: ESEMPIO

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



Function Execution Stack



Browser API



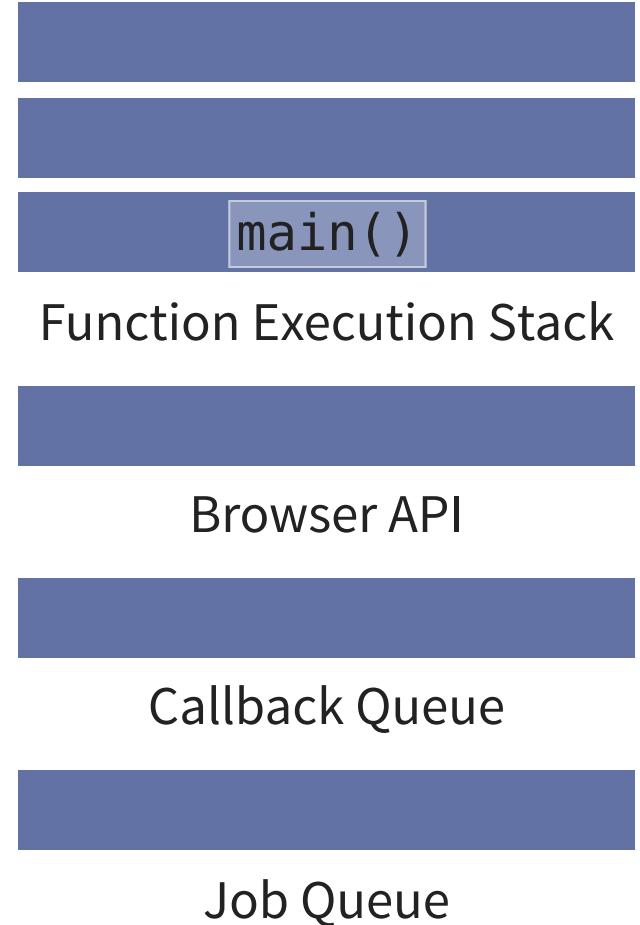
Callback Queue



Job Queue

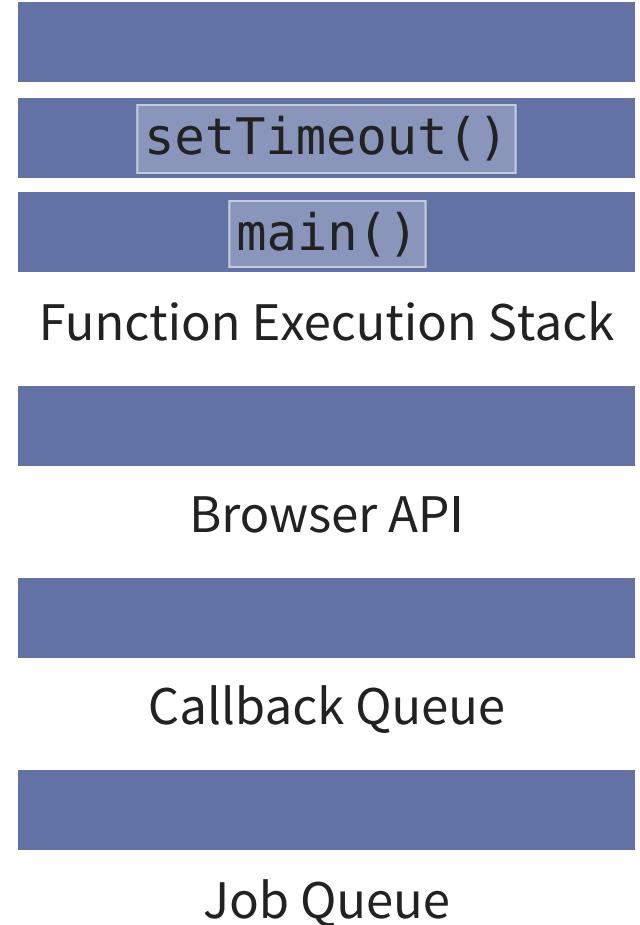
# PROMISE: ESEMPIO

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



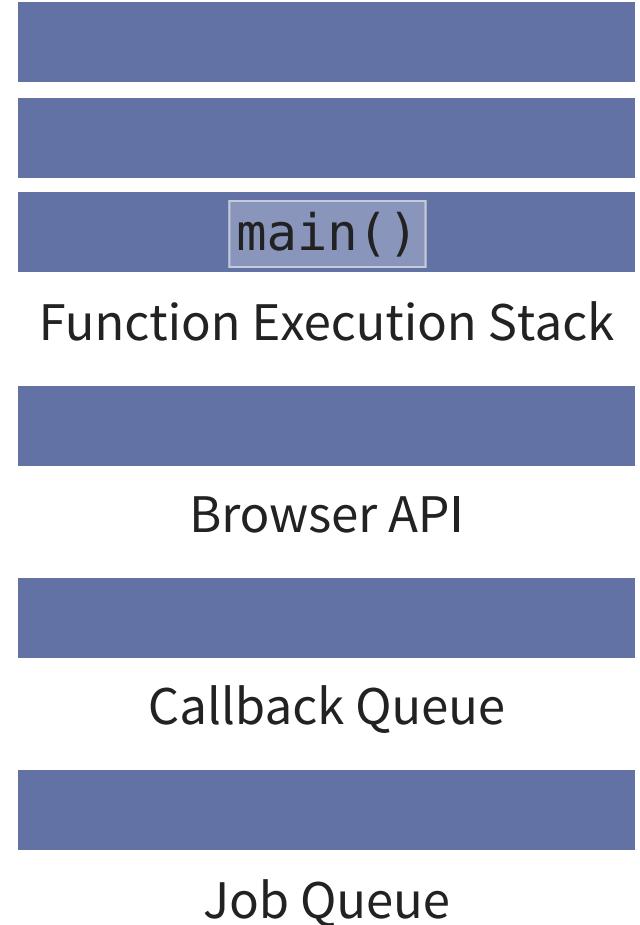
# PROMISE: ESEMPIO

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



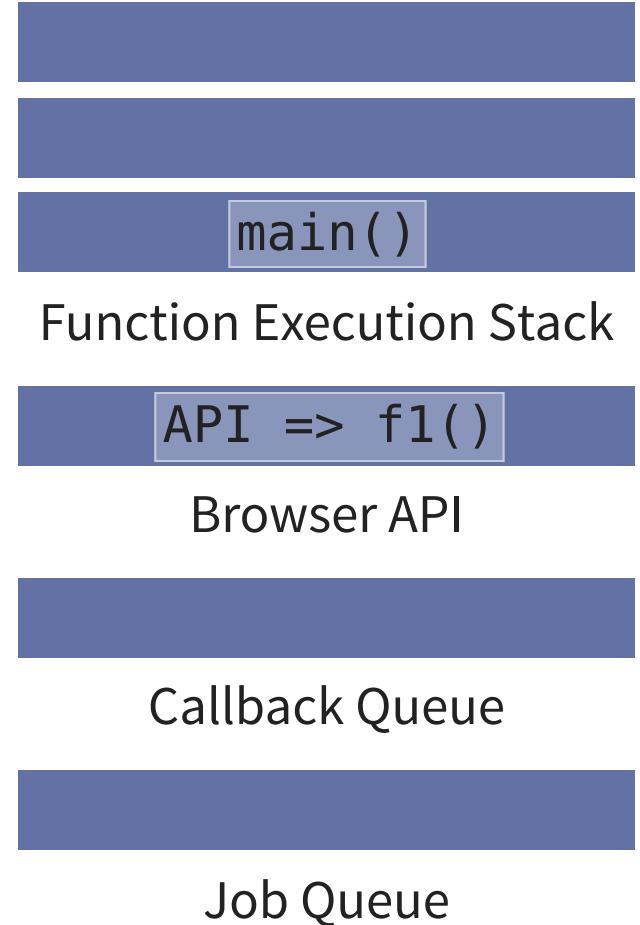
# PROMISE: ESEMPIO

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



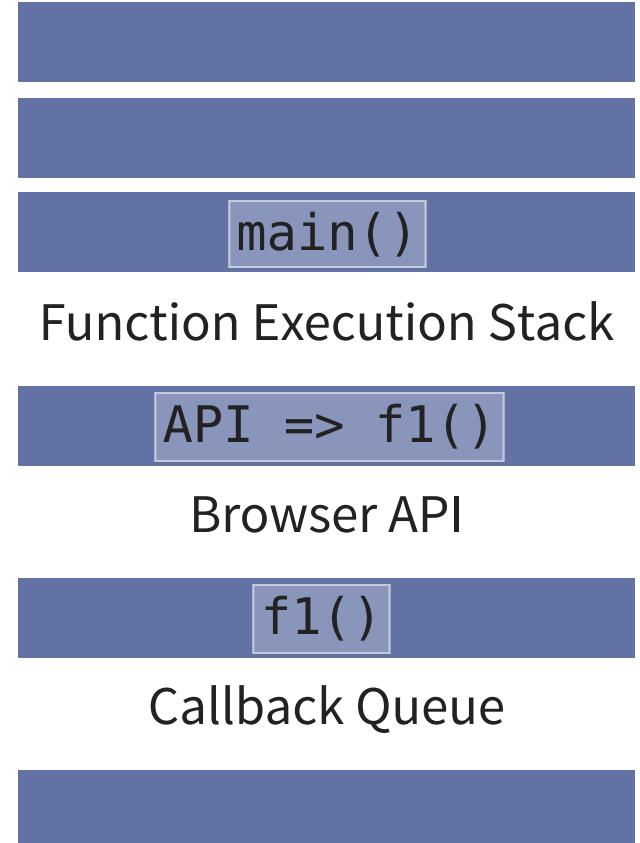
# PROMISE: ESEMPIO

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



# PROMISE: ESEMPIO

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



Function Execution Stack

API => f1()

Browser API

f1()

Callba

ck Queue

# PROMISE: ESEMPIO

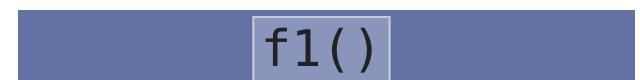
```
function f1() {  
    console.log('f1');  
}  
  
function f2() {  
    console.log('f2');  
}  
  
function main() {  
    console.log('main');  
    setTimeout(f1, 0);  
    new Promise((resolve, reject) =>  
        resolve('promise')  
    ).then(resolve => console.log(resolve))  
    f2(); 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



Function Execution Stack



Browser API



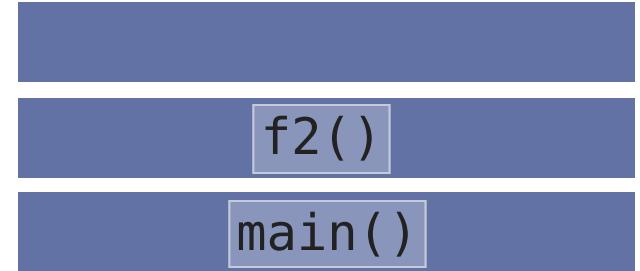
Callback Queue



Job Queue

# PROMISE: ESEMPIO

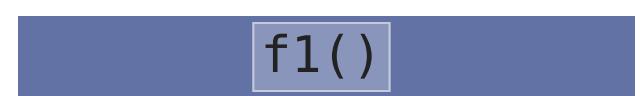
```
function f1() {  
    console.log('f1');  
}  
  
function f2() {  
    console.log('f2');  
}  
  
function main() {  
    console.log('main');  
    setTimeout(f1, 0);  
    new Promise((resolve, reject) =>  
        resolve('promise')  
    ).then(resolve => console.log(resolve))  
    f2(); 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



Function Execution Stack



Browser API



Callback Queue



Job Queue

# PROMISE: ESEMPIO

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

console.log('f2')

f2()

main()

Function Execution Stack

API => f1()

Browser API

f1()

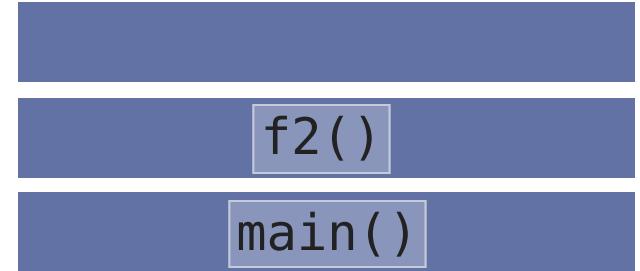
Callback Queue

anonymous

Job Queue

# PROMISE: ESEMPIO

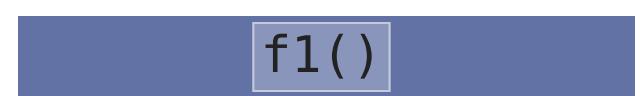
```
function f1() {  
    console.log('f1');  
}  
  
function f2() {  
    console.log('f2');  
}  
  
function main() {  
    console.log('main');  
    setTimeout(f1, 0);  
    new Promise((resolve, reject) =>  
        resolve('promise')  
    ).then(resolve => console.log(resolve))  
    f2(); 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



Function Execution Stack



Browser API



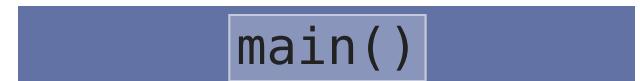
Callback Queue



Job Queue

# PROMISE: ESEMPIO

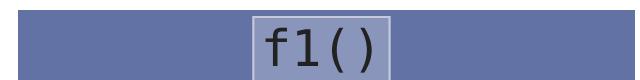
```
function f1() {  
    console.log('f1');  
}  
  
function f2() {  
    console.log('f2');  
}  
  
function main() {  
    console.log('main');  
    setTimeout(f1, 0);  
    new Promise((resolve, reject) =>  
        resolve('promise')  
    ).then(resolve => console.log(resolve))  
    f2(); 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



Function Execution Stack



Browser API



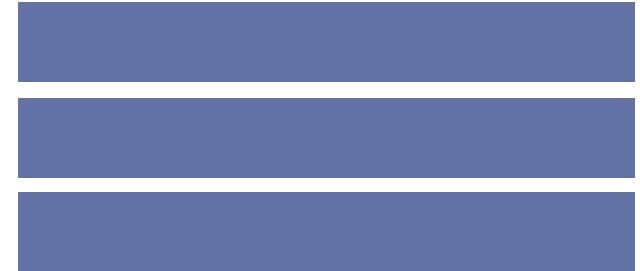
Callback Queue



Job Queue

# PROMISE: ESEMPIO

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



Function Execution Stack

API => f1()

Browser API

f1()

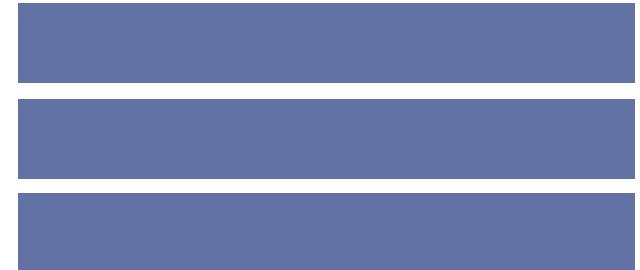
Callback Queue

anonymous

Job Queue

# PROMISE: ESEMPIO

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



Function Execution Stack

API => f1()

Browser API

f1()

