

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
WEB TECHNOLOGIES – LECTURE 02

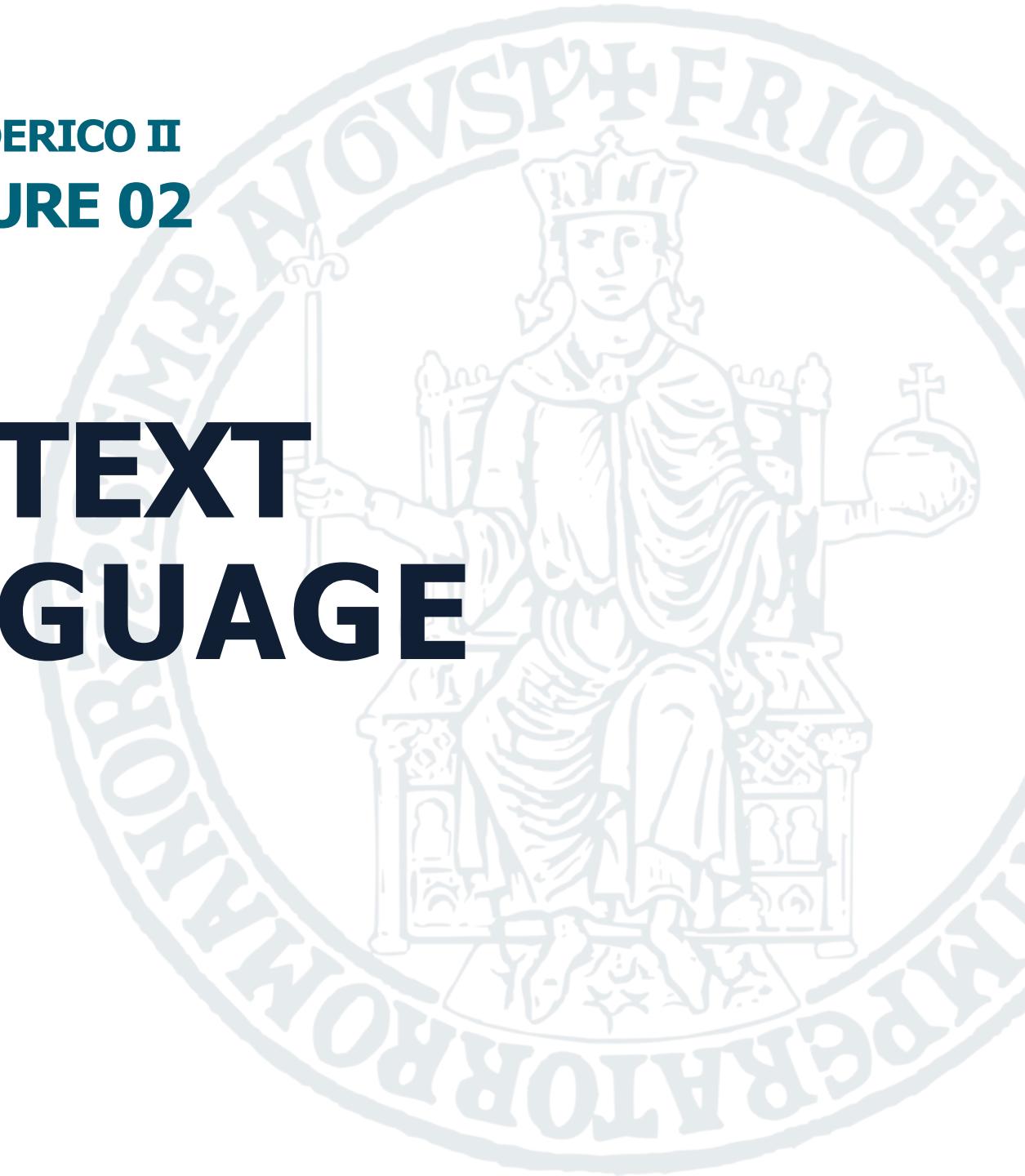
HTML: HYPERTEXT MARKUP LANGUAGE

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

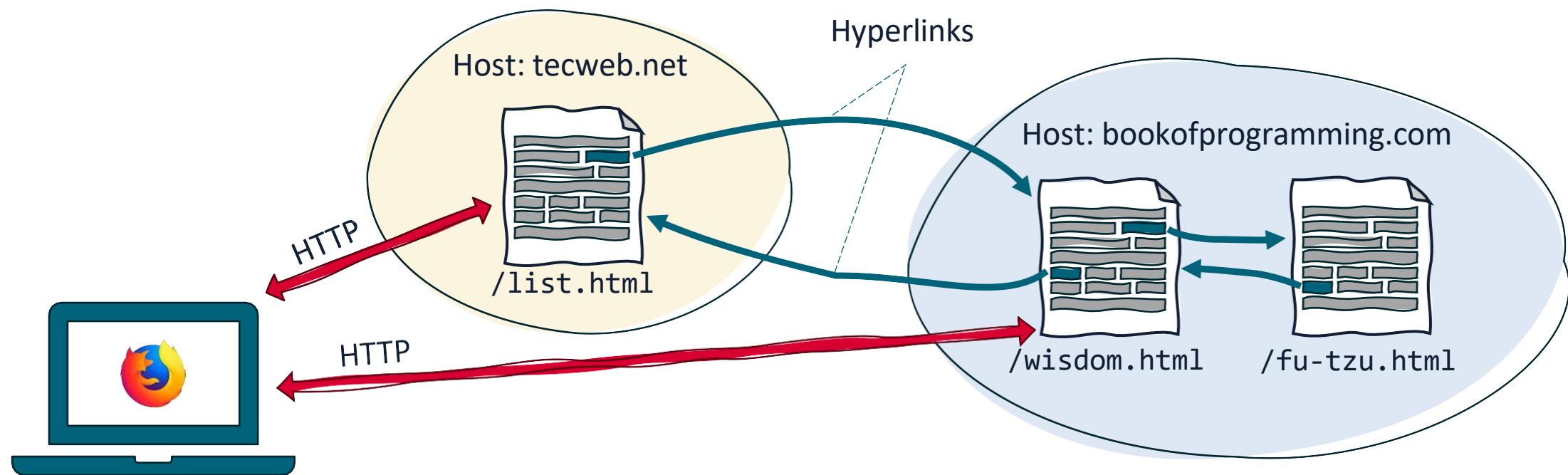
<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



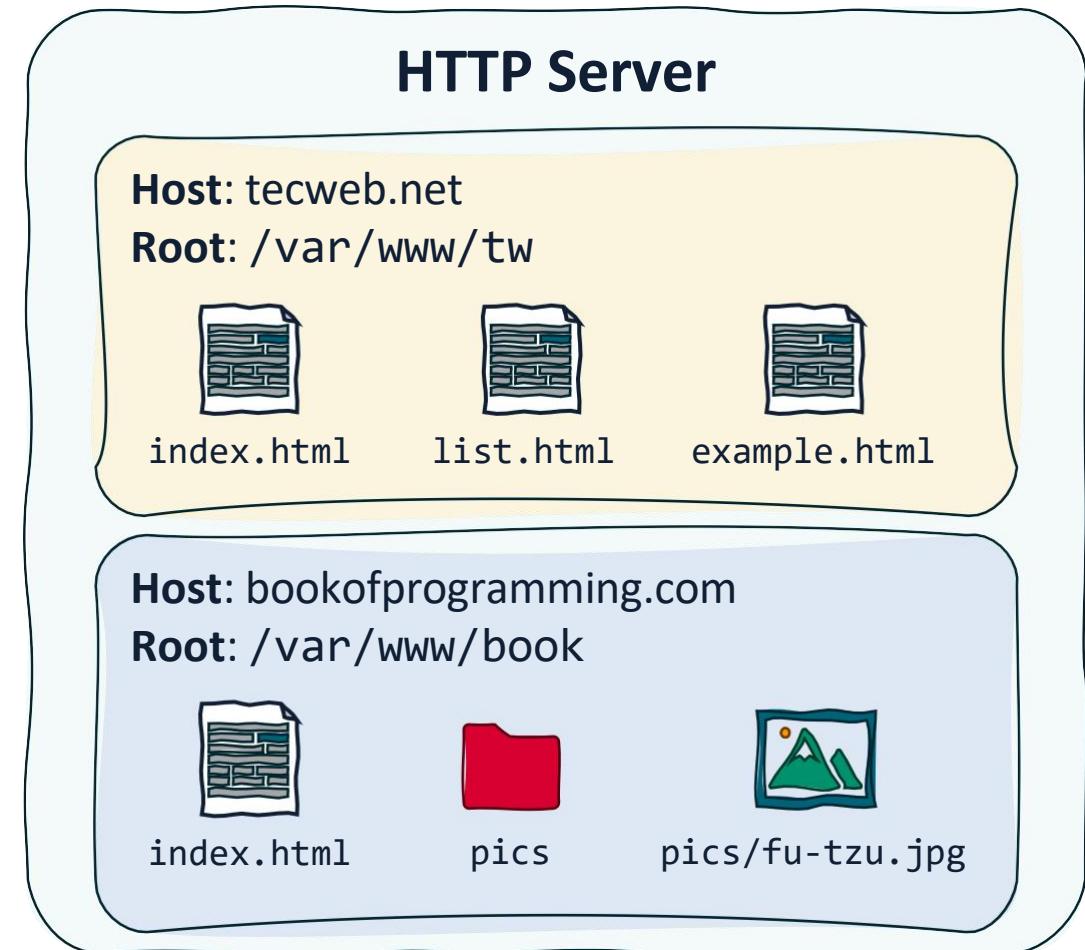
PREVIOUSLY, ON WEB TECHNOLOGIES

- We've seen **the web** is a system of interconnected **hypertext documents**, which are linked to each other through **hyperlinks**.
- Clients (typically web browsers) use HTTP to fetch these hypertexts



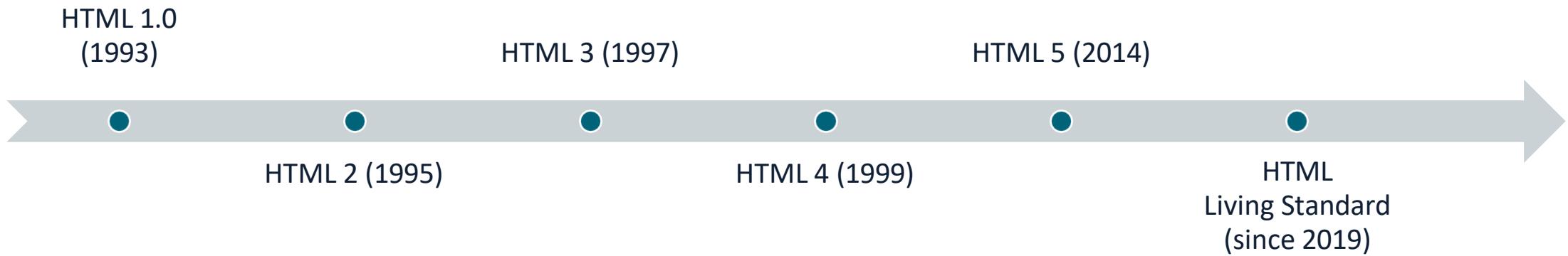
INSIDE AN HTTP SERVER

- An HTTP Server is a **software**
- **Listens** for HTTP requests on a certain port (e.g.: 80)
- Might manage multiple hosts
 - That's why theres a **Host** header in HTTP requests!
- Handles connections by serving files from the **Document root** in its filesystem
 - We do not want all our files to be accessible via HTTP, right?

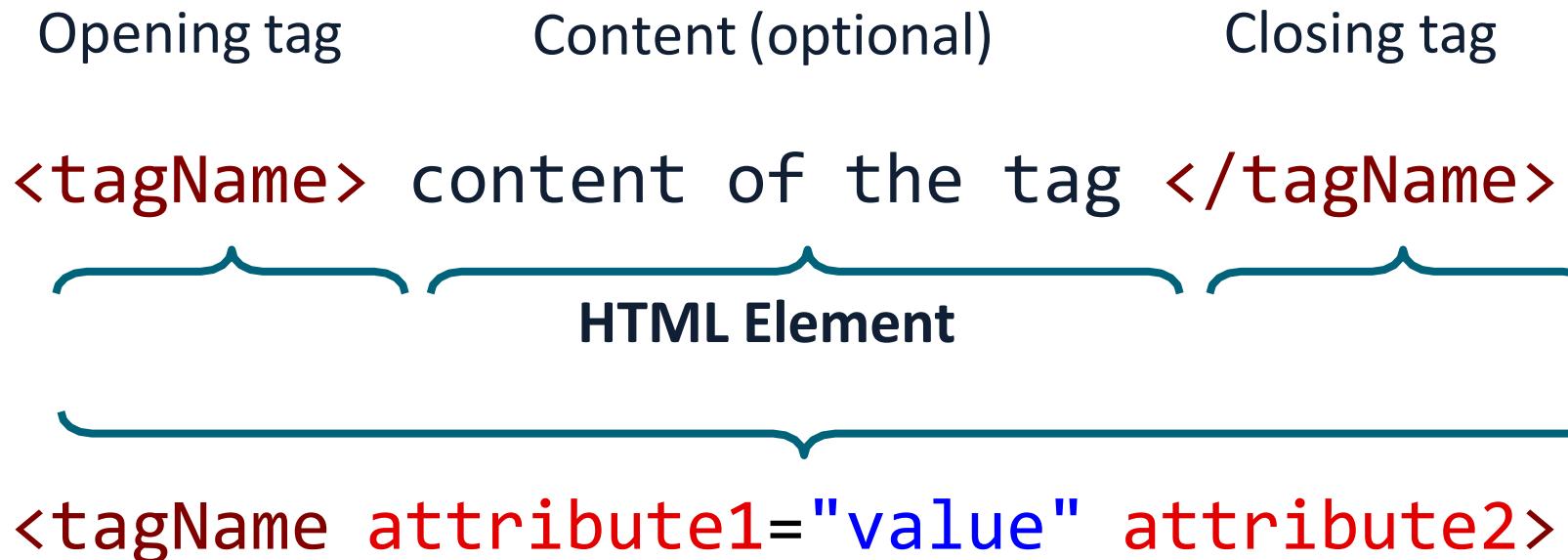


HTML: HYPERTEXT MARKUP LANGUAGE

I browser web mostrano documenti descritti usando HTML, una web app consiste di uno o più documenti (aka pagine web), il concetto chiave ruota attorno ai linguaggi di markup e i documenti sono arricchiti con un insieme di annotazioni per controllarne la struttura, formattazione o la relazione tra le parti.



HTML: HYPERTEXT MARKUP LANGUAGE



In HTML, le annotazioni sono chiamate tag e sono denotati usando le parentesi ad angolo(<>):

- <tagName>: apertura del tag
- content of the tag: contenuto (opzionale) del tag
- </tagName>: chiusura del tag

Tutto questo è un elemento HTML, inoltre i tag possono contenere attributi chiave-valore (il valore è opzionale).