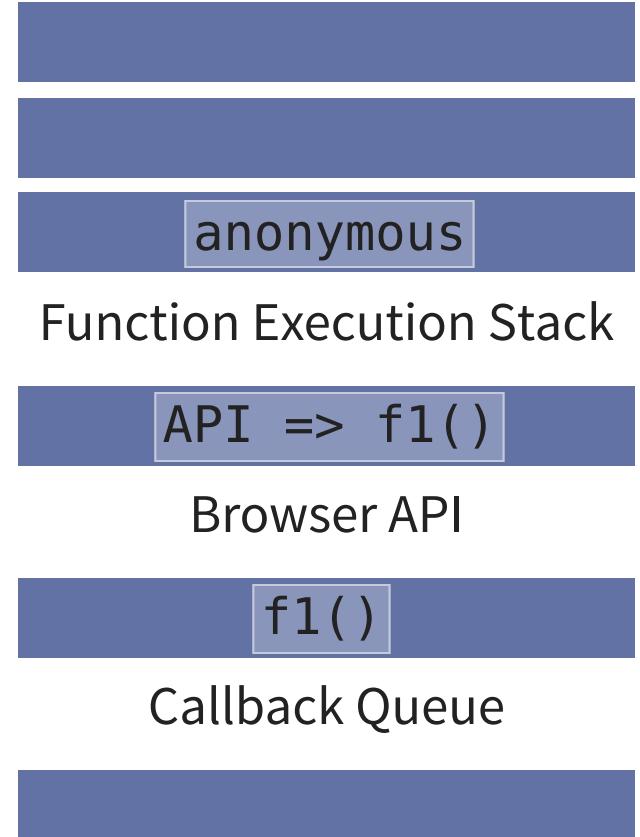
Callback Queue



Job Queue

# PROMISE: ESEMPIO

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



Job Queue

Callback Queue

Function Execution Stack

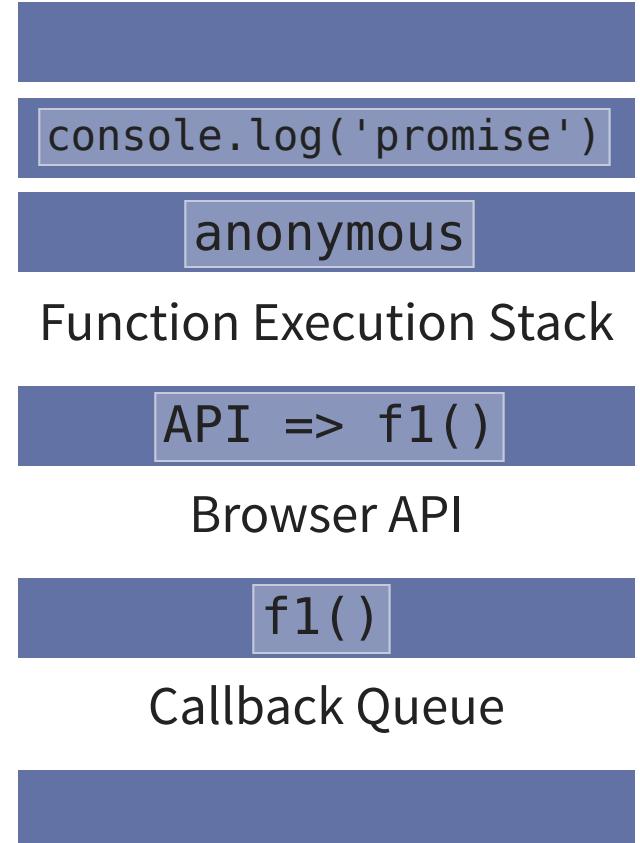
API => f1()

Browser API

f1()

# PROMISE: ESEMPIO

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



Function Execution Stack

API => f1()

Browser API

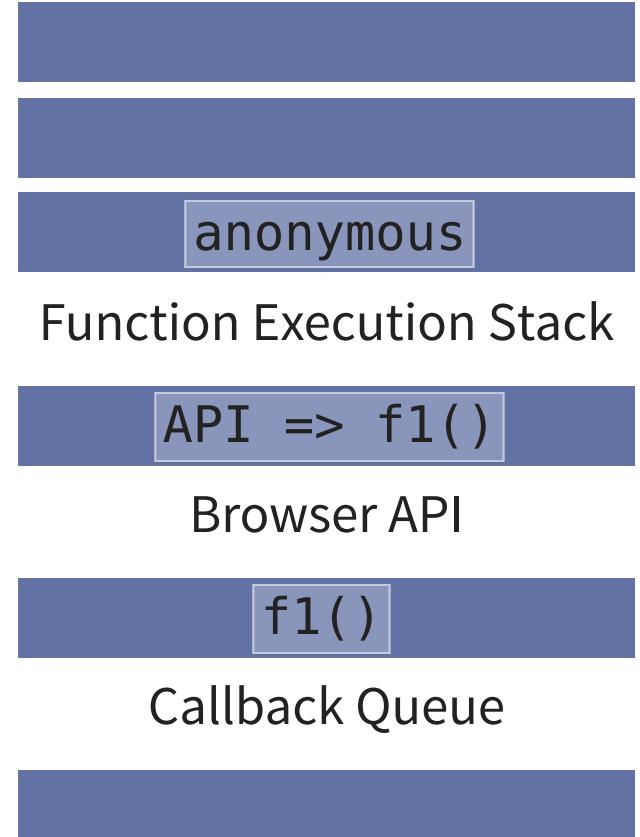
f1()

Callback Queue

Job Queue

# PROMISE: ESEMPIO

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



Job Queue

Callback Queue

Function Execution Stack

API => f1()

Browser API

f1()

# PROMISE: ESEMPIO

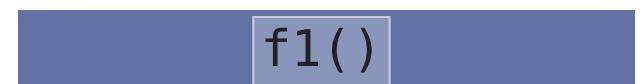
```
function f1() {  
    console.log('f1');  
}  
  
function f2() {  
    console.log('f2');  
}  
  
function main() {  
    console.log('main');  
    setTimeout(f1, 0);  
    new Promise((resolve, reject) =>  
        resolve('promise')  
    ).then(resolve => console.log(resolve))  
    f2(); 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



Function Execution Stack



Browser API



Callback Queue



Job Queue

# PROMISE: ESEMPIO

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



Function Execution Stack

API => f1()

Browser API



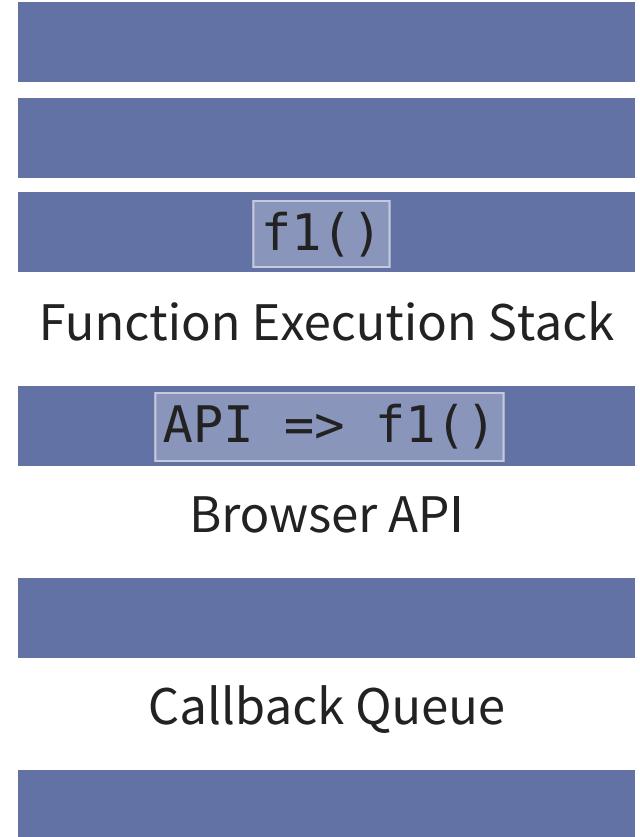
Callback Queue



Job Queue

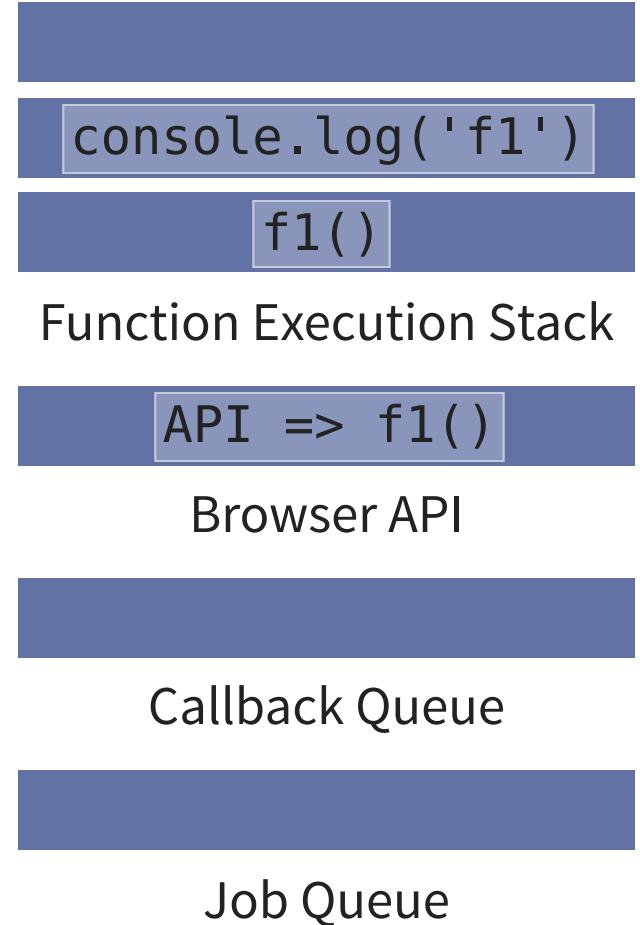
# PROMISE: ESEMPIO

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



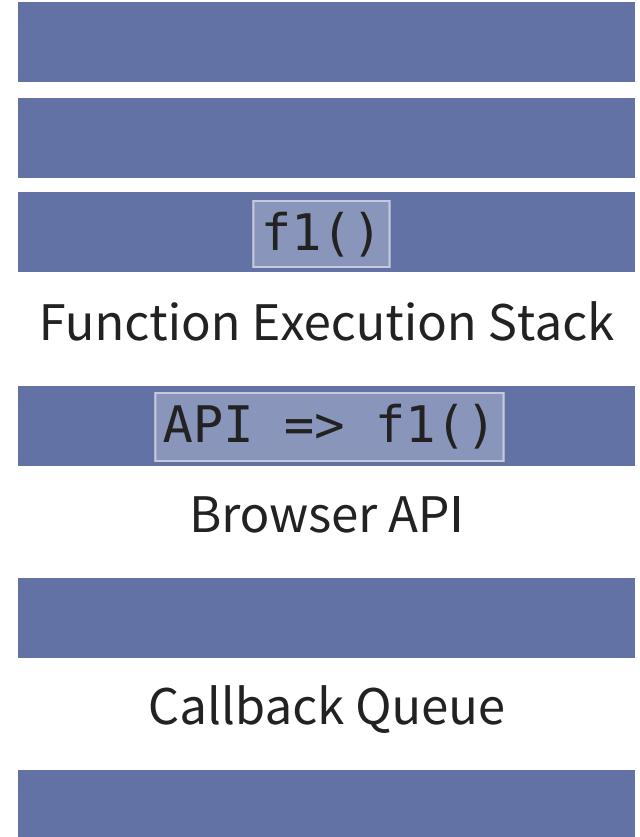
# PROMISE: ESEMPIO

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



# PROMISE: ESEMPIO

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



Function Execution Stack

API => f1()

Browser API

Callback Queue

Job Queue

# PROMISE: ESEMPIO

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



Function Execution Stack

API => f1()

Browser API



Callback Queue



Job Queue

# PROMISE: ESEMPIO

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



Function Execution Stack



Browser API



Callback Queue



Job Queue

# async FUNCTIONS

Con ECMAScript 2017 è stato aggiunto il supporto per scrivere funzioni asincrone, un altro modo per gestire le Promise.

Una funzione asincrona è una funzione che:

- È stata dichiarata con la parola chiave `async`
- Permette l'uso della parola chiave `await` al suo interno
- Restituisce una Promise

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

# async FUNCTIONS

Si può scrivere in forma più elegante

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

Oppure con una arrow function

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

# async FUNCTIONS

Per consumare la Promise posso usare `then()`

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

Oppure la forma abbreviata

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

# await

I vantaggi delle funzione asincrone si vedono quando  
combinate con la parola chiave `await` :

- può essere messa davanti ad una Promise
- può essere messa davanti ad una funzione asincrona
- può essere usata soltanto dentro una funzione asincrona
- mette in pausa il codice finchè la Promise non viene valorizzata,  
quindi restituisce il suo valore di ritorno

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

# ESEMPIO DI FUNZIONE ASINCRONA

```
function dopo2sec() {  
    return new Promise(resolve => {  
        setTimeout(() => {  
            resolve('Fatto!');  
        }, 2000);  
    })  
}  
  
async function asincrona() {  
    console.log('Eseguo e aspetto');  
    const result = await dopo2sec();  
    console.log(result);  
}  
asincrona();
```

# CONCATENARE

```
1 function setTimeoutPromise(delay) {
2   return new Promise((resolve, reject) => {
3     if (delay < 0) return reject("No ritardo")
4     setTimeout(() => {
5       resolve(`Hai atteso ${delay} millisecondi`);
6     }, delay);
7   })
8 }
9
10 asincrona();
11
12 async function asincrona() {
13   const msg1 = await setTimeoutPromise(250)
14   console.log(msg1)
15   const msg2 = await setTimeoutPromise(500)
16   console.log(msg2)
17 }
```

# GESTIONE ERRORI

Con le promise:

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

Con codice asincrono:

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

# JAVASCRIPT

# AJAX

# WEB CLASSICO

Quando si inserisce un URL nel browser:

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

Alcuni URL invocano *programmi* eseguiti dal web server, inviando dei parametri:

`http://www.sailing.org/16937.php?meetid=82`

- **Server:** www.sailing.org
- **Programma:** 16937.php
- **Parametri:** meetid=82

# WEB CLASSICO

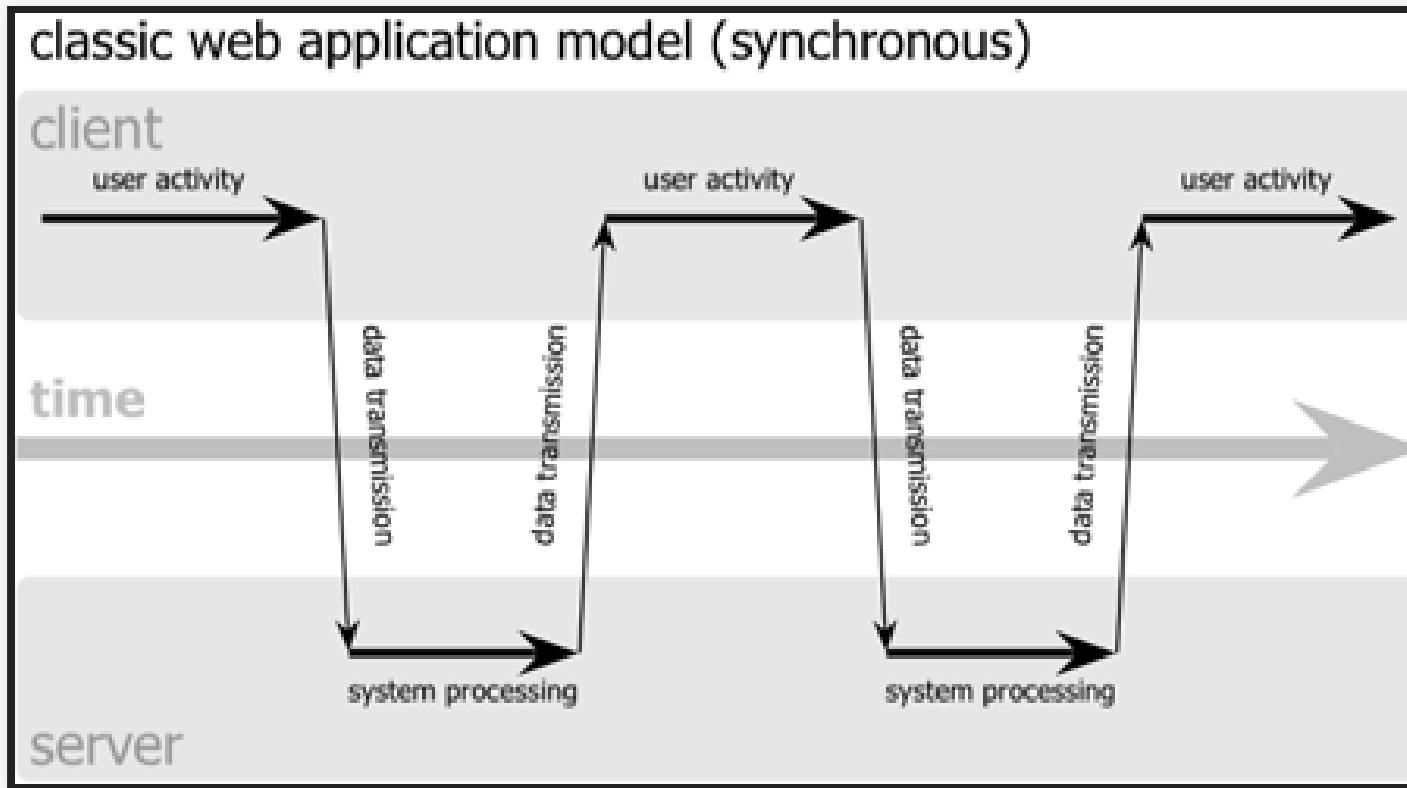


Diagramma di sequenza di un applicazione web senza Ajax ([fonte](#))

# WEB CLASSICO

Cambiare i dati === ricaricare la pagina

- ogni richiesta blocca l'interfaccia grafica fino al termine
- devo ritrasmettere sempre tutto
- user experience scomoda: perdo il riferimento visivo
- validazione spesso solo all'invio (es: username già esistente)

Ma noi vogliamo fare Web Applications, cioè un sito internet che sia molto simile nell'uso ad un'applicazione desktop.

# WEB INTERATTIVO

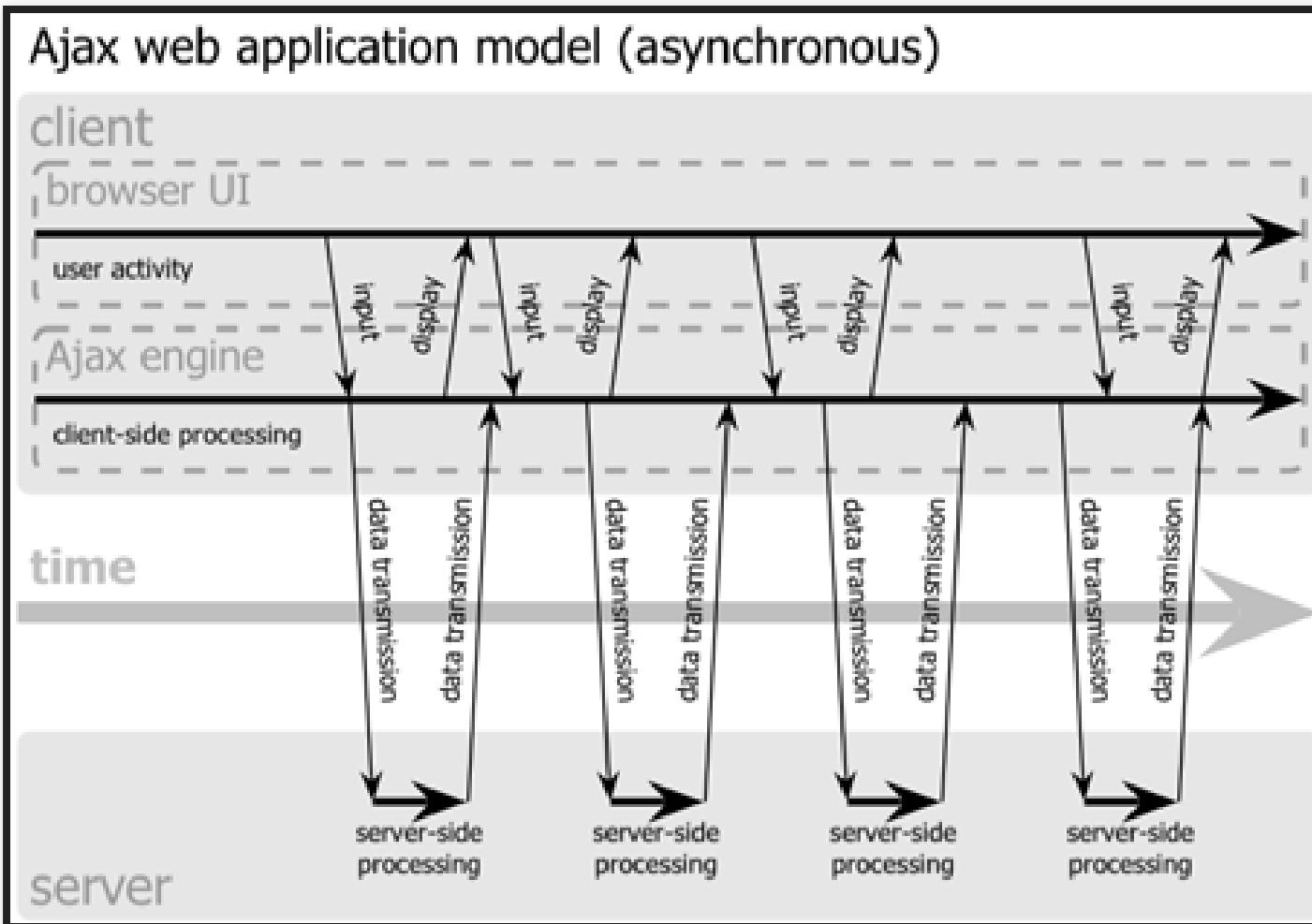


Diagramma di sequenza di un applicazione web con Ajax ([fonte](#))

# AJAX

## AJAX

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

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

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

In pratica: Ajax = utilizzare l'oggetto JavaScript XMLHttpRequest

# AJAX IN PRATICA:

## XMLHttpRequest

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

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

Specifico disponibile su <http://www.w3.org/TR/XMLHttpRequest>,  
documentazione su [w3schools](http://w3schools.com)

*Nota: ad oggi, specifica ≠ implementazioni*

# AJAX IN PRATICA: XMLHttpRequest

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

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

# PREPARARE E INVIARE

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

Bisogna dire all'oggetto XMLHttpRequest :

- url → chi contattare
- myFunction → cosa fare quando riceve la risposta
- il parametro booleano nell' open specifica che la richiesta è asincrona

# LEGGERE LA RISPOSTA

"Quando arriva la risposta"...

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

Di solito ci interessa solo il 4.

# LEGGERE LA RISPOSTA

Quando è disponibile ( `readyState == 4` ) e non ci sono stati errori, la risposta si trova in:

- `xhr.responseText` → corpo della risposta, come stringa
- `xhr.responseXML` → corpo della risposta, come oggetto di tipo **Document**, solo se:
  - il corpo della risposta HTTP era un documento xml valido
  - il MIME type ricevuto è null o di tipo xml (`text/xml`, `application/xml`, ...)

Quale usare? Dipende!

# LEGGERE LA RISPOSTA COME TESTO

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

# LEGGERE LA RISPOSTA COME TESTO

Alcune opzioni:

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

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

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

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

# LEGGERE LA RISPOSTA: XML

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

**Nota:** succinta introduzione su XML su [w3schools](#)

# LEGGERE LA RISPOSTA: XML

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

**Nota:** è possibile **convertire** l'XML in un oggetto.

# LEGGERE LA RISPOSTA: JSON

Ma dobbiamo usare sempre XML per comunicare? NO!

## JAVASCRIPT OBJECT NOTATION

Standard di trasmissione dei dati in un formato leggibile, formato da coppie attributo-valore ed elenchi ordinati di valori.

Deriva dall'Object Literal di JavaScript:

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

- più compatto
- più facile da manipolare in JavaScript

# LEGGERE LA RISPOSTA: JSON

Si occupa di tutto `JSON.parse()`:

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

# **SAME ORIGIN POLICY**

## **SAME ORIGIN POLICY**

**Restrizione** che impedisce ad uno script di comunicare con server diversi da quello da cui origina il documento che lo ospita

Lo script su `www.trieste.it/index.html` non può fare  
`xhr.open("GET", "http://udine.it", true)`

# ESERCIZIO JS-AJAX

Creare (almeno) 3 file (JSON o XML o testo) ognuno con la scheda di un animale. La scheda contiene almeno 3 attributi (ad esempio: nome, colore, ...).

Creare un documento HTML con una casella di testo. Quando l'utente scrive il nome dell'animale il documento si aggiorna con una tabella con i dati dell'animale cercato, se disponibile.

**Nota:** *in Chrome, c'è una restrizione che impedisce di caricare documenti col protocollo "file:///".*

**Nota:** *quando si caricano documenti col protocollo "file://" su FireFox, i valori di xhr.status non sono quelli classici di HTTP*

**Nota:** *Usate i web server embedded in VSCode o Atom*

# fetch()

L'uso di XMLHttpRequest è scomodo:

- pensato nel 2006, vecchie versioni di JavaScript
- è codice asincrono: potrebbe eusare Promise o async
- → funzione `fetch` introdotta di recente

# fetch() : ESEMPIO

Ispirato alla funzione `ajax()` di JQuery, ma:

- la promise non è rigettata se c'è uno status HTTP di errore
- di default non manda cookie ad altri url
- usa gli oggetti `Response` e `Request`

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

# fetch(): ESEMPIO

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

# **WEB SERVER (1)**

# HTTP: CENNI

Hyper Text Transfer Protocol - [RFC 2616](#)

- Trasmissione di informazioni sul WEB
  - Non necessariamente via TCP
- Richiesta/Risposta
- Connessione chiusa il prima possibile
  - Con HTTP/1.1 (1999): una connessione per più richieste
  - Pro: riduco connessioni attive
  - Contro: implicitamente stateless

# HTTP: REQUEST

Tre sezioni per request:

1. Request line

- Metodo
- URI
- Versione del protocollo

2. Header: opzionale?

- Obbligatorio con HTTP/1.1: HOST
- Fine header: <CR><LF>

3. Body

- Non tutte le request lo richiedono

```
GET /index.html HTTP/1.1
Host: www.esempio.it
<CR><LF>
```

# HTTP: REQUEST HEADER

Coppia chiave-valore, una per riga

- Accept : content-type accettabili nella risposta
  - Inoltre: accept-language , accept-encoding (compressione), accept-charset , ...
- User-agent : programma che effettua la richiesta
- Host : nome DNS del server che sto contattando
- Cookies
- Authorization
- Gestione connessione
- Gestione cache, proxy, ...

# HTTP: AUTENTICAZIONE

Gestita nei Request Headers

**Basic:**

1. Richiedo la pagina
2. Risposta: non autorizzato + info per autenticazione Realm = "Credenz Esse3"
3. Il browser chiede all'utente le credenziali
4. Richiedo di nuovo la pagina, inserendo nell'header username:password

Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==

**Attenzione:**

- BASE64 encoding → plain text!!
- Meglio usarla solo con HTTPS

# HTTP: AUTENTICAZIONE

Gestita nei Request Headers

Digest ([RFC-2617](#)):

1. Richiedo la pagina
2. Risposta: non autorizzato + info per autenticazione

WWW-Authenticate: Digest realm="testrealm@host.com",  
nonce="dcd98b7102dd2f0e8b11d0f600bf0c093"

3. Il browser chiede all'utente le credenziali
4. Richiedo di nuovo la pagina, inserendo nell'header

Authorization: Digest username="Mufasa", ...,  
MD5(MD5(username:realm:password):nonce)

# HTTP: LOGOUT

Il browser memorizza le credenziali per non chiederle  
ogni volta: come mi scrollo?

Non previsto!

- ... potrei inviare credenziali volutamente sbagliate (non tutti i browser)
- ... funzione `logout()` in JavaScript (non supportata)

In sintesi: si pone il problema del logout  $\Rightarrow$  usare un  
sistema di autenticazione session-based

# HTTP: REQUEST METHOD

- GET:
  - richiede la rappresentazione di una risorsa
  - non modifica il contenuto
  - body vuoto
- HEAD:
  - identico al GET, ma non ritorna il response body (es: cache)
- POST:
  - aggiunge dati ad una entità esistente → identificata dall'URI
  - crea una risorsa non ancora identificata → se ne occupa l'URI richiamata
  - l'entità da aggiungere è contenuta nel body
- PUT:
  - salva l'entità contenuta nel body nell'URI specificato
  - se esiste già: la sovrascrive ○ altrimenti: ne viene creata una nuova
- PATCH:
  - modifica parziale di una risorsa (RFC-5789)
- DELETE:
  - rimuovere l'entità identificata

# HTTP: RESPONSE

Tre sezioni per request:

## 1. Status line

- 1xx Informational
- 2xx Successful
- 3xx Redirection
- 4xx Client error
- 5xx Server error

## 2. Header

- Server: tipo e versione
- Content-Type
- Set-Cookie
- ..., gestione cache, ETag, ecc.

## 3. Body

# HTTP: RESPONSE