HTML: DOCUMENT STRUCTURE

I documenti HTML devono iniziare con una dichiarazione `<!DOCTYPE HTML>`, non è un tag, dice al client quale tipo di documento deve aspettarsi, invece il tag `<html>` rappresenta l'intero documento. Contiene un `<head>`, che contiene i metadata del documento, e un `<body>`, che contiene i documenti effettivi del documento, si usa lang per incoraggiare l'accessibilità.

```
<!DOCTYPE HTML>
<html lang="en"> ←
  <head>
    <title>Hello World!</title>
  </head>
  <body> <!-- a comment -->
    <p>Hello World!</p>
  </body>
</html>
```

lang attribute is encouraged for **accessibility**

<head> contains document **metadata**

<body> contains the actual document contents

THE HEAD ELEMENT

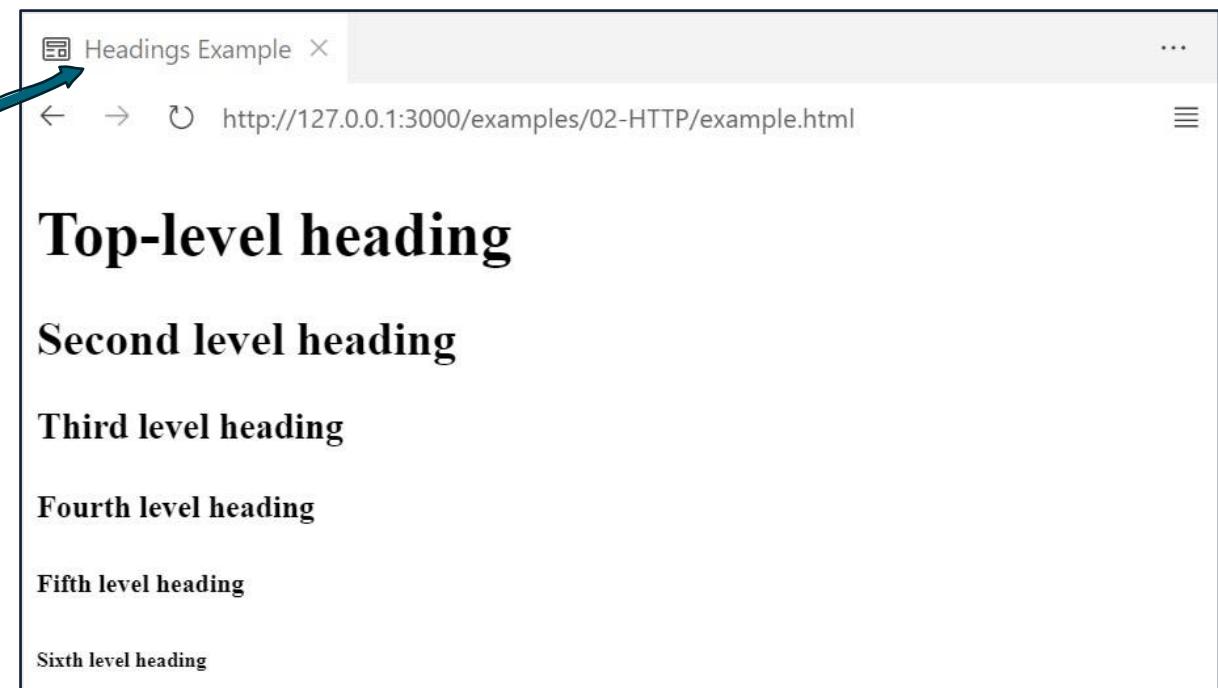
L'elemento head è un container dei metadata, che sono dati sui dati, ad esempio dati sull'attuale documento. Solitamente non sono mostrati all'utente, ma sono utili al browser e ai motori di ricerca, è richiesto il contenere un <title>.

```
<head>
  <title>The Book of Programming</title>
  <meta charset="UTF-8">
  <meta name="description" content="Fragments from the Book of Programming">
  <meta name="keywords" content="Wisdom, programming">
  <meta name="author" content="L. L. L. Starace">
  <meta http-equiv="refresh" content="30">
</head>
```

CORE HTML ELEMENTS: HEADINGS

- Da `<h1>` ad `<h6>` c'è una rappresentazione di titoli o sottotitoli.

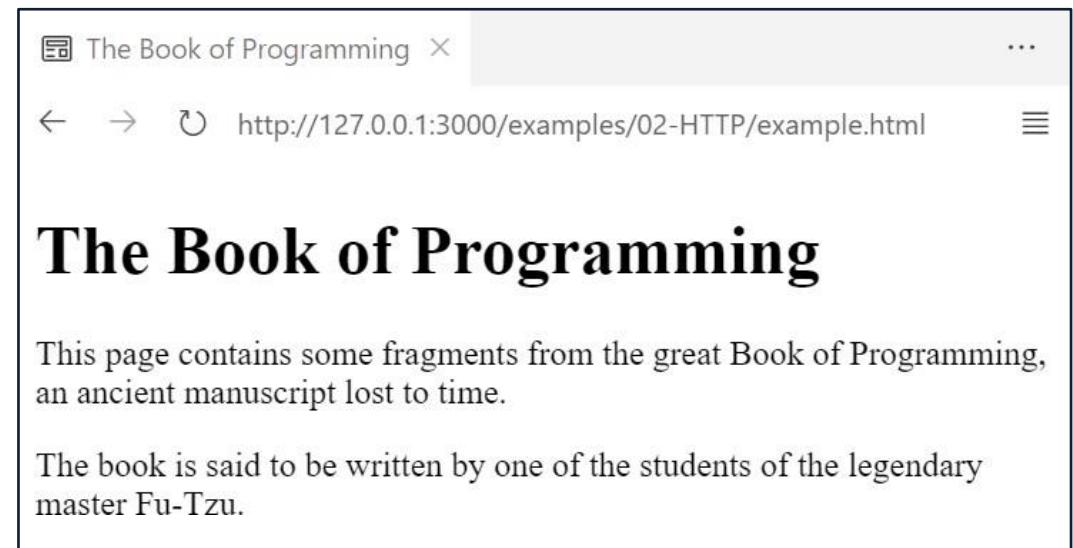
```
<!DOCTYPE HTML>
<html lang="en">
<head>
  <title>Headings Example</title>
</head>
<body>
  <h1>Top-level heading</h1>
  <h2>Second level heading</h2>
  <h3>Third level heading</h3>
  <h4>Fourth level heading</h4>
  <h5>Fifth level heading</h5>
  <h6>Sixth level heading</h6>
</body>
</html>
```



CORE HTML ELEMENTS: PARAGRAPHS

- <p> rappresenta i paragrafi e iniziano solitamente a capo.

```
<h1>The Book of Programming</h1>
<p>
    This page contains some fragments
    from the great Book of Programming,
    an ancient manuscript lost to time.
</p>
<p>
    The book is said to be written by
    one of the students of the legendary
    master Fu-Tzu.
</p>
```



CORE HTML ELEMENTS: COMMENTS

I commenti sono ignorati dal browser e iniziano con <!-- e finiscono con --> e può essere utile aggiungere note o nascondere temporaneamente il contenuto.

```
<h1>The two aspects</h1>
<!-- TODO: add more wisdom -->
<p>
  Below the surface of the machine,
  the program moves. Without effort,
  it expands and contracts.
</p>
<!--
<p>
  The forms on the monitor are but
  ripples on the water. The
  essence stays invisibly below.
</p> -->
```