```
HTTP/1.0 200 OK
Date: Mon, 28 Jun 2004 10:47:31 GMT
Server: Apache/1.3.29 (Unix) PHP/4.3.4
X-Powered-By: PHP/4.3.4
Vary: Accept-Encoding,Cookie
Cache-Control: private, s-maxage=0,
max-age=0, must-revalidate
Content-Language: it
Content-Type: text/html; charset=utf-8
Age: 7673
X-Cache: HIT from wikipedia.org
Connection: close
```

# WEB SERVER

Applicazione che gestisce le richieste di trasferimento  
di pagine web tramite il protocollo HTTP o HTTPS

- Apache
- MS ISS
- nginx
- LiteSpeed

# APACHE

- Creato nel 1995
- Nella Apache Foundation dal 1999 (primo software)
- Infiniti tutorial
- Manuale di oltre 500 pagine
- Espandibile con moduli
- File di configurazione
- Installazione tipica LAMP:
  - Linux
  - Apache
  - MySQL
  - PHP

# APACHE: GESTIONE CONCORRENZA

Multi-Processing Modules: quando è nato Apache, non esistevano le connessioni HTTP keep-alive

- Process (pre-fork):
  - Processi single-thread, uno per request
  - Molto veloce
  - Consuma molta RAM
  - Unica soluzione in certi casi (es: il modulo PHP non è thread safe)
- Worker:
  - Un processo controllore, che istanzia altri processi
  - Ogni child-process può gestire più thread, un thread per request
  - Problema: ho un solo controllore...
- Event:
  - Simile al worker, ma ottimizzato per le connessioni keep-alive
  - Thread dedicati per gestire le richieste keep-alive inattive
  - Default da Apache 2.4

# NGINIX

- Creato nel 2004
- Scopo: risolvere il problema [C10K](#)
- Architettura asincrona event driven
  - Worker processes (di solito, uno per CPU core)
  - Il worker gestisce N socket
  - Ogni socket può essere in Listen (attendo qualcuno) o Connection (sono connessi qualcuno)
  - Gli eventi accadono sui socket ed il worker li gestisce:
    - Se Listen socket: nuova connessione → diventa Connection
    - Se Connection socket: il client ha inviato una richiesta → la gestisco e poi processo gli eventi successivi
  - Se il client invia una successiva richiesta sulla stessa connessione:
    - È un nuovo evento
    - Il worker non deve quindi restare ad aspettare che quel singolo client abbia finito tutte le sue richieste

# CONTENUTI STATICI E DINAMICI

## Apache

- Contenuti Statici: file-based
- Contenuti Dinamici:
  - moduli aggiuntivi, attivabili facilmente (riavvio del processo server)
  - in ogni worker incorpora il language-processor richiesto
  - tutto autoconfigurato

## Nginix

- Contenuti Statici: tutto ok
- Contenuti Dinamici:
  - non gestiti
  - inoltra ad un processo esterno, attende risposta e la inoltra
  - non serve sempre interprete: minore overhead

# CONFIGURAZIONE

## Apache

- Centralizzata (file di testo)
- Distribuita (.htaccess)
  - Per ogni richiesta, controlla tutto il path per i file .htaccess e applica le direttive
  - Vantaggio:
    - posso cambiare la configurazione senza riavviare il server
    - alcuni utenti possono configurarsi da soli parte del server
    - Svantaggio: prestazioni

## Nginix

- Centralizzata:
  - Molto più efficiente
  - Più sicuro

# CONFIGURAZIONE

## Apache

- Nasce come web server → richieste al file system
- Prende la document root, ci aggiunge la request e la cerca nel file system
- Possibile fare altrimenti (Alias, Location)

## Nginix

- Non solo web server, ma anche proxy
- Traduce nel file system solo se necessario
- Controlla il filesystem solo quando è pronto a fornire la risposta

# MODULI

## Apache

- Attivabili/disattivabili dinamicamente
- Nucleo di Apache sempre presente
- Varie funzioni:
  - Scripting (PHP, CGI, etc)
  - URL Rewriting
  - Autenticazione
  - Caching
  - Compressione
  - Proxy
  - ...

## Nginix

- Vanno compilati nel core
- Non utilizzabile il packaging dei distributori di software

# **DEVOPS**

## **(IN A NUTSHELL)**

Insieme di pratiche che combinano lo sviluppo del software (dev) e la gestione delle operazioni di IT (ops)

# SVILUPPO SOFTWARE

- Software va inserito in un contesto:
  - Eseguito in un certo SO
  - Funziona con determinate librerie
  - Potrebbe richiedere un certo framework
- Deve convivere con altri software
- Problema: incompatibilità

# CONTAINER

- I *container* permettono di spostare software da un sistema ad un altro
- Tecnologia abbastanza vecchia, ma alla ribalta dal 2013
- Aiutano lo sviluppatore a costruire applicazioni rapidamente
- Crea un ambiente dedicato, facilmente trasportabile

# CONTAINER

- Ambiente di esecuzione autonomo contenente tutto il contesto necessario
- Pacchetto con:
  - software
  - tutte le dipendenze
  - librerie
  - altri eseguibili
  - configurazione
- Formato standard: App Container Image (ACI) o Open Container Initiative (OCI)

# CONTAINER VS VIRTUAL MACHINE

- VM: contiene sia il software che il SO
- VM: serve un server (hypervisor), che gira su un SO
- Container: c'è un solo sistema operativo
- Container: ogni container ha il suo mount point, ma le risorse sono condivise

# TIPI DI CONTAINER

- **Docker:** software che semplifica creazione e gestione dei container
- **Podman:** come Docker, sviluppato da RedHat, compatibile con Docker

# ESECUZIONE DI CONTAINER

Per eseguire un container servono due cose:  
un'immagine e un Container RunTime

Un Container RunTime è un software che esegue e  
gestisce le componenti di un container

runC è un RunTime molto usato (altro: gvisor o  
kata )

Il container viene eseguito come un processo

# ESECUZIONE DI CONTAINER

Il container viene eseguito come un processo

- Sfrutta i servizi del kernel ospite
- Requisiti di isolamenti e sicurezza: gestiti dal kernel
  - Control Groups
  - Namespaces

# CONTROL GROUPS

- Usati dal kernel per gestire uso di risorse dai processi
- Organizzati in modo gerarchico:
  - gestiti come filesystem virtuale ( `/sys/` )
- Usato per limitare risorse utilizzabili da un container

# CONTROL GROUPS

Limitare la memoria utilizzabile dal processo con PID  
1102 a 50MB (52428800 bytes)

```
$ mkdir /sys/fs/cgroup/memory/gruppo
$ echo 52428800 > /sys/fs/cgroup/memory/gruppo/memory.limit_in_bytes
$ echo 1102 > /sys/fs/cgroup/memory/gruppo/cgroup.procs
```

# NAMESPACES

- Usati dal kernel per isolare i processi
- Se un processo è inserito in un namespace, accede solo alle risorse nel namespace
- Ogni container avrà i suoi namespace

# IMMAGINE

I container si basano su delle **immagini**

- Una immagine è una descrizione di un container
- Docker usa una immagine per eseguire un container
- Ci possono essere più versioni della stessa immagine
  - *repository*
- Ogni versione è identificata con un *tag*
  - *latest*: tag di default per la versione più recente

# UNION FILESYSTEM

- Le immagini sono stratificate in una pila di *layer*
  - Ogni layer è di per se una immagine
  - I layer non hanno un tag associato
- I layer sono **immutabili**
- Più immagini possono codividere lo stesso layer
- Ultimo layer della pila è il *layer container* ed è modificabile

# GESTIONE CONTAINER

- docker ps : elenco dei container
- docker ps -a : elenco di tutti i container, anche non attivi
- docker run --name con\_name : esegue in container con nome con\_name
- docker run -d con\_name : esegue in modalità *detached* (background)
- docker run -it con\_name : modalità interattiva
- docker rm id : rimuove il container con un certo id
- docker start id : esegue il container spento con id id
- docker attach id : si collega al container con id id
- docker stop id : spegne il container con id id

# FILESYSTEM

Il filesystem di ogni container è completamente isolato

- Non posso vedere il filesystem da altri container o dall'host
- *data volume*: ponte tra FS del container e FS dell'host
- mappa un FS locale in uno del container

```
docker run -v /home/andrea:/home/container con_name
```

# NETWORKING

La rete di ogni container è completamente isolata

- Posso mappare una porta tra il container e l'host
- Attenzione: su windows gira come VM, cambia l'ip!

```
docker run -p 8088:80 con_name
```

# CONFIGURAZIONE DOCKER

- Immagine deve essere descritta tramite un file:
  - Dockerfile
  - docker-compose
- Crea il container che verrà puoi eseguito

# DOCKERFILE

Script che contiene le operazioni da fare per  
inizializzare il container

```
FROM ubuntu
RUN apt update
RUN apt install -y apache2
RUN apt install -y apache2-utils
RUN apt clean
EXPOSE 80
CMD ["apache2ctl", "-D", "FOREGROUND"]
```

# DOCKERFILE

- FROM : immagine di base da cui partire
- ENV chiave=valore : imposta variabili di ambiente
- RUN : esegue dei comandi all'interno dell'immagine (crea un nuovo layer)
- ADD : aggiunge un file (anche remoto) all'immagine
- COPY : come ADD , ma non da remoto e più veloce
- ENTRYPOINT : esegue una istruzione appena il container è avviato, non crea layer
- WORKDIR : *working directory* del container
- EXPOSE : porte su cui sarà in ascolto il container (non aperte di default)
- VOLUME : path nel container dove mappare un volume (da fare a runtime)

# BUILD

Per creare un'immagine da un Dockerfile, va fatto il  
build

- docker build -t TAG build\_context
- il *build context* definisce il path in cui trovare il Dockerfile e da cui fare gli eventuali COPY/ADD

# ESEMPIO

Dockerfile:

```
FROM httpd:2.4
EXPOSE 80
COPY ./src/ /usr/local/apache2/htdocs/
```

```
docker build -t our_server:1.0 .
```

```
docker run -d -P our_server:1.0
```

# DOCKER-COMPOSE

Tool di configurazione che permette di comporre un container

```
version: '3.9'
services:
  apache:
    image: httpd:latest
    container_name: my-apache-app
    ports:
      - '8080:80'
    volumes:
      - ./website:/usr/local/apache2/htdocs
```

# **PROGRAMMAZIONE LATO SERVER**

# WEB SERVER

## WEB SERVER

Programma che consegna documenti web *su richiesta*

Caso semplice:

1. utente digita l'url del documento  
`(http://www.units.it/risultati-esami.html)`
2. browser→web server: «caro web server, dammi il documento "risultati-esami.html"»  
`(GET /risultati-esami.html  
HTTP/1.1)`
3. il web server cerca il documento sul disco
4. web server→browser: «ecco il documento HTML»

**Nota:** qualcuno aveva messo su disco il documento

# WEB SERVER

(Esempio di) caso normale:

1. utente vuole comprare l'oggetto  $X$
2. browser  $\rightarrow$  web server: richiesta complessa (contiene un qualche riferimento a  $X$ )
3. il web server esegue un programma
  1.  $X$  è a magazzino?
  2. calcola il prezzo di  $X$  per l'utente
  3. ...
  4. "crea" documento HTML con descrizione risultato operazione
4. web server  $\rightarrow$  browser: «ecco il documento HTML»

# WEB SERVER: ALTRO PUNTO DI VISTA

Il web server è un programma che riceve richieste secondo il protocollo HTTP

Quando riceve la richiesta, esegue operazioni, alcune per produrre il documento HTML:

- trovare il documento sul disco
- generare il documento
- generare il documento e dialogare con altri programmi/calcolatori
- ...

# PROGRAMMA PER GENERARE IL DOCUMENTO

Cosa deve fare il programma nel "generare il documento"? Dipende:

- "compro  $X$ " → documento "hai comprato  $X$ "
- "mi iscrivo all'esame  $E$ " → documento "iscrizione effettuata"
- "quanto vale l'azione  $a$ " → documento "scheda azione  $a$ "
- ...

bisogna riscrivere un web server per ogni situazione?

No!



server-side scripting/programming

# WEB SERVER-SIDE SCRIPTING/PROGRAMMING

- Il web server è modulare/programmabile
- Il programmatore:
  - gli fornisce i moduli (programmi)
  - gli dice "per le richieste di tipo X usa il modulo Y"
- I programmi moduli:
  - eseguono in un contesto
  - devono usare/implementare una certa interfaccia
- Il server modulare/programmabile fa il resto:
  - gestisce le connessioni
  - prepara il contesto
  - eventualmente "sveglia/addormenta" i moduli
  - multithreading, sessioni, ...

# WEB SERVER MODULARE/PROGRAMMABILE: OPZIONI

Ce ne sono tante:

- PHP
- ASP
- Java
- ...

Noi vedremo una soluzione basata su Node.js

# NODE.JS

# NODE.JS

- Ambiente server open-source e free
- Multipiattaforma: (Windows, Linux, Unix, Mac OS X, ...)
- Usa JavaScript lato server
- Si basa su programmazione asincrona

# DOCUMENTI WEB DINAMICI

## LETTURA DI UN FILE DATO UN INPUT

Richiesta gestita da un linguaggio classico (PHP, Java, ASP, ...):

1. Riceve la request ed estrae il parametro di input
2. Invia il task al file system
3. Attende che il file system apra e legga il file
4. Invia la risposta al client
5. Pronto per gestire la prossima richiesta

# DOCUMENTI WEB DINAMICI

## LETTURA DI UN FILE DATO UN INPUT

Richiesta gestita da Node.js:

1. Riceve la request ed estrae il parametro di input
2. Invia il task al file system
3. Pronto per gestire la prossima richiesta
4. Quando il file system ha letto il file invia la risposta al client

# NODE.JS È ASINCRONO

Viene eseguito un singolo thread non bloccante e viene programmato in maniera asincrona

# NODE.JS: POSSIBILITÀ

- Generazione di documenti web dinamici
- Gestione del filesystem sul server
- Operazioni CRUD su database
- Raccolta di dati da form

# FUNZIONAMENTO

- Creo un documento JavaScript
- Lo inizializzo da riga di comando tramite l'eseguibile `node`

```
$ node myfile.js
```

- Quando viene eseguito?
- Di base subito, a noi interessa quando riceviamo una richiesta HTTP

# MODULI

- I moduli sono delle librerie JavaScript: insieme di funzioni importare
- Node.js ha dei moduli di default: **built-in**
- Un modulo si include tramite `require()`

```
const http = require('http');
```

# MODULI

I moduli si possono creare ed includere a piacimento

mymodule.js

```
exports.myFunction = function () {  
  return "something";  
};
```

```
const myf = require('./mymodule');  
console.log(myf.myFunction())
```

# NODE.JS COME WEB SERVER

Modulo principale built-in: http

```
const http = require('http');

http.createServer(function (req, res) {
  res.write('Buongiornissimo!');
  res.end();
}).listen(8080);
```

# MODUOLO http

Ha vari metodi, a noi interessa `createServer()`:

- parametro opzionale `requestListener`:
  - invocato ogni volta viene ricevuta una Request HTTP
  - gestisce la richiesta e permette di scrivere una Response HTTP
  - parametri:
    - `request` : oggetto `IncomingMessage`
    - `response` : oggetto `ServerResponse`
- restituisce un oggetto `HTTPServer`
  - può essere messo in ascolto su una porta

# HEADER RESPONSE

## Scrittura HTTP Response

```
1 const http = require('http');
2
3 http.createServer(function (req, res) {
4     res.writeHead(200, {'Content-Type': 'text/html'});
5     res.write('Buongiornissimo!');
6     res.end();
7 }).listen(8080);
```

# LETTURA REQUEST

Posso leggere la HTTP Request

```
1 const http = require('http');
2
3 http.createServer(function (req, res) {
4     res.writeHead(200, {'Content-Type': 'text/html'});
5     res.write(req.url);
6     res.end();
7 }).listen(8080);
```

# LETTURA DEI PARAMETRI

Posso fare il parsing dei parametri della query string

```
1 const http = require('http');
2 const url = require('url');
3
4 http.createServer(function (req, res) {
5   res.writeHead(200, {'Content-Type': 'text/html'});
6   const q = url.parse(req.url, true).query;
7   res.write(` ${q.name} ${q.surname}`);
8   res.end();
9 }).listen(8080);
```

# GESTIONE FILE SYSTEM

Leggiamo un documento da disco:

- Usiamo il module `fs`
- Di default **asincrono**, ma c'è la versione sincrona

```
1 const http = require('http');
2 const fs = require('fs');
3
4 http.createServer(function (req, res) {
5   fs.readFile('index.html', function(err, data) {
6     res.writeHead(200, { 'Content-Type': 'text/html'});
7     res.write(data);
8     return res.end();
9   });
10 }).listen(8080);
```

# GESTIONE FILE SYSTEM

fs ha diverse funzionalità:

- Creazione/aggiornamento file
  - fs.open()
  - fs.appendFile()
  - fs.writeFile()
- Eliminazione file
  - fs.unlink()
- Rinominazione file
  - fs.rename()

# GESTIONE FILE SYSTEM

```
1 const http = require('http');
2 const fs = require('fs');
3
4 http.createServer( (req, res) => {
5   fs.readFile('index.html', (err, data) => {
6     res.writeHead(200, {'Content-Type': 'text/html'});
7     res.write(data);
8     return res.end();
9   });
10 }).listen(8080);
```

**Nota:** possiamo usare arrow function, promise, async, ...

# EVENTI

- Ogni azione in Node.js è un evento
  - apertura file
  - connessione
  - ...
- Posso lanciare o ascoltare eventi "custom" con il module events

```
const http = require('http');
const events = require('events');
const eventEmitter = new events.EventEmitter();
const myEventHandler = function () {
  console.log('Ho sentito qualcosa!');
}
eventEmitter.on('rumore', myEventHandler);
eventEmitter.emit('rumore');
```

# ESEMPIO: UPLOAD FORM (1)

```
1 const http = require('http');
2
3 http.createServer(function (req, res) {
4   res.writeHead(200, {'Content-Type': 'text/html'});
5   res.write('<form action="fileupload" method="post" enctype="multipart/form-data">');
6   res.write('<input type="file" name="filetoupload"><br>');
7   res.write('<input type="submit">');
8   res.write('</form>');
9   return res.end();
10 }).listen(8080);
```

# ESEMPIO: UPLOAD FORM (2)

```
1 const http = require('http');
2 const formidable = require('formidable');
3
4 http.createServer(function (req, res) {
5   if (req.url == '/fileupload') {
6     const form = new formidable.IncomingForm();
7     form.parse(req, function (err, fields, files) {
8       res.write('File uploaded');
9       res.end();
10    });
11  } else {
12    res.writeHead(200, {'Content-Type': 'text/html'});
13    res.write('<form action="fileupload" method="post" enctype="multipart/form-data">');
14    res.write('<input type="file" name="filetoupload"><br>');
15    res.write('<input type="submit">');
16    res.write('</form>');
17    return res.end();
18  }
19 }).listen(8080);
```

# ESEMPIO: UPLOAD FORM (3)

```
1 const fs = require('fs');
2 /* ... */
3
4 http.createServer(function (req, res) {
5   if (req.url == '/fileupload') {
6     const form = new formidable.IncomingForm();
7     form.parse(req, function (err, fields, files) {
8       const oldpath = files.filetoupload.filepath;
9       const newpath = `${__dirname}/${files.filetoupload.originalfilename}`;
10      fs.copyFile(oldpath, newpath, function (err) {
11        if (err) throw err;
12        res.write('File uploaded and moved!');
13        res.end();
14      });
15    });
16  } else {
17    res.writeHead(200, {'Content-Type': 'text/html'});
18    res.write('...');
19    return res.end();
20  }
}
```

# TEMPLATE HTML

Generare un documento web da zero è scomodo

- Potrei riciclare alcune parti
- Voglio evitare di mettere HTML nel mio codice JS

Soluzione: template

# TEMPLATE HTML

Schema HTML con parti variabili:

- Modifico **solo** le parti variabili a runtime
- Non includo HTML nel mio codice
- Posso usare un modulo che faccia da template engine

Quale **modulo**?

# ESERCIZIO NODE 1

*Creare (almeno) 3 file (JSON o XML o testo) oggetti ognuno con la scheda di un animale. La scheda contiene almeno 3 attributi (ad esempio: nome, colore, ...).*

*Creare un documento HTML con una casella di testo. Quando l'utente scrive il nome dell'animale il documento si aggiorna con una tabella con i dati dell'animale cercato, se disponibile.*

*Fornire la risposta tramite un server scritto in Node.js*

# ESERCIZIO NODE 2

*Scrivere un documento HTML con una casella di input e un bottone: quando l'utente clicca sul bottone, la pagina mostra un elenco con le prime 20 permutazioni della parola scritta nella casella*

Le permutazioni vengono calcolate sul server con Node.js e inviate alla pagina web

# LIMITI DI http

- Gestire richieste a URL diversi
- Difficile gestire i diversi metodi (GET, POST, DELETE, ...)
- Fornire file statici
- Usare dei template

Soluzione?

# LIMITI DI http

- Gestire richieste a URL diversi
- Difficile gestire i diversi metodi (GET, POST, DELETE, ...)
- Fornire file statici
- Usare dei template

Soluzione? Usiamo un framework!

# EXPRESS

Il framework più utilizzato in ambito Node.js è **Express**

- Handler per gestire metodi e URL diversi
- Integra dei template engine
- Permette di gestire risorse statiche
- È facilmente estendibile con una pipeline di *middleware*

# EXPRESS

- Di base Express è molto minimale
- Esistono molti module per gestire i casi d'uso più comuni:
  - cookie
  - sessioni
  - autenticazione
  - header
  - sicurezza
  - ...
- Si basano sul concetto di *middleware*
- Contro: dobbiamo scegliere la libreria giusta

# "OPINIONATED" VS. "UNOPINIONATED"

I framework si dividono in due tipologie:

- Opinionated
  - Descrivono "un modo" per risolvere un problema
  - Sviluppo rapido in un particolare dominio
  - Minore flessibilità
- Unopinionated
  - Non descrivono "un modo" per risolvere un problema
  - Maggiore libertà al programmatore
  - Più codice da scrivere

# "OPINIONATED" VS. "UNOPINIONATED"

I framework si dividono in due tipologie:

- Opinionated
  - Descrivono "un modo" per risolvere un problema
  - Sviluppo rapido in un particolare dominio
  - Minore flessibilità
- Unopinionated
  - Non descrivono "un modo" per risolvere un problema
  - Maggiore libertà al programmatore
  - Più codice da scrivere

Express è unopinionated

# EXPRESS

- Una applicazione web tipicamente:
  - Aspetta una Request HTTP
  - Decide che azione fare a seconda dell'URL richiesto e del metodo usato
  - Se necessario, crea un HTML dinamico
  - Invia la Response HTTP
- Express permette di specificare:
  - Delle funzioni da richiamare a seconda del metodo e dell'URL richiesto
  - Quale template engine utilizzare
  - Dove trovare le risorse statiche e i template
  - Che middleware usare per gestire tutto il resto

# EXPRESS: ESEMPIO

```
1 const express = require("express");
2 const app = express();
3 const port = 3000;
4
5 app.get("/", (req, res) => {
6   res.send("Buongiornissimo");
7 });
8
9 app.listen(port, () => {
10   console.log(`In ascolto sulla porta ${port}!`);
11 }) ;
```

# ROUTING

- Che azione fare in base a URL richiesto e metodo usato?
- Viene definita una callback per ogni opzione interessante
- La callback viene assegnata tramite un metodo dedicato a ogni metodo HTTP
  - get()
  - post()
  - delete()
  - put()
  - all()
- I parametri sono: il pattern della risorsa e la callback

# ROUTING: ESEMPIO (1)

```
1 const express = require("express");
2 const app = express();
3 const port = 3000;
4
5 app.get('/book', (req, res) => { res.send('Leggi libro') });
6 app.post('/book', (req, res) => { res.send('Aggiungi libro') });
7 app.put('/book', (req, res) => { res.send('Modifica libro') });
8
9 app.listen(port, () => {
10   console.log(`In ascolto sulla porta ${port}!`);
11 }) ;
```

# ROUTING: ESEMPIO (2)

Si può compattare

```
1 const express = require("express");
2 const app = express();
3 const port = 3000;
4
5 app.route('/book')
6   .get((req, res) => { res.send('Leggi libro') })
7   .post((req, res) => { res.send('Aggiungi libro') })
8   .put((req, res) => { res.send('Modifica libro') });
9
10 app.listen(port, () => {
11   console.log(`In ascolto sulla porta ${port}!`);
12 }) ;
```

# RESPONSE

- `res.end()` termina la response
- `res.json()` invia un json
- `res.redirect()` forza il client a fare una redirect
- `res.render()` invoca un template engine
- `res.send()` invia una risposta generica
- ... altre ancora

# ROUTING

Spesso è utile raggruppare gli handler di una sezione del sito

## wiki.js

```
1 const express = require("express");
2 const router = express.Router();
3
4 router.get("/", function (req, res) {
5   res.send("Wiki home page");
6 });
7
8 router.get("/about", function (req, res) {
9   res.send("Informazioni");
10});
11
12 module.exports = router;
```

## app.js

```
1 const wiki = require("./wiki.js");
2 /* ... */
3 app.use("/wiki", wiki);
```

# ROUTING PATHS

- Le path di routing possono anche essere dei pattern
- ? : risorsa avrà 0 o 1 ripetizione del carattere/gruppo precedente
- + : risorsa avrà 0 o più ripetizione del carattere/gruppo precedente
- \* : risorsa può avere una stringa arbitraria al posto del simbolo
- ( ) : la stringa contenuta tra parentesi è un gruppo

# ROUTING PARAMETERS

Una parte della risorsa può essere parametrica: *named segment*

Un *named segment* si definisce con : come prefisso

```
1 app.get("/utente/:userId/libro/:bookId", (req, res) => {  
2   console.log(req.params.userId, req.params.bookId);  
3   res.send(req.params);  
4 }) ;
```

# MIDDLEWARE

- Largamente utilizzato nelle applicazioni Express
  - Servire file statici
  - Gestire errori
  - Comprimere le risposte
  - ... molto altro
- Sono uno "stack" di funzioni da eseguire
- Le route function concludono il ciclo request-response HT

# MIDDLEWARE

## Le funzioni middleware

- Sono funzioni che accedono a
  - oggetto request
  - oggetto response
  - la successiva funzione middleware
- Possono
  - Eseguire codice
  - Fare modifiche agli oggetti request e response
  - Terminare il ciclo request-response
  - Chiamare la prossima funzione middleware nello stack con `next()`

# MIDDLEWARE

- Middleware può effetturare qualsiasi operazione
- Le definisco con il metodo `use()`
- Possono **terminare** il ciclo request-response
- Se non lo fanno chiamano il prossimo middleware con `next()`

# MIDDLEWARE

```
1 const express = require("express");
2 const logger = require("morgan");
3 const app = express();
4 app.use(logger("dev"))
```

# MIDDLEWARE

Specifica dell'interfaccia dei vari metodi `get()` , `post()` , `delete()` ,...

```
app.METHOD(path, callback [, callback ...])
```

- `path` → The path for which the middleware function is invoked; can be any
  - A string representing a path
  - A path pattern
  - A regular expression pattern to match paths
  - An array of combinations of any of the above
- `callback` → Callback functions; can be:
  - A middleware function
  - A series of middleware functions (separated by commas)
  - An array of middleware functions
  - A combination of all of the above

# APPLICATION LEVEL MIDDLEWARE

Esegui una funzione per qualsiasi request

```
1 app.use((req, res, next) => {
2   /* fai qualcosa */
3   next();
4 })
```

Esegui una funzione per una certa risorsa

```
1 app.use('/user/:id', (req, res, next) => {
2   /* fai qualcosa */
3   next();
4 }) ;
```

# APPLICATION LEVEL MIDDLEWARE

Esegui una funzione se viene fatta una request GET ad una certa risorsa

```
1 app.get('/user/:id', (req, res, next) => {
2   /* fai qualcosa */
3   next();
4 }) ;
```

Esegui una sequenza di funzioni

```
1 app.use('/user/:id', (req, res, next) => {
2   /* fai qualcosa */
3   next();
4 }, (req, res, next) => {
5   /* fai qualcosa */
6   next();
7 })
```

# FILE STATICI

Un middleware per gestire i contenuti statici è di base  
in express: `express.static`

Utile quando dobbiamo gestire request per immagini,  
CSS, JavaScript e pagine statiche. Se il file non viene  
trovato, viene passata la request al prossimo  
middleware.

```
1 const express = require("express");
2 app.use(express.static("public"));
3 app.use(express.static("media"));
4 app.listen(port, function () {
5   console.log('Esempio file server');
6 }) ;
```

# PARSING DEL BODY

Il body di una richiesta HTTP può contenere diverse informazioni

Esiste un middleware express per fare il parsing del body

- form con parametri → `express.urlencoded()`
- json → `express.json()`

```
1 const express = require("express");
2 app.use(express.urlencoded());
3 app.post("/form", (req, res) => {
4   console.log(req.body.param1)
5 });
6 app.listen(port, function () {
7   console.log('Esempio file server');
8 });
```

# SESSIONI

# HTTP: STATELESS

HTTP è un protocollo di comunicazione stateless

Caso semplice:

1. client C fa richiesta  $req_1$  a web server S che risponde con  $resp_1$
2. client C fa richiesta  $req_2$  a web server S che risponde con  $resp_2$

La seconda richiesta è slegata dalla prima:  $resp_2$  viene prodotto solo sulla base di  $req_2$ !

**Nota:** *in caso contrario si dice stateful*

# HTTP: STATELESS WEB APPLICATIONS?

Esigenza reale (web application):

1.  $C \rightarrow S: req_1, S \rightarrow C: resp_1 = f(req_1)$
2.  $C \rightarrow S: req_2, S \rightarrow C: resp_2 = f(req_2, req_1)$

Da un altro punto di vista:

1.  $req_1 \rightarrow resp_1 = f(req_1, \emptyset), s_1 = g(req_1, \emptyset)$
2.  $req_2 \rightarrow resp_2 = f(req_2, s_1), s_2 = g(req_2, s_1)$

# STATEFUL WEB APPLICATION

1.  $req_1 \rightarrow resp_1 = f(req_1, \emptyset), s_1 = g(req_1, \emptyset)$

2.  $req_2 \rightarrow resp_2 = f(req_2, s_1), s_2 = g(req_2, s_1)$

- $f$  e  $g$  sono la web application (es.: e-commerce)
- $req_1$ : "sono il cliente  $X$ , mostrami la pagina delle cucce per cani"
- $resp_1$ : pagina cucce cani con "ciao cliente  $X$ "
- $s_1$ : <cliente= $X$ ;cart= $\emptyset$ >
- $req_2$ : "compra cuccia per cani  $Y$ "
- $resp_2$ : pagina "bravo  $X$ , hai comprato la cuccia  $Y$ "
- $s_2$ : <cliente= $X$ ;cart=[ $Y$ ]>

# STATO

Da un altro punto di vista:

$$1. s := s_0$$

$$2. req \rightarrow resp = f(req, s), s := g(req, s)$$

$s$  è lo **stato** del dialogo tra  $C$  e  $S$

- potrebbe contenere tantissime cose (tutta la storia dalla prima richiesta di  $C$ , i dati dell'app, ...)
- quando contiene dati di breve termine, si chiama **sessione**

# STATO: SERVER CON PIÙ CLIENT

Il web server  $S$  è multithreaded:

- possono esistere più dialoghi contemporaneamente con più client  $C$  differenti
- quando arriva una richiesta  $req$  da  $C_i$  come fa  $S$  a usare lo stato giusto tra quelli  $s_j$ ?

# STATO: SERVER CON PIÙ CLIENT

Risposta sbagliata:

$S$  sa chi è  $C_i$  e "ripesca"  $s_i$  da una lista/mappa

- Cosa sa  $S$  di  $C$  quando arriva  $req$ ? (in altre parole: cosa contiene  $req$  su  $C$ ?)
  - indirizzo IP
  - una vaghissima descrizione di qualcosa che riguarda un po'  $C$  (user-agent)
- Non va bene:
  - NAT
  - anonymizer
  - ...

# SERVER CON PIÙ CLIENT: IDEA DI BASE

$S$ , alla "prima" richiesta di  $C$

- nella risposta: "in tutte le prossime richieste, includi questo token"
- salva  $t$  in una lista dedicata

Quando arriva una richiesta  $req$  da un  $C$ :

- se non contiene il token, è la "prima"
- altrimenti ripesca lo stato

**Richiede la collaborazione del client!**

- potrebbe non capire/buttare il token
- potrebbe inserire in  $req$  un token a caso

Il token potrebbe essere  $S$ , di solito è molto più "piccolo"

# SERVER CON PIÙ CLIENT: IDEA DI BASE

*S* include nella risposta: "in tutte le prossime richieste, includi questo **token** *t*"

Meccanismi:

- **cookie**: il valore di un header HTTP
  - *S* mette *t* nel cookie *c* della risposta (*t* = *c*)
  - *C* mette il *c* nelle richieste
- **url rewriting**
  - *S* riscrive tutti i link interni dei documenti delle risposte a *C* appendendo *t* come valore di un parametro GET

`href="store/dogs/index.jsp?  
type=jack_russel&sess_id=t`

# COOKIES VS. URL REWRITING

- C potrebbe non accettare i cookie
- improbabile che C "depuri" i link dal sess\_id
- ... ma la soluzione dei cookie pare più comoda/elegante

# SESSIONI IN EXPRESS

- Usiamo il middleware `express-session`
- Si basa su cookies
- Viene creato una proprietà `session` all'interno dell'oggetto `req`
- Gestito a livello di applicazione: se legato ad autenticazione bisogna fare attenzione!
- Salvato in memoria (se voglio persistere posso legarlo ad un database)
- `secret` : parametro richiesto, è il token per firmare il cookie ID
- `resave` : forza il salvataggio della sessione, non sarebbe richiesto ma il valore di default (true) è deprecato

# SESSIONI IN EXPRESS