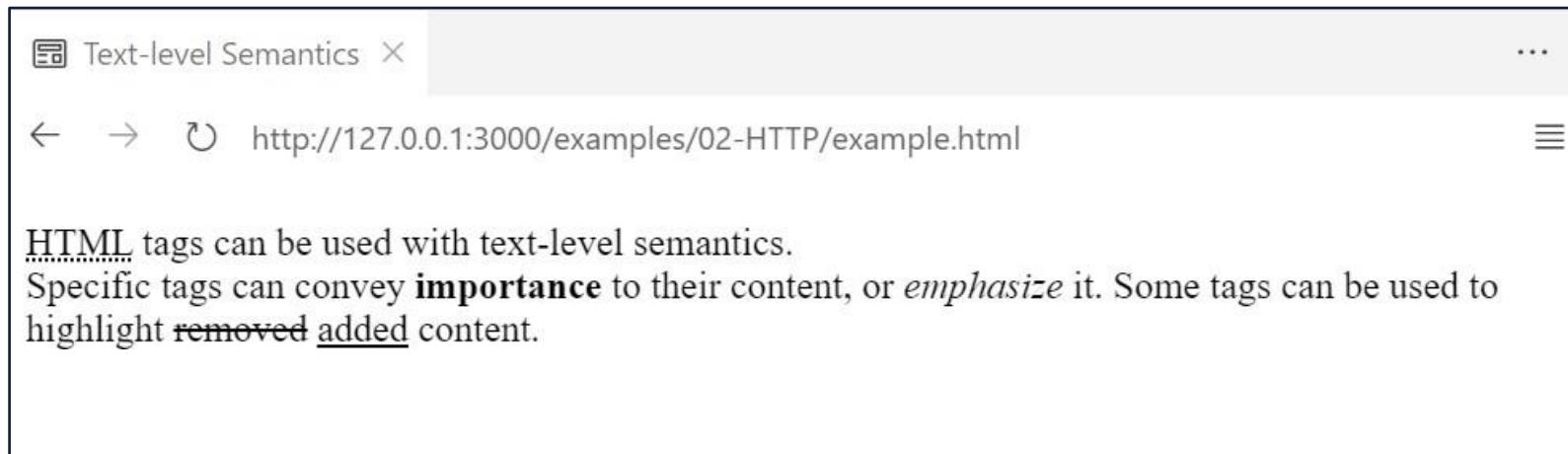
CORE HTML ELEMENTS: TEXT SEMANTICS

I tag possono specificare la semantica a livello del testo:

- : aggiunge enfasi al contenuto
- : rappresenta una **forte** importanza (aggiunge il grassetto alla parola)
-
: passa alla riga successiva
- <abbr title="description">: definisce acronimi e abbreviazioni
- : è il contenuto che è stato eliminato dal documento (riga che passa attraverso al contenuto)
- <ins>: è il contenuto che è stato inserito nel documento (riga che sottolinea il contenuto)

CORE HTML ELEMENTS: TEXT SEMANTICS

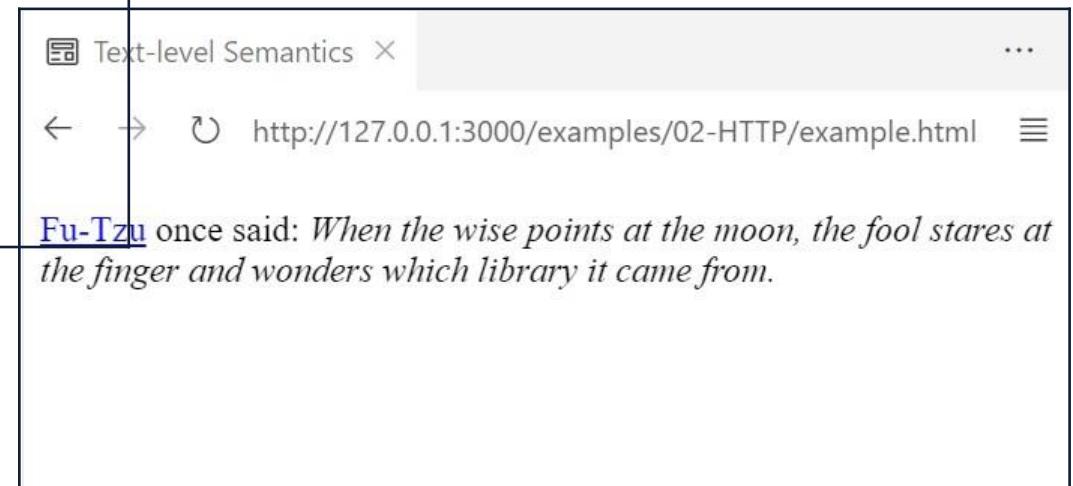
```
<p>
  <abbr title="HyperText Markup Language">HTML</abbr> tags can be used with
  text-level semantics.<br/> Specific tags can convey <strong>importance</strong>
  to their content, or <em>emphasize</em> it.
  Some tags can be used to highlight <del>removed</del> <ins>added</ins> content.
</p>
```



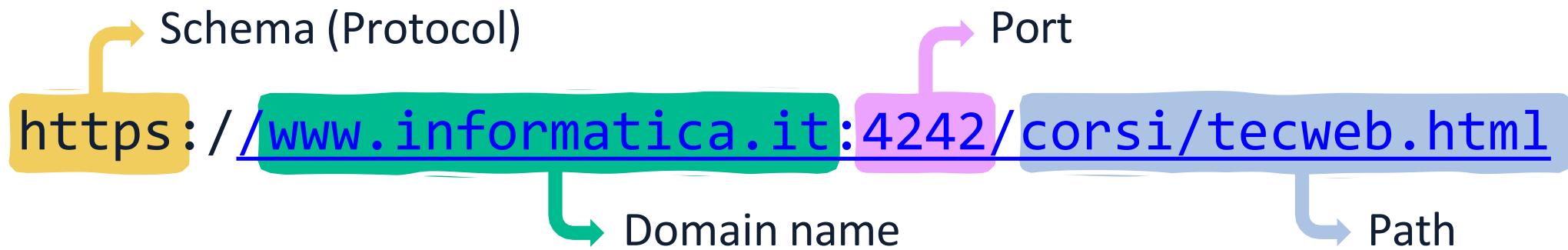
CORE HTML ELEMENTS: ANCHORS

Gli iperlink possono essere definiti usando il tag di ancoramento `<a>`, l'attributo `href` può essere usato per puntare al target URL.

```
<p>
  <a href="/fu-tzu.html">Fu-Tzu</a> once said:
  <em>
    When the wise points at the moon, the
    fool stares at the finger and wonders
    which library it came from.
  </em>
</p>
```



A LOOK BACK ON URLs



ANCHORS: RELATIVE VS ABSOLUTE URLs

Gli URL sono specificati da href e possono essere assoluti o relativi. Gli URL assoluti includono schemi e hostname, e contengono tutte le informazioni necessarie per raggiungere la risorsa.

Ad esempio:

Gli URL relative specificano solo il percorso. Lo schema e il nome dell'host dipendono dall'attuale contesto.

Ad esempio: or

ANCHORS: RELATIVE URLs

Quando un URL relative inizia con un “/”, l’intero percorso è rimpiazzato. Se il contesto corrente è la pagina: <http://bookofprogramming.com/fu-tzu/fu-tzu.html>, un’arcora come `` punta a <http://bookofprogramming.com/index.html>.

ANCHORS: RELATIVE URLs

Quando un URL relative non inizia con “/”, solo l’ultima parte del percorso è rimpiazzata. Se il contesto corrente è alla pagina: <http://bookofprogramming.com/fu-tzu/fu-tzu.html>, un’ancora come `` punta a: <http://bookofprogramming.com/fu-tzu/pic.jpg>.

ANCHORS: DOT SEGMENTS IN URLs

Gli URL relativi possono anche contenere segmenti «dot»: «.» e «..». Il punto da solo rappresenta la directory corrente, il doppio punto rappresenta la directory genitore.

- Assume the current path is `/a/b/c/hello.html`

HREF	RESULTING PATH
<code>./index.html</code>	<code>/a/b/c/index.html</code>
<code>../foo.html</code>	<code>/a/b/foo.html</code>
<code>../../pic.jpg</code>	<code>/a/pic.jpg</code>
<code>../../..pic.jpg</code>	<code>/a/pic.jpg (same as above)</code>

ANCHORS: RELATIVE OR ABSOLUTE URLs?

- Should we use **relative** or **absolute** URLs?

Back to [homepage](#)



Back to `homepage`

Back to `homepage`

Gli URL relativi dovrebbero essere preferiti quando si collegano risorse nella stessa web-app, in questo modo, se il nome dell'host cambia non c'è necessità di cambiare le page html.

Quando si collegano pagine web o risorse esterne, non c'è altra scelta se non gli URL assoluti.

ANCHORS: TARGET ATTRIBUTE

L'attributo target può essere usato per specificare dove aprire la risorse collegata, `` dovrebbe essere aperta nello stessa finestra/tab del browser che la pagina corrente (questo è il comportamento standard, se non viene specificato l'attributo target). `` dovrebbe essere aperta in una nuova finestra/tab del browser.

ON URLs AND INDEX.HTML

- What happens when a URL points to a **directory** and not to a file?

I server web tipicamente rispondono con il contenuto del file index.html, se esiste nella directory richiesta.

Questo comportamento è configurabile:

- Altri filenames di default usati includono: home.html, defual.html
- Se non c'è il file di default, i server HTTP possono essere configurati automaticamente per generare un indice per la directory.



A screenshot of a Firefox browser window showing the contents of a directory. The address bar shows 'Index of /02-HTML/' and the URL '127.0.0.1:3000/02-HTML/'. The page title is 'Index of /02-HTML/'. The table lists files and their details:

Name	Size	Date Modified
..		
comments.html	406.0 B	8/10/23, 7:40:49
divs.html	251.0 B	8/09/23, 11:26:52
forms.html	2.4 kB	8/11/23, 8:18:06
fu-tzu.html	432.0 B	8/09/23, 8:41:24
image.html	288.0 B	8/09/23, 12:10:43
lists.html	454.0 B	8/09/23, 11:27:27
pic.jpg	107.3 kB	8/09/23, 8:45:14
semantics.html	786.0 B	8/09/23, 11:26:53
special.html	134.0 B	8/09/23, 11:26:55
tree.html	308.0 B	8/11/23, 8:49:22

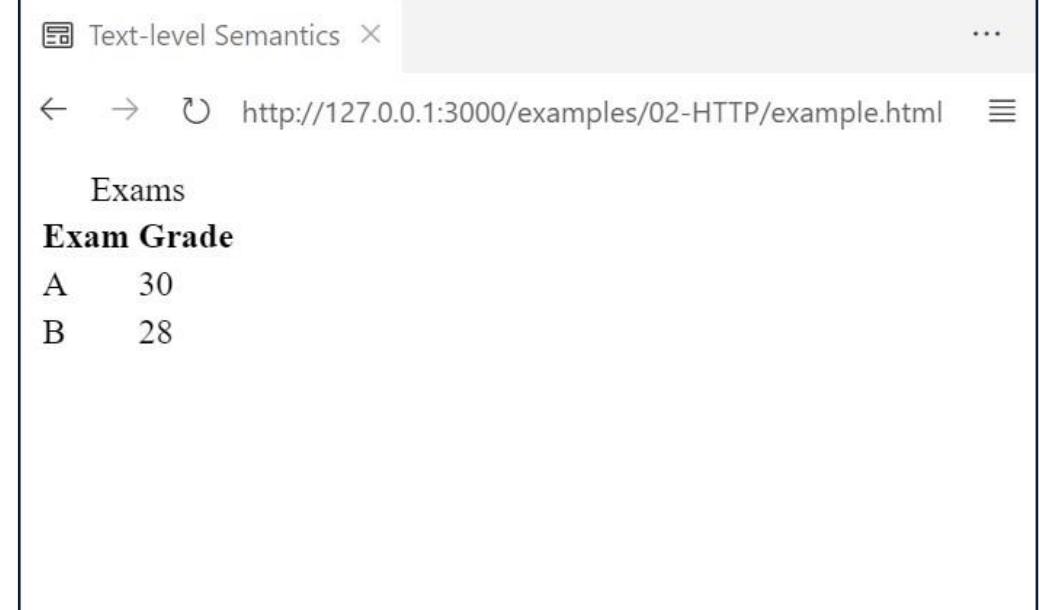
CORE HTML ELEMENTS: TABLES

Una <table> contiene un insieme di <tr> (righe della tabella), ogni riga può contenere una o più: <td> (celle di dati della tabella) e <th> (titoli della tabella). <td> e <th> contengono i valori da mostrare nelle rispettive celle.

Una <table> può anche contenere una <caption> che la descrive.

CORE HTML ELEMENTS: TABLES

```
<table>
  <caption>Exams</caption>
  <tr>
    <th>Exam</th><th>Grade</th>
  </tr>
  <tr>
    <td>A</td><td>30</td>
  </tr>
  <tr>
    <td>B</td><td>28</td>
  </tr>
</table>
```



CORE HTML ELEMENTS: LISTS

HTML definisce 3 tipi di liste:

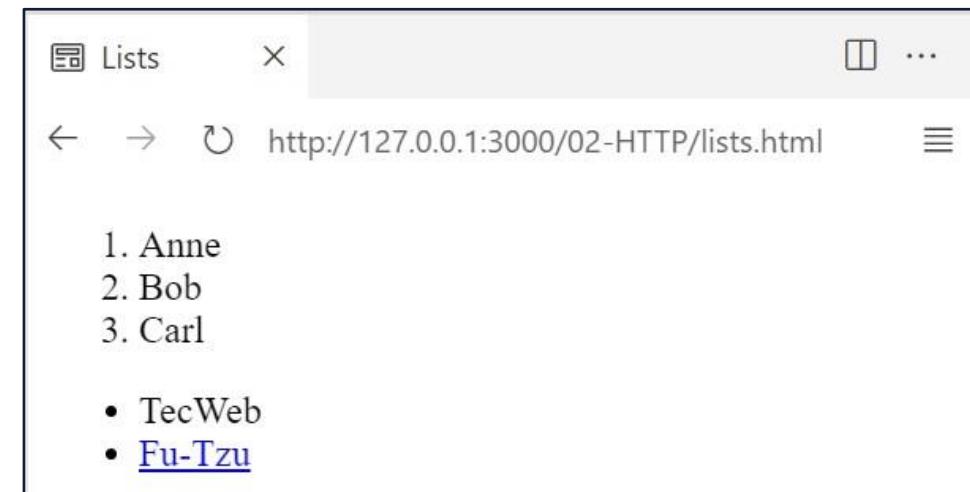
- Ordinate , usate per le enumerazioni
- Non-ordinate , usate per le liste col punto
- Liste descrittive <dl>, consistono di termini e delle loro descrizioni, spesso usate per i glossari