```
1 const session = require('express-session');
2 app.use(session({
3     secret: 'segreto',
4     resave: false
5 }));
6 app.post('/login', (req, res) => {
7     /* codice per verificare se utente esiste */
8     if(user !== null) {
9         req.session.user = {
10             email: user.email,
11             name: user.name
12         };
13         res.redirect('/account');
14     } else {
15         /* Login non valido */
16     }
17 });
18 app.get('/info', (req, res) => {
19     if(!req.session.user) {
20         res.redirect('/login');
21     } else {
22         res.send(req.session.user);
23     }
24 });
```

# **AUTORIZZAZIONE E AUTENTICAZIONE**

# ESIGENZA DI BASE

L'accesso ad alcune risorse deve essere limitato

- "accesso a risorse" ≈ "svolgimento di operazioni"
- "limitato"...
  - nel tempo (dalle 9:00 alle 19:00)
  - secondo la provenienza fisica (solo richieste provenienti dalla macchina/rete C)
  - secondo il tipo di canale (HTTPS)
  - ...
  - secondo l'identità di chi richiede l'accesso

Perché limitare? Sicurezza!

# LIMITAZIONE ACCESSO E IDENTITÀ RICHIEDENTE

L'accesso ad alcune risorse deve essere limitato, secondo l'identità di chi richiede l'accesso

1. **Client → Server:** «Vorrei accedere alla risorsa *req*»
2. **Server:** «*req* è tra le risorse ad accesso limitato? Sì»
3. **Server → Client** «Chi sei?»
4. **Client → Server:** «Sono *C*» ← **autenticazione**
5. **Server:** «Il client è veramente *C*? Sì»
6. **Server:** «*C* può accedere a *req*? Sì» ← **autorizzazione**
7. **Server → Client** «Eccoti *req*!»

A questo punto, il server considera *C* *logged in* per un po' di tempo/richieste (~ sessioni)

# AUTENTICAZIONE E AUTORIZZAZIONE

- S: «Chi sei?» C: «Sono C» S: «Il client è veramente C? Sì»
  - Il server sa se/come chiedere al client chi è
  - Deve anche essere in grado di verificare l'affermazione di C: assoluto complicato...
- S: «C può accedere a *req*? Sì»
  - Il server sa chi può acceder a *req*: semplice (una lista)

# SICUREZZA DICHIARATIVA/PROGRAMMATICA

**Nota:** *Per noi, "sicurezza" ≈ "autenticazione/autorizzazione"*

Dichiarativa → i processi di autenticazione/autorizzazione sono  
**definiti nella configurazione**

- quali risorse sono ad accesso limitato
- che meccanismo di richiesta credenziali usare, come validarle (autenticazione)
- per ogni risorsa, chi può accedere (autorizzazione)

# SICUREZZA DICHIARATIVA/PROGRAMMATICA

Programmatica → i processi di autenticazione/autorizzazione sono definiti (anche) nel codice

- Necessaria nei casi non banali

alla risorsa *req* possono accedere gli utenti  $U_1, U_2, U_3$  solo se l'utente  $U_0$  ha avuto accesso ad *req* almeno 5 volte

La definisce il programmatore "a manina"

# AUTENTICAZIONE SU EXPRESS

Posso implementarla in due modi

- Soluzione autocostruita
  - Sfrutto le sessioni
- Uso un middleware
  - `passport`
  - Libreria pronto, ma fa tante cose

# AUTENTICAZIONE SU EXPRESS

## Autocostruita

```
app.post('/login', function(req, res) {
  /* recupero credenziali dal DB */
  if(user.id === req.body.id && user.pass === req.body.pass) {
    req.session.user = user;
    res.redirect('/restricted');
  }
  res.send('Credenziali non valide');
});

app.get('/restricted', checkSignIn, function(req, res) {
  res.send('dati protetti');
});

function checkSignIn(req, res, next) {
  if(req.session.user) { return next() }
  next(new Error("Non loggato!"));
}
```

# AUTENTICAZIONE SU EXPRESS

## Passport.js

```
1 const passport = require('passport');
2 const LocalStrategy = require('passport-local');
3 passport.use(new LocalStrategy((user, pass, done) => {
4     /* verifico se username e pass sono nel DB */
5     if( !authenticated ) { return done(null, false); }
6     if( authenticated ) return done(user);
7 }));
8
9 app.get("/restricted", (req, res, next) => {
10     /* gestisci login */
11     if (req.isAuthenticated()) { return next(); }
12     res.redirect('/')
13 },
14 (req, res) => {
15     /* body del metodo se autenticato */
16 });
```

# JWT - JSON WEB TOKEN

*JSON Web Tokens are an open, industry standard  
RFC 7519 method for representing claims securely  
between two parties.*

- Non salvo più le informazioni nella sessione, ma nel client.
- Come? Con una stringa (token) firmata!

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDI  
SfLKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV\_adQssw5c

Tre sezioni, separate dal ":" e codificate in Base64

- Header → tipo di token e algoritmo usato per la firma
- Payload
- Firma

# JWT - JSON WEB TOKEN

1. Il client invia al server le credenziali
2. Se tutto va bene, il server firma e restituisce il JWT
3. Il client in tutte le comunicazioni successive invia il JWT al server
4. Il server verifica la firma ed estrarre i dati

Come si invia il token al server?

- Header HTTP: Authorization: **Bearer** token
- Custom...

# JWT - JSON WEB TOKEN

1. Il client invia al server le credenziali
2. Se tutto va bene, il server firma e restituisce il JWT
3. Il client in tutte le comunicazioni successive invia il JWT al server
4. Il server verifica la firma ed estrarre i dati

Cosa ci guadago?

- I server nel punto 1 e 3 possono essere diversi
- Posso usare le stesse credenziali su più server
- Non devo gestire a mano le sessioni (proprie di ogni container)

# JWT IN NODE.JS

```
1 const jwt = require('jsonwebtoken');
2 app.post('/api/login', (req,res)=>{
3     const user = { /* dati utente */ };
4     jwt.sign({user:user}, 'secretkey', (err,token)=>{
5         res.json({token,});
6     });
7 });
8 app.post('/api/profile', (req,res,next) => {
9     const bearerHeader = req.headers['authorization'];
10    if(typeof bearerHeader!=='undefined'){
11        const bearerToken = bearerHeader.split(' ')[1];
12        req.token=bearerToken;
13        next();
14    } else {
15        res.sendStatus(403);
16    }, (req,res)=>{
17        jwt.verify(req.token, 'secretkey', (err, authData)=>{
18            if(err){ res.sendStatus(403); }
19            else{ res.json({ /* dati */}); }
20        });
21    });
}
```

# JWT IN NODE.JS

```
1 const jwt = require('jsonwebtoken');
2 app.post('/api/login', (req,res)=>{
3     const user = { /* dati utente */ };
4     const token = jwt.sign({user:user}, 'secretkey', {expiresIn: 86400});
5     res.cookie("token", token, {httpOnly: true});
6     res.json("Autenticated!");
7 });
8 app.post('/api/profile', (req,res,next) => {
9     const bearerHeader = req.headers['authorization'];
10    if(typeof bearerHeader!=='undefined'){
11        const bearerToken = bearerHeader.split(' ')[1];
12        req.token=bearerToken;
13        next();
14    } else {
15        res.sendStatus(403);
16    }, (req,res)=>{
17        jwt.verify(req.token, 'secretkey', (err, authData)=>{
18            if(err){ res.sendStatus(403); }
19            else{ res.json({ /* dati */ }); }
20        });
21    });
}
```

# NPM

# NODE PACKAGE MANAGER

- Incluso in Node.js
- Non legato all'uso di Node.js
- Gestisce l'installazione di moduli/pacchetti (sinonimi)
- Semplifica la vita...

# PACKAGE.JSON

- Manifest dell'applicazione
- Nome e versione dei moduli richiesti (dipendenze)
- Vincoli sulle versioni (es: posso aggiornarlo?)
- Crea gli script NPM

Si crea semplicemente con `npm init`

# INSTALLARE, RIMUOVERE E AGGIORNARE

- Aggiungere moduli: `npm install cosa --save`

Perché `--save` ?

  - I moduli vengono scaricati in `node_modules`
  - Non vogliamo portare la cartella nel controllo versione → salviamo in `package.json`
- Dipendenze solo per sviluppo: `npm install cosa --save-dev`
- Installare i moduli richiesti: `npm install`
  - in produzione: `npm install --production`
- Rimuovere: `npm uninstall cosa --save`
- Aggiornare: `npm update cosa`
- Elencare: `npm list`
- cosa può specificare la versione con `@`

# VERSIONI

Le versioni sono sempre **Major.Minor.Patch**

- **Major:** può rompere l'API
- **Minor:** nuove feature, non rompe l'API
- **Patch:** corregge bug

`^Maj.min.patch` → installa la minor/patch più recente

`~Maj.min.patch` → installa la patch più recente

`Maj.min.patch` → installa esattamente quella

`*` → installa la versione più recente (pericoloso)

# PACCHETTI LOCALI E GLOBALI

I pacchetti di default sono **locali**, propri di un progetto.

I pacchetti **globali** sono installati per tutto il PC.

- `npm install -g cosa`

Magari con sudo

- `npm remove -g cosa`

# SCRIPT NPM

Permettono di eseguire task

- Definiti in package.json
- Eseguiti con node run nomeScript
- Alcune piattaforme cercano lo script start

Si esegue direttamente con node start , senza  
run

# MONGODB

# MONGODB

Caso tipico di una web application: interagire con un database

La tipologia del database non è importante  
(SQL/NOSQL, indifferente)

Abbinamento Node.js e MongoDB viene "naturale"

# MONGODB CON NODE.JS

- Connessione
- Operazioni CRUD

Useremo il modulo `mongodb`

# CONNESSIONE

Connection URI:

```
mongodb+srv://user:pass@sample.host:27017/?poolSize=20&writeConcern=majority
```

protocol

credentials

hostname/IP and port of  
instance(s)

connection options

Fonte: [MongoDB](#)

- Oggetto MongoClient per gestire connessione
- Esporta una promise connect() che apre la connessione
- Andrebbe chiuso in caso di errore o a fine operazioni con close()
- MongoClient restituisce il database di MongoDB tramite db()
- dal database posso accedere alla collection tramite collection()

# CONNESSIONE

```
1 const { MongoClient } = require("mongodb");
2 const uri = "mongodb://localhost:27017";
3 const client = new MongoClient(uri);
4 async function connect() {
5   try {
6     await client.connect();
7   } finally {
8     await client.close();
9   }
10 }
11 connect().catch(console.log);
```

# OPERAZIONI CRUD

- Recupero documenti, ritornano o un singolo documento o un `FindCursor`
  - `Collection.find()`
  - `Collection.findOne()`
  - `Collection.aggregate()`
- Inserimento documenti
  - `Collection.insert()`
  - `Collection.insertOne()`
- Modifica documenti, posso modificare (serve un filtro) o sostituire
  - `Collection.updateOne()`
  - `Collection.updateMany()`
  - `Collection.replaceOne()`
- Eliminazione documenti
  - `Collection.deleteMany()`
  - `Collection.deleteOne()`

# OPERAZIONI CRUD

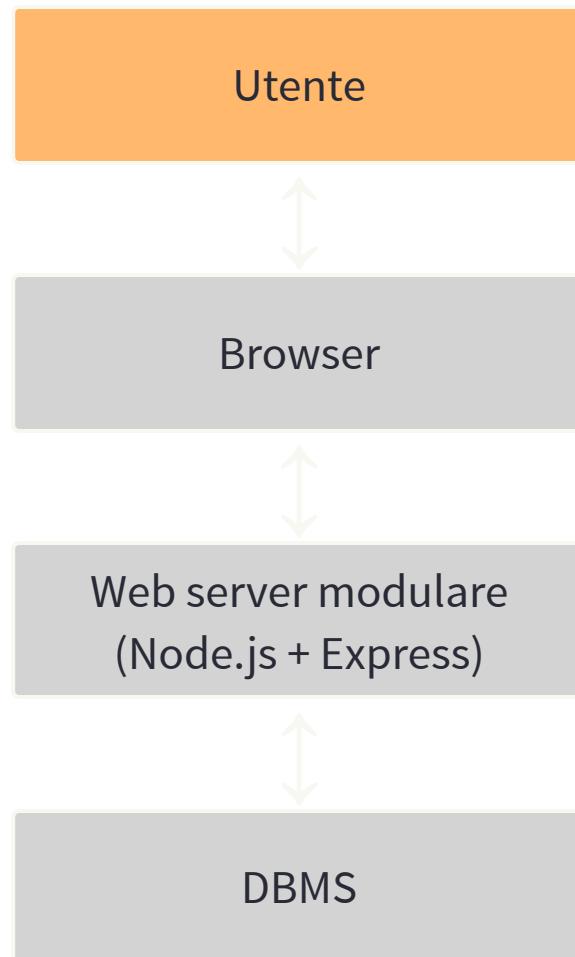
```
1 const query = { title: "Titolo cercato" };
2 const options = {
3   sort: { "imdb.rating": -1 }
4 };
5 const movie = await movies.findOne(query, options);
```