CORE HTML ELEMENTS: (UN)ORDERED LISTS

- Le liste numerate e non contengono una sequenza di elementi della lista

```
<ol>
  <li>Anne</li>
  <li>Bob</li>
  <li>Carl</li>
</ol>

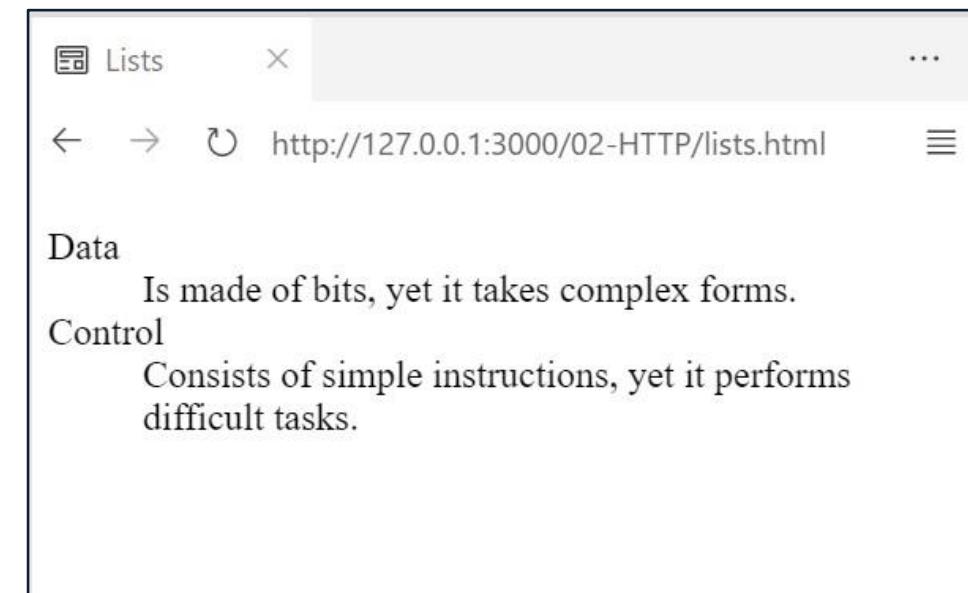
<ul>
  <li>TecWeb</li>
  <li><a href="/">Fu-Tzu</a></li>
</ul>
```



CORE HTML ELEMENTS: DESCRIPTION LISTS

Le `<dl>` contengono una sequenza di termini `<dt>` e descrizione dei termini `<dd>`.

```
<dl>
  <dt>Data</dt>
  <dd>
    Is made of bits,
    yet it takes complex forms.
  </dd>
  <dt>Control</dt>
  <dd>
    Consists of simple instructions,
    yet it performs difficult tasks.
  </dd>
</dl>
```



CORE HTML ELEMENTS: ENTITIES

- Alcuni caratteri sono riservati ad HTML
- What if we want to write in a document:
- <p>3<x and y>6</p>? //3<x and y>6

Il browser potrebbe mischiare i simboli con i tag.

Le Character entities dovrebbero essere usato per mostrare i caratteri riservati e hanno la seguente forma:

&entity_name; or *&#entity_number;*

SOME USEFUL HTML ENTITIES

Result	Description	Entity Name
	Non-breaking space	
<	Less than	<
>	Greater than	>
&	Ampersand	&
"	Double quote	"
'	Single quote (apostrophe)	'
©	Copyright	©

The example in the previous slide should be: <p>3<x and y>6</p>

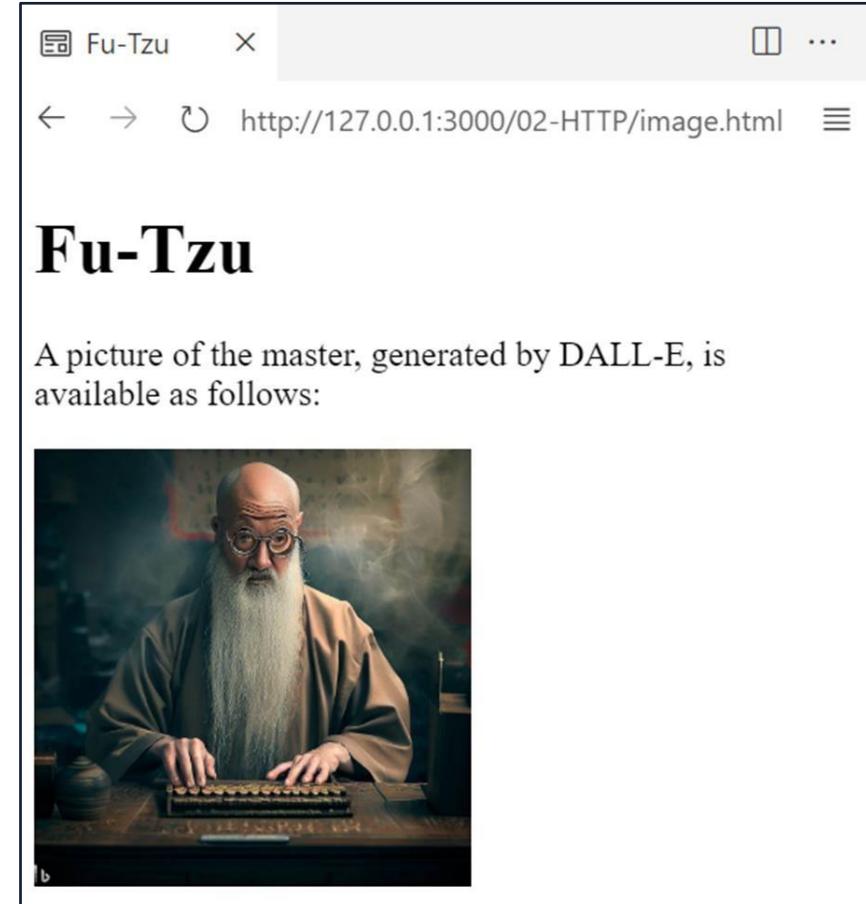
CORE HTML ELEMENTS: IMAGES

 è usato per fissare un'immagine a un documento HTML, è un elemento void (e non dovrebbe contenere altri elementi). L'attributo src specifica l'URL dell'immagine da includere, l'attributo alt specifica un testo descrittivo alternativo per l'immagine. La larghezza e l'altezza possono essere usati per specificare la dimensione (in pixel) dell'immagine fissata nella pagina web.

CORE HTML ELEMENTS: IMAGES

```
<h1>Fu-Tzu</h1>
<p>
    A picture of the master,
    generated by DALL-E, is
    available as follows:
</p>

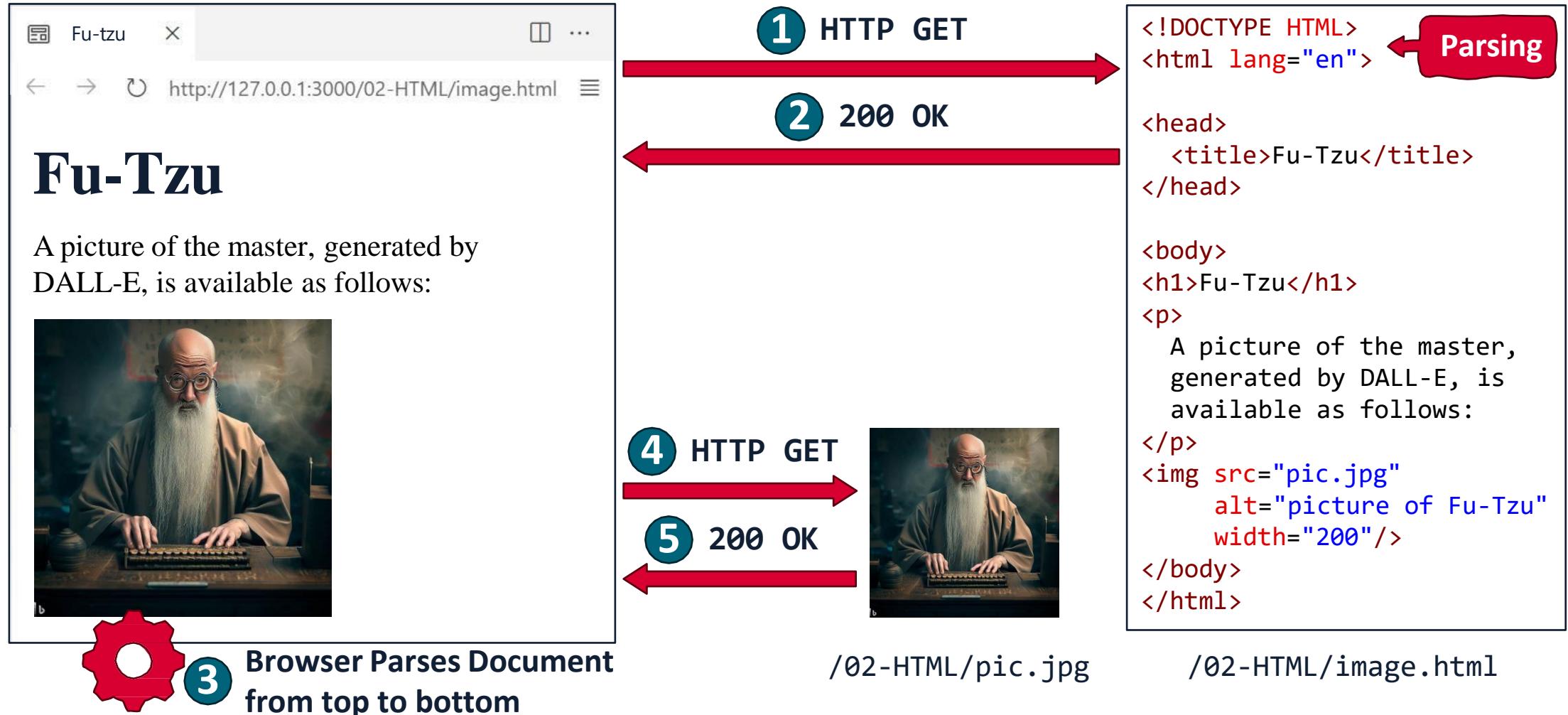
```



IMAGES: BEHIND THE SCENES

Qualcosa di nuovo (e abbastanza interessante) sta per succedere. Finora i documenti HTML erano interamente auto-contenuti. Tutti i dati nel documento erano nel e del documento. Abbiamo i link, ma non siamo liberi di navigarli e l'ultimo esempio di pagina web includeva del contenuto esterno (l'immagine) provveduto solo dal suo URL. L'immagine in sè non è inclusa nel documento HTML, per visualizzare il documento, il browser ha bisogno di richiamare (fetch) risorse extra!

IMAGES: BEHIND THE SCENES



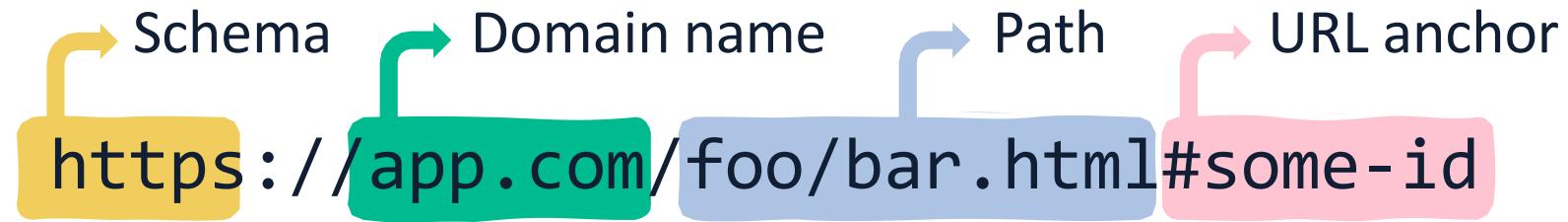
CORE HTML ELEMENTS: GLOBAL ATTRIBUTES

Abbiamo visto alcuni attributi (e.g.: href, target, src, alt), che sono significati solo per alcuni elementi, infatti `<strong src="pic.jpg">Hi!` non ha alcun senso!

Alcuni attributi sono globali, ad esempio: possono essere usati con ogni elemento HTML.

Global Attr.	Description
<code>id</code>	Specifies a unique (in the document) identifier for an element
<code>lang</code>	Specifies the language for the element's content
<code>style, class</code>	Used for styling (we'll see in the next lecture)

URLS: ANCHORS



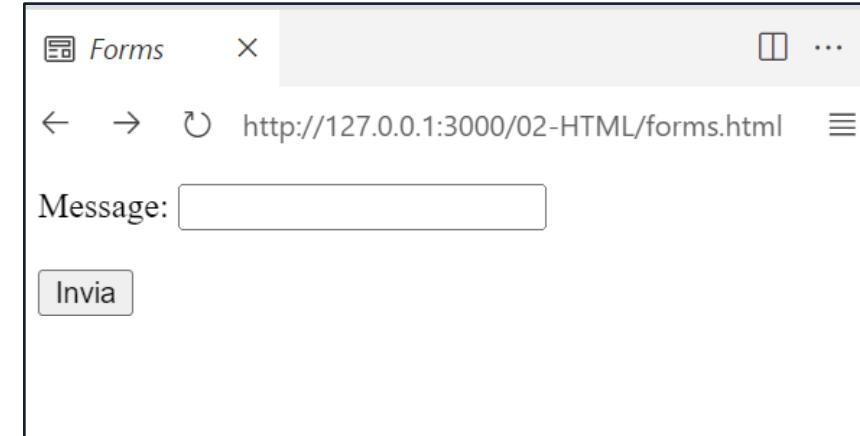
L'attributo id nell'elemento html può essere usato nelle ancora URL, un'ancora rappresenta una sorta di “segnalibro” all'interno della risorsa, usato per dire ai browser di mostrare il contenuto locato in quel luogo segnato. Nell'esempio `#some-id` è un'ancora che punta a una specifica parte della risorsa stessa, nominativamente l'elemento con id: `some-id`.

- E.g.: <https://luistar.github.io/publications/#conference>

CORE HTML ELEMENTS: FORMS

Gli elementi <form> sono usati per ricevere input dall'utente, l'input è tipicamente inviato a un server per processarlo. I form contengono elementi form come: <input>, <label>, <textarea>, <select>, <form>.

```
<form>
  Message:
  <input type="text"><br><br>
  <input type="submit">
</form>
```



FORMS: INPUTS

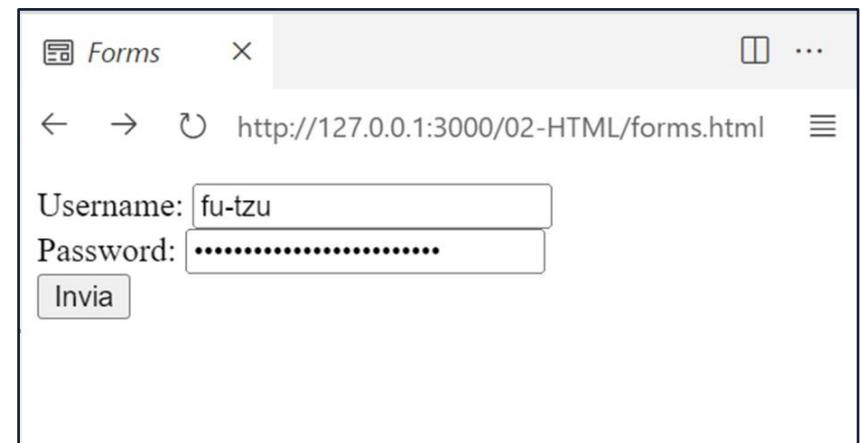
- Different kinds of input elements are available via the **type** attribute
- Some supported types include:

Type	Description
<code><input type="text"></code>	Displays a single-line text input field
<code><input type="password"></code>	Displays an input field for passwords (input is hidden with *****)
<code><input type="number"></code>	Displays an input for numbers
<code><input type="radio"></code>	Displays a radio button (for selecting one of many choices)
<code><input type="checkbox"></code>	Displays a checkbox (for selecting zero or more of many choices)
<code><input type="button"></code>	Displays a clickable button