```
1 const query = { runtime: { $lt: 15 } };
2 const options = {
3   sort: { "imdb.rating": -1 }
4 };
5 const cursor = movies.find(query, options);
6 await cursor.forEach(console.dir);
7 console.log(await cursor.toArray());
```

# **WEB SERVICES**

# ARCHITETTURA WEB APP

Vista nel corso:



# UTENTE E APP

L'utente può usare la app tramite:

- browser su desktop
- browser su smartphone
- app su smartphone
- app su smartTv
- ...
- utente è un'altra app

Tutti vogliono parlare con noi tramite Internet...

# INTEROPERABILITÀ

## WEB SERVICES

Web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks.

Ma cosa cambia rispetto ad una pagina web normale?

# INTEROPERABILITÀ

## WEB SERVICES

Web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks.

Ma cosa cambia rispetto ad una pagina web normale?

Cambia il destinatario:

- Pagina Web → essere umano
- Web Service → programma

# INTEROPERABILITÀ

## WEB SERVICES

Web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks.

Ci sono tre possibili strade:

1. Autocostruita
2. Seguire uno **standard** esistente
3. Implementare un **design pattern** predefinito

Quale scegliere?

- Chi deve usare il sistema?
- Che linguaggi deve supportare?

# **STRADA 1: AUTOCOSTRUITA**

# SOLUZIONE AUTOCOSTRUITA

Decidiamo noi lo standard con cui esporre il servizio

- Abbiamo già tutti gli strumenti per farlo
- Essenziale fornire una specifica su input e output

```
1 const express = require('express');
2 const app = express();
3 app.get('*', (req, res) => {
4   if(req.query.program !== undefined) {
5     /*crea risposta XML per il programma chiamante*/
6   } else {
7     /*crea pagina HTML per l'essere umano*/
8   }
9 }) ;
```

# SOLUZIONE AUTOCOSTRUITA

Decidiamo noi lo standard con cui esporre il servizio.

Problemi:

- Deserializzazione manuale dell'input (stringhe)
- Quali linguaggi supporteremo?
- Descrivere in modo ossessivo il tutto
  - "I numeri useranno il . quale separatore decimale"
  - "Il valore booleano falso è 'FALSE', qualunque altro è vero"
  - "Un intero può avere al massimo 64 bit"
  - ...
- Non tragico se produttore == consumatore
- ...che succede se domani altri dovranno usarlo?

**STRADA 2: SEGUIRE  
UNO STANDARD**

# SEGUIRE UNO STANDARD: BREVE STORIA

- 1990: TCP/IP diventa **LO** standard
- Nascono tante tecnologie, indipendenti dal linguaggio:
  - **CORBA**: usa Interface Description Language per descrivere le interfacce esposte, e CORBA si occupa poi di mapparlo al linguaggio di implementazione.

**Nota:** E come si fa con C, che non è ad oggetti?

- **DCOM** (Distributed COM): di Microsoft, per combattere CORBA. Aggiunge a COM il Marshalling e una Garbage Collection distribuita.

**Nota:** Firewall non felici, non usa porta 80

- **Java RMI** (Remote Method Invocation): chiamate da una JVM all'altra poi esteso per esporre CORBA.

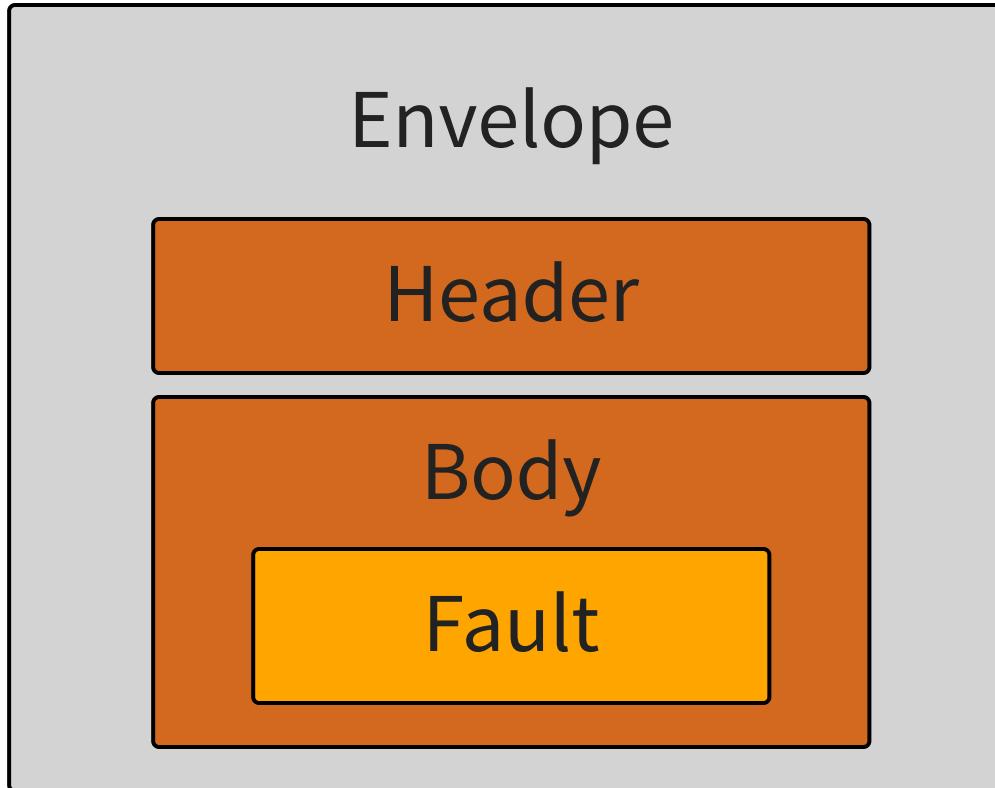
# SEGUIRE UNO STANDARD: BREVE STORIA

- 2000: nasce la religione 🕯 XML 🕯
- **XML-RPC** (Remote Procedure Call):
  - usa XML per codificare le chiamate;
  - trasporto via HTTP
  - i parametri non hanno nomi e vanno passati in ordine;
  - 6 pagine di specifiche.
- **SOAP** (Simple Object Access Protocol): XML-RPC evoluto
  - usa XML per codificare le chiamate;
  - usa HTTP(s), SMTP, JMS, ...;
  - parametri con nome;
  - pensato per trasmettere anche documenti;
  - 44 pagine di specifiche

**Nota:** *XML-RPC nasce perché Microsoft tentennava nell'approvare SOAP!*

# SOAP

Standard di W3C, attualmente alla [versione 1.2](#). Elementi principali:



**Header:** meta-informationi (es: versione, informazioni su transazioni, ecc.)

**Nota:** Solo *Envelope* e *Body* sono obbligatori

# SOAP MESSAGE

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
    ...
  </soap:Header>
  <soap:Body>
    ...
    <soap:Fault>
      ...
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

**Nota:** *L'encoding dei dati non è predefinito, e possono convivere più tipi di encoding*

# SOAP BODY

Usato per Request...

```
<soap:Body>
  <m:GetPrice xmlns:m="http://www.w3c.com/prices">
    <m:Item>Apples</m:Item>
  </m:GetPrice>
</soap:Body>
```

...e Response

```
<soap:Body>
  <m:GetPriceResponse xmlns:m="http://www.w3c.com/prices">
    <m:Price>1.90</m:Price>
  </m:GetPriceResponse>
</soap:Body>
```

...ed eventuali errori

# SOAP BINDING

Finora non abbiamo parlato del canale o del server  
con cui parlare...

Se si usa HTTP, gli header devono specificare:

- Content-Type : di solito è application/soap+xml
- Content-Length
- **SOAPAction: "URI"**
  - non deve per forza essere risolvibile;
  - non dipende dalle specifiche SOAP, ma dall'applicazione

# SOAP: MESSAGGIO COMPLETO

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn
SOAPAction: "/GetStockPrice"

<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
  soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding"
<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>
```

# SOAP ERRORS

Il server in risposta dovrebbe dare solo:

- 2xx: messaggio ricevuto e compreso
  - potrei comunque avere errori in Body/Fault
- 500: il messaggio non può essere processato
  - mancano informazioni
  - non compreso, ecc.
  - dovrei comunque ottenere una envelope il cui Body/Fault contiene la spiegazione.

# **SOAP E ALLEGATI**

E se dovessi inviare degli allegati?

Soluzione 1: codificato nell'envelope in BASE64

... molto poco efficiente...

# SOAP E ALLEGATI - MTOM

E se dovessi inviare degli allegati?

Soluzione 2: Message Transmission Optimization Mechanism

utilizzando MIME e Multipart

```
MIME-version: 1.0
Content-Type: Multipart/Related; ...
--MIME_boundary
Content-Type: text/xml; ...
<?xml version="1.0" ?>
<env:Envelope ...
    <someTag href="cid:attached.gif@company.com"/>
</end:Envelopeattached.gif@company.com">
...binary dell'immagine...
```

# SOAP E NODE.JS

Come realizzo un Web Service?

1. Scrivo un programma e gestisco il metodo POST

- molto lavoro
- possibile che non ci sia una soluzione già pronta?

# SOAP E NODE.JS

Come realizzo un Web Service?

1. Scrivo un mio script e gestisco il metodo POST

- molto lavoro
- possibile che non ci sia una soluzione già pronta?

2. Utilizzo [node-soap](#) :

- Modulo dedicato per servizi SOAP
- Alternativo ad Express

# SOAP E DOCUMENTAZIONE

Ma ho ancora un grosso problema: come fa l'utente a sapere

- che WebService espongo?
- che parametri accettano?
- cosa otterrò come risposta?

Devo scrivere documentazione?

# SOAP E DOCUMENTAZIONE

Ma ho ancora un grosso problema: come fa l'utente a sapere

- che WebService espongo?
- che parametri accettano?
- cosa otterrò come risposta?

Devo scrivere documentazione? NO!

Web Services Description Language, [standard W3C](#)

# WSDL

Usa XML per definire le interfacce, composto da quattro elementi:

- <types> : tipi di dati usati dal WS (non necessario se uso i primitivi già noti)
- <message> : definizione dei dati che vengono trasmessi
- <portType> : operazioni che si possono compiere e relativi messaggi
- <binding> : protocollo e formato dei dati usati per ogni porta

# WSDL

<portType> non gestisce solo domande e risposte:

- **One-way**: il WS riceve dati ma non ne restituisce
- **Request-response**: il WS riceve ed invia dati
- **Solicit-response**: il WS invia dati al client e attende una risposta
- **Notification**: il WS invia solo dati

**Nota:** *Si passa da uno all'altro cambiando l'ordine e la presenza degli elementi input/output*

# WSDL - ESEMPIO

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>
<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>

<binding type="glossaryTerms" name="b1">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/
  <operation>
    <soap:operation soapAction="http://example.com/getTerm"/>
    <input><soap:body use="literal"/></input>
    <output><soap:body use="literal"/></output>
  </operation>
</binding>
```

# **STRADA 3: IMPLEMENTARE UN DESIGN PATTERN**

# RESTFUL WEB SERVICE

Representational State Transfer (RESTful)

- trasferimento dello stato (attuale o desiderato) di una rappresentazione della risorsa
- operazioni sulle risorse sono quelle tipiche (**CRUD**: crea, leggi, aggiorna, cancella)
- **non** è uno standard per descrivere l'interfaccia
- **non** è uno standard per descrivere lo scambio dati
- **nessun** rigido contratto client-server
- un client rest è assimilabile ad un browser

Nasce dalla **tesi** di Roy T. Fielding

# RESTFUL WEB SERVICE

Elementi chiave:

- A REST API should be independent: on any single communication protocol, though its successful mapping to a given protocol may be dependent on the availability of metadata, choice of methods, etc.
- A REST API should not contain any changes to the communication protocols aside from filling-out or fixing the details of underspecified bits of standard protocols

# RESTFUL WEB SERVICE: HATEOAS

## Hypermedia as the Engine of Application State

- HTTP + JSON
- È il modo più diffuso per implementare API REST
- Una base URL per ogni risorsa, es. `http://myapp.com/api/entry`
- La tecnologia Java è JAX-RS
  - non c'è un vero e proprio standard
  - non genera documentazione per i client (ma posso farmi aiutare: [Swagger](#), conforme a [OpenAPI](#))

# RESTFUL WS: OPERAZIONI HATEOAS

- operazioni su collection: <http://myapp.com/api/entry>
  - GET: legge gli item della collection (eventualmente con parametri)
  - PUT: sostituisce la collecion
  - POST: inserisce un elemento nella collection
  - PATCH: modifica parziale (?)
  - DELETE: elimina l'intera collection
- operazioni su singolo item: <http://myapp.com/api/entry/17>
  - GET: legge l'item 17
  - PUT: sostituisce l'item 17
  - DELETE: elimina l'item 17
  - ....

**Nota:** non è necessario implementare tutto

# REST: ESEMPIO

Risorsa	Metodo
Lista libri	[GET] /books.
Dettaglio libro	[GET] /books/<id>.
Creazione libro	[POST] /books POST.
Modifica libro (o creazione libro con uno specifico ID)	[PUT] /books/<id>.
Cancellazione libro	[DELETE] /books/<id>.

# RESTFUL WS: ESEMPIO

GET <http://myapp.com/api/entry>

```
[ {  
    id: 17,  
    date: "2013/15/5 11:58:00",  
    amount: 17.38,  
    currency: "eur",  
    description: "pranzo"  
, {  
    id: 185,  
    date: "2013/15/5 8:30:00",  
    amount: 1.20,  
    currency: "eur",  
    description: "caffè"  
}]
```

# RESTFUL: OPENAPI

Alcune idee di REST sono belle, ma di fatto impraticabili... occorre documentazione!

*OpenAPI is an API description format  
for REST APIs*

- Inizialmente chiamata "Swagger Specification"
- Insieme di strumenti per generare codice e documentazione
  - codice server da specifiche
  - documentazione da codice server
  - client da documentazione
- Usa YAML o JSON

# RESTFUL: COME CREARLI

Alcune opzioni

- a mano
- Express!

# XSS

# CROSS-SITE SCRIPTING (XSS)

- Vulnerabilità informatiche che affliggono le applicazioni web
- Sono complesse da trovare, prevenire e capire
- Si possono organizzare in 3 tipi:
  - Reflected
  - Stored
  - DOM-based

# REFLECTED (NON-PERSISTENT)

- Documento web dinamico che contiene valori estratti da una richiesta HTTP
- XSS: i valori dei parametri non sono filtrati

# REFLECTED (NON-PERSISTENT)

```
const http = require('http');
const url = require('url');

http.createServer(function (req, res) {
  const queryObject = url.parse(req.url, true).query;
  res.write('<html>');
  if(queryObject.par1 !== undefined) {
    res.write(queryObject.par1);
  }

  res.write('</html>');
  res.end();
}).listen(3000);
```

http:.../?par1=<script>alert("ciao");<script>

# **STORED (PERSISTENT)**

- Documento web **dinamico** che contiene valori estratti da un **database**
- L'attacco XSS era stato inserito precedentemente da un altro utente

# "È SOLO UNA INNOCUA ALERT!"

- Stiamo inserendo del codice in un programma
- Possiamo far fare qualsiasi cosa al browser
- `document.cookie`
  - invio i cookie ad un attaccante

# **"SI, MA COME METTO IL PARAMETRO MALEVOLO?"**

- L'attacante deve creare un link fatto apposta
- Deve far cliccare l'utente sul link malevolo

# "MA C'È LA SAME ORIGIN POLICY!"

- SOP si applica agli script e ai frame, ma...
- ... nel DOM ci elementi grafici che usano URL
  - img
  - link
- ... questi vengono risolti automaticamente senza restrizioni!

```

```

# DOM BASED

- Documento web **dinamico** che contiene valori ricevuti dall'**utente** e manipolati lato **client**
- Uno script modifica il DOM in base a questi valori

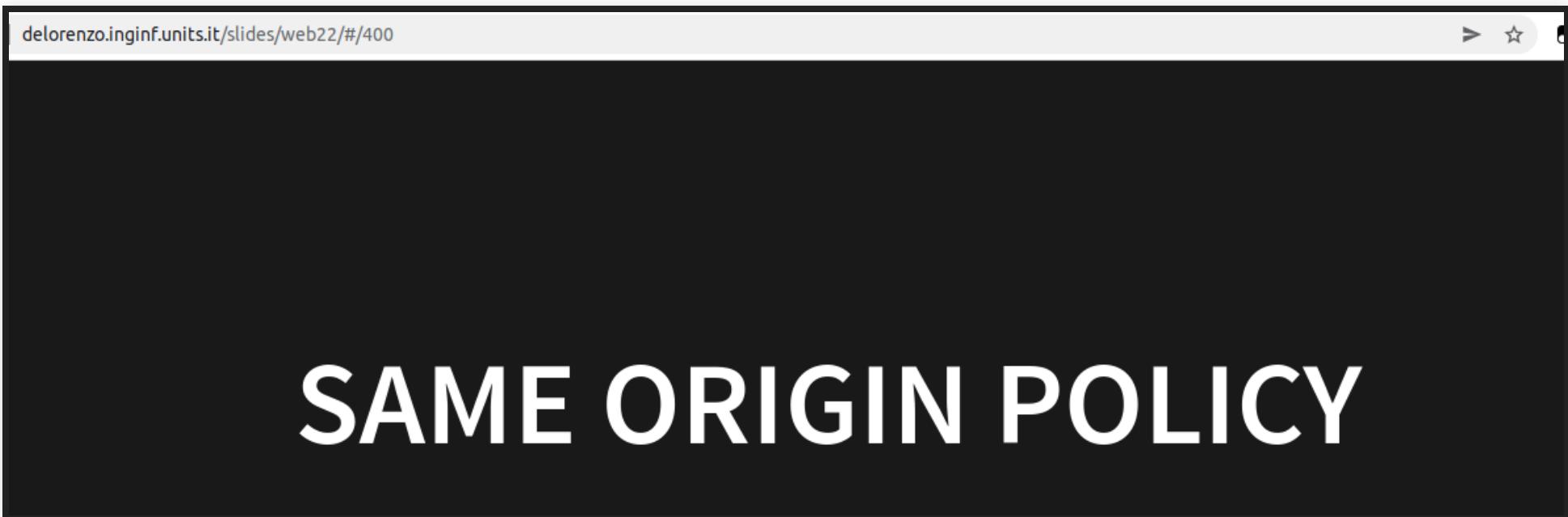
# DOM BASED

Come dovrebbe funzionare l'attacco?

- Link malevolo che contiene del codice da far eseguire
- Non viene inviato al server, quindi non può essere filtrato/rilevato lato server
- Uno script legge l'url e prende parte di questo per fare "qualcosa"

# DOM BASED

Molto comune nelle Single Page Application



L'anchor ( # ) viene utilizzato per navigare tra le slide: del codice javascript leggerà l'url ed estrarrà il numero della slide

# DOM BASED

```
url = new URLSearchParams(location.search);
x = url.get('x');
document.write(x);
```

[http://trusted.com?n=<script>alert\(...\)</script>](http://trusted.com?n=<script>alert(...)</script>)

# SOURCE E SINK

- Alcuni oggetti sono potenzialmente controllabili dall'attaccante (**source**)
  - `document.URL`
  - `document.referrer`
  - `location.search`
- Alcune funzioni sono potenzialmente pericolose (**sink**)
  - `document.write()`
  - `innerHTML`
  - `eval()`

# SOURCE E SINK

- In realtà i browser recenti aggiungono delle restrizioni
- `document.URL` viene convertito
- devo esplicitamente invocare `decodeURIComponent()`

# NON FUNZIONA

```
const pos=document.URL.indexOf("n=")+2;  
const n = document.URL.substring(pos,document.URL.length)  
document.write(n);
```

[http://trusted.com?n=<script>alert\(...\)</script>](http://trusted.com?n=<script>alert(...)</script>)

→ stampa %3Cscript%3Ealert(1)%3C/script%3E

# ESEMPIO FUNZIONANTE

```
<body>
  <main>
    
  </main>
</body>
```

```
const search = location.search;
const params = new URLSearchParams(search);
const data = params.get("n");
const main = document.querySelector("main");
main.innerHTML = "<img src='"+data+'.png' />";
```

[http://trusted.com?n=%27%20onerror=%27alert\(\)%27](http://trusted.com?n=%27%20onerror=%27alert()%27)

# ESEMPIO FUNZIONANTE

```
<body>
  <main>
    <img src=' ' onerror="alert()"/>
  </main>
</body>
```

```
const search = location.search;
const params = new URLSearchParams(search);
const data = params.get("n");
const main = document.querySelector("main");
main.innerHTML = "<img src='"+data+".png' />";
```

[http://trusted.com?n=%27%20onerror=%27alert\(\)%27](http://trusted.com?n=%27%20onerror=%27alert()%27)

# PREVENZIONE

In pillole:

- Considerare **ogni input** dell'utente come **pericoloso**
- Fare sempre *sanitizzazione*
  - Express: [express-validator](#)
  - Libreria [Joi](#), definizione di schemi e validazione

**VUE.JS**

# VUE.JS

*Vue (pronounced /vju:/, like view) is a progressive framework for building user interfaces. Unlike other monolithic frameworks, Vue is designed from the ground up to be incrementally adoptable.*

- È un framework JavaScript
- Semplifica la creazione di GUI e front-end
- *Progressive*: non occorre imparare tutto subito

# PERCHÈ VUE.JS

- Curva di apprendimento relativamente meno ripida rispetto altri framework (REACT, Angular, ...)
- Molto leggero (20KB)
- Componenti:
  - struttura ordinata del progetto
  - non separiamo le responsabilità (MVC), ma i componenti (html/stile/codice uniti)

# VUE.JS: USO

Devo includere Vue nel mio documento web

Conviene usare una CDN:

```
<script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
```

Versione di produzione (senza warning e debug):

```
<script src="https://unpkg.com/vue@3/dist/vue.global.prod.js"></script>
```

# VUE.JS: USO

Ogni applicazione inizia creando una nuova istanza  
Vue tramite `createApp()` :

```
const { createApp } = Vue
const app = createApp({
  /* opzioni */
})
```

# VUE.JS: MOUNTING

Una applicazione deve essere legata ad un *container*, un elemento HTML dove fare il rendering dell'applicazione

Si fa tramite il metodo `mount()` :

- viene invocato dopo aver configurato e inizializzato la app
- prende come input un elemento HTML o un selettore CSS

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

```
app.mount('#app')
```

# **REACTIVITY SYSTEM**

Vue.js si basa sul **reactivity system**: se un oggetto JavaScript viene modificato, la *view* viene aggiornata

# DATA

Gli oggetti che voglio controllare col **reactivity system** vanno passati tramite la proprietà `data`

`data` sarà un metodo che restituisce gli oggetti da monitorare.

Il **reactivity system** si occuperà di "reagire" ogni volta che una di queste proprietà viene modificata eseguendo il rendering.

```
var data = { a: 1 }

const app = createApp({
```

# INTERPOLAZIONI

Vue.js usa una sintassi basata su template HTML dinamici. Per legare i dati al render ha delle funzioni di interpolazione

- `{{ x }}` detto "mustache", nell'html viene sostituito dalla valutazione di `x`
- se voglio specificare che `x` va resa come html, devo usare la direttiva `v-html`

# METODI, FUNZIONI E COMPORTAMENTI

- È possibile associare ad una istanza Vue delle funzioni
- Queste vanno messe nella proprietà `methods`
- Anche queste funzioni sono reattive al cambiamento
- Possono essere usate come handler degli eventi DOM

# DIRETTIVE

- Le direttive sono attributi speciali con il prefisso `v-`
- Sono pensate per ricevere una **singola espressione JavaScript**
- Le direttive applicano un effetto reattivo quando la loro espressione cambia
- Possono accettare degli argomenti passati con `i :`

```
<p v-if="seen">Ora mi vedi</p>
```

La direttiva `v-if` rimuove/inserisce l'elemento se l'espressione "seen" è falsa/vera

# DIRETTIVE

- `v-if` mostra o nasconde un elemento se l'espressione è vera o falsa
- `v-for` ripete il componente per ogni elemento di un oggetto iterabile
- `v-on:event` associa una funzione all'evento event
- `v-bind:attr` lega il valore di una attributo attr ad una espressione
- `v-html` imposta innerHTML dell'elemento con l'espressione

# SCORCIATOIE

- Alcune direttive si possono abbreviare per migliorare leggibilità del codice
  - v-on:event → @event
  - v-bind:attr → :attr

# COMPUTED PROPERTIES

- Proprietà particolari con valori dinamici
- Ricavate da altre proprietà
- Definite nella proprietà `computed`
- Si aggiornano automaticamente quando una dipendenza cambia
- Permettono di fare caching e non richiamare funzioni inutilmente

```
const { createApp } = Vue
const data = {firstname: "Andrea", lastname: "De Lorenzo"};
const app = createApp({
  data() {
    return data
  },
  computed: {
    fullname: function() {
      return this.firstname + ' ' + this.lastname;
    }
  }
});
app.mount('#app');
```

# CLASSI CSS

- Possiamo modificare l'attributo `class` con la direttiva `:class`
- Molto comodo per manipolare come appare il DOM
- Posso associare dei booleani ad ogni classe `:class="{'cl1': hasCl1, 'cl2': hasCl2}"`
- Posso farlo anche come array: `:class="['cl1', 'cl2']"`

# EVENTI

- Con v-on:event o @event possiamo gestire gli eventi
- Possiamo aggiungere dei modificatori:
  - .stop : blocca il bubbling
  - .prevent : previene l'azione standard
  - .once : invoca la callback una sola volta, poi la rimuove

```
<a href="..." @click.prevent="onClick">Ancora</a>
```

# INPUT UTENTE

- Posso legare l'input dell'utente ad un modello **bidirezionale**:
  - Cambiamenti all'input o al modello influiranno sull'altra componente
- Il modello è creato con la direttiva `v-bind="myModel"`

```
<main id="app">
  <input type="text" v-model="testo">
    {{testo}}
</main>
```

```
const app = createApp({
  data() {
    return {testo: ""}
  }
});
```

# COMPONENTI

- Sono una porzione di HTML arricchita
- Posso usarli con un custom tag `<mio-comp></mio-comp>`
- Se definiti nativamente (no build) vanno dichiarati con alcune [caveat](#)
- Vanno registrati prima della creazione dell'istanza Vue
  - **globalmente** se disponibili per tutta la app
  - **localmente** se solo dentro ad un altro componente

# COMPONENTI

## Registrazione globale

```
app.component('mio-comp', {  
  [...]  
}) ;
```

## Registrazione locale

```
export default {  
  [...],  
  components: {  
    'mio-comp': {  
      [...]  
    }  
  }  
} ;
```

# COMPONENTI: data

- Ad una componente può essere passato una proprietà data
- data rappresenta il modello dei dati reattivo per quel componente
- Non passo un oggetto, ma una funzione!
  - Potrei avere più istanze dello stesso componente
  - Voglio avere modelli separati per ogni componente

```
1 export default {
2   data: function() {
3     return {
4       name: 'Andrea',
5       age: 37
6     };
7   }
8 }
```

# COMPONENTI: template

- Ogni componente avrà una struttura HTML
- Viene passata tramite il parametro `template`
- Può essere definito in diversi modi:
  - Stringa
  - Single File Components

# COMPONENTI

## Stringa inline

```
const ButtonCounter = {
  data() {
    return {
      count: 0
    }
  },
  template: `
    <button @click="count++">
      Mi hai cliccato {{ count }} volte.
    </button>
  `
}
```

# COMPONENTI

Stringa in modulo separato

```
export default {
  data() {
    return {
      count: 0
    }
  },
  template: `
    <button @click="count++">
      Mi hai cliccato {{ count }} volte.
    </button>
  `
}
```

# COMPONENTI: props

- Ogni componente ha uno scope isolato
- Posso passare dei valori al componente da un altro componente
- Questi valori vengono definiti tramite la proprietà `props`
  - Array di stringhe
  - Proprietà di un oggetto (posso definire il tipo)
- Posso essere statici o dinamici
  - Statici: attributi del componente
  - Dinamici: shortcut col nome della prop

# COMPONENTI: props

```
<div id="app">
  <people-list :people="people"></people-list>
  <div>
    <input v-model="name" placeholder="name" /><br />
    <input v-model="surname" placeholder="surname" /><br />
    <button @click.prevent="addPerson">Add person</button>
  </div>
</div>
```

# COMPONENTI: props

```
const { createApp } = Vue

const PeopleList = {
  props: ['people'],
  template: `<person v-for="person in people" :person="person"></person>`
}
const Person = {
  props: ['person'],
  template: `<div><i>{{ person.name }}</i> <b>{{ person.surname }}</b></div>`
}
const app = createApp({
  data() { return { people: [{ name: 'Andrea', surname: 'De Lorenzo' }], name: '', surname: '' } },
  methods: {
    addPerson: function() {
      this.people.push({
        name: this.name,
        surname: this.surname
      });
      this.name = '';
      this.surname = '';
    }
  }
});
app.component("Person", Person);
app.component("PeopleList", PeopleList);
app.mount('#app');
```

# COMPONENTI: SINGLE FILE COMPONENT

- Posso definire componenti in file separati con dentro tutto
- Molto comodo ed ordinato:
  - Ogni componente sarà un file sorgente
  - Ogni componente contiene stile, funzionalità e struttura html
- Necessità di uno strumento di build:
  - [Vueify](#) da usare con [Browserify](#)
  - [vue-loader](#) da usare con [Webpack](#)

# COMPONENTI: SINGLE FILE COMPONENT

- Ogni componente sarà un file sorgente
- Ogni componente contiene stile, funzionalità e struttura html

## OUTPUT

```
<template>
  <div class="messaggio">
    <h1>{{msg.title}}</h1>
  </div>
</template>
```

## FUNZIONALITÀ

```
<script>
  export default {
    name: "messaggio",
    data() {
      return {
        msg: {title:"Help"}
      }
    }
  }
</script>
```

## STILE

```
<style scoped="">
  h1{
    font-size: 20px;
  }
</style>
```

# CREAZIONE DI UN COMPONENTE

- Creare un file .vue
- Creare le tre sezioni:
  - <template>
  - <script>
  - <style>
- Importare il file nel componente in cui va usato

# ROUTING

Spesso le Web App sfruttano il concetto di **Single Page Application**

Tutta l'applicazione sta in una singola pagina html le cui componenti cambiano dinamicamente a seconda del contenuto

# ROUTING

Comodo offrire dei *permalink* alle risorse in una Single  
Page Application

Si sfruttano le **ancore** dell'url (la parte che segue # )

In Vue.js si usa il **routing** per caricare template in base  
all'url

# ROUTING

```
import Home from './Home.js'
import About from './About.js'
import NotFound from './NotFound.js'

const routes = {
  '/': Home,
  '/about': About
}

export default {
  data() {
    return { currentPath: window.location.hash }
  },
  computed: {
    currentView() {
      return routes[this.currentPath.slice(1) || '/') || NotFound
    }
  },
  mounted() {
    window.addEventListener('hashchange', () => {
      this.currentPath = window.location.hash
    })
  },
  template: `<a href="/">Home</a> |<a href="#/about">About</a> | <a href="#/non-existent-path">Broken Link</a>
  <current-view><></current-view>`
}
}
```