FORMS: LABELS

<label> può essere usato per etichettare ogni elemento da inserire. Facendone uso si migliora l'usabilità e l'accessibilità. L'attributo for dei label dovrebbe essere uguale all'id dell'input corrispondente.

```
<form>
  <label for="id_usr">Username: </label>
  <input type="text" name="usr" id="id_usr">
  <br/>
  <label for="id_pwd">Password: </label>
  <input type="password" name="pwd" id="id_pwd">
  <br/>
  <input type="submit">
</form>
```



FORMS: SUBMISSION

I form possono essere inviati per collezionare dati di qualche gestore dei form. Oltre alle submissions, una nuova richiesta HTTP viene tipicamente performata, l'URL del gestore del form, a cui viene inviata la richiesta, è specificata dall'attributo d'azione del form (solitamente è la stessa pagina del form). È anche possibile specificare il metodo HTTP da usare, in base all'attributo del metodo (impostato di base su GET).

```
<form action="/handler.html" method="GET">
    <!-- form elements here -->
</form>
```

FORMS: SUBMISSION

Oltre alla submission, l'input dell'user collezionato è rappresentato in una serie di coppie nome/valore del form name1=value1&...&nameN=valueN. Ogni nome è il nome di un elemento d'input e il suo corrispettivo valore al momento della sottomissione.

```
<form action="/handler.html" method="GET">
  Message: <input type="text" name="msg"><br>
  Number: <input type="number" name="num"><br>
  <input type="submit">
</form>
```

msg=Hello!&num=42

Forms ...
← → ⚡ http://127.0.0.1:3000/02-HTML/forms.html ⏺
Message: Hello!
Number: 42
Invia

FORMS: SUBMISSION WITH GET

Se il metodo è GET, gli input sono appesi all'URL del gestore, l'URL della richiesta è:
`/handler.html?msg=Hello!&num=42`, dove `?msg=Hello!&num=42` è chiamato query string.

```
<form action="/handler.html" method="GET">
  Message: <input type="text" name="msg"><br>
  Number: <input type="number" name="num"><br>
  <input type="submit">
</form>
```

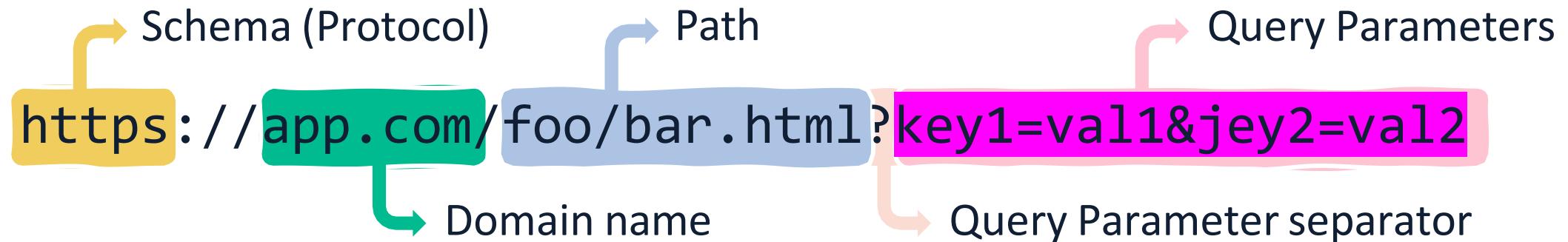


A screenshot of a web browser window titled "Forms". The address bar shows the URL `http://127.0.0.1:3000/02-HTML/forms.html`. Below the address bar is a form with two text inputs. The first input is labeled "Message" and contains the value "Hello!". The second input is labeled "Number" and contains the value "42". At the bottom of the form is a button labeled "Invia".

**msg and num are called
query parameters**

```
GET /handler.html?msg=Hello!&num=42 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0
Accept: text/plain
Accept-Language: en-us
Connection: keep-alive
```

URLS: QUERY PARAMETERS



I parametri query sono parametri extra che provveduti dal server. Questi parametri sono una lista di coppie chiave/valore separate con “&”, il server web può essere usato da questi parametri per fare cose extra restituendo la risorse.

- E.g.: <https://search-engine.com/search?q=web+technologies&lang=en>

FORMS: SUBMISSION WITH POST

Se il metodo è POST, gli input sono inviati al body di chi richiede. L'URL della richiesta è: /handler.html.

```
<form action="/handler.html" method="POST">
  Message: <input type="text" name="msg"><br>
  Number: <input type="number" name="num"><br>
  <input type="submit">
</form>
```

A screenshot of a web browser window titled "Forms". The address bar shows the URL "http://127.0.0.1:3000/02-HTML/forms.html". The page contains a form with two inputs: one for "Message" with the value "Hello!" and one for "Number" with the value "42". Below the inputs is a button labeled "Invia".

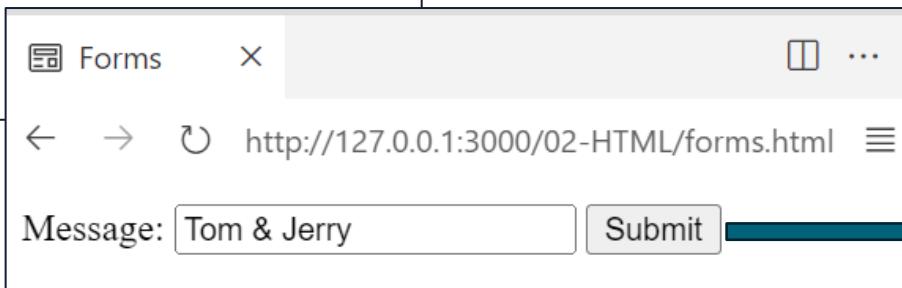
```
GET /handler.html HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0
Accept: text/plain
Accept-Language: en-us
Connection: keep-alive
```

msg=Hello!&num=42

URL ENCODING

L'input del form è codificato come una stringa di chiave-valori separati da "&", se l'utente inserisse caratteri speciali (come &) verrebbero rimpiazzati da tiplette del carattere nella forma %XX, dove XX sono 2 digit esadecimali che rimpiazzano il carattere in ASCII, gli spazi sono rimpiazzati dal %20 o dal simbolo +.

```
<form>
  <label for="msg">Message:</label>
  <input id="msg" name="msg" type="text">
  <input type="submit">
</form>
```



The screenshot shows a browser window with the URL `http://127.0.0.1:3000/02-HTML/forms.html`. Inside, there's a form with a label "Message:" and a text input field containing "Tom & Jerry". A "Submit" button is at the bottom. A blue arrow points from the browser screenshot up towards the URL "msg=Tom+%26+Jerry".

?

URL ENCODING: ASCII TABLES

Dec	Hex	Char	Description	Dec	Hex	Char	Description
32	20	space	Space	43	2B	+	Plus
33	21	!	Exclamation mark	44	2C	,	Comma
34	22	"	Double quote	45	2D	-	Minus
35	23	#	Number	46	2E	.	Period
36	24	\$	Dollar sign	47	2F	/	Slash
37	25	%	Percent	91	5B	[Left square bracket
38	26	&	Ampersand	92	5C	\	Backslash
39	27	'	Single quote	93	5D]	Right square brack.
40	28	(Left parenthesis	94	5E	^	Caret / circumflex
41	29)	Right parenthesis	95	5F	_	Underscore
42	2A	*	Asterisk	96	60	`	Grave / accent

MORE INPUTS: CHECKBOX

```
<form>
  <p>Which exams will you take?</p>
  <input type="checkbox" name="exams" value="web" id="web_tech">
  <label for="web_tech">Web Technologies</label><br>
  <input type="checkbox" name="exams" value="pl2" id="pl2">
  <label for="pl2">Programming Languages II</label><br><br>
  <input type="submit">
</form>
```

Forms

← → ⚡ http://127.0.0.1:3000/02-HTML/forms.html ⏹

Which exams will you take?

Web Technologies

Programming Languages II

Invia

On submit → "exams=web"

MORE INPUTS: CHECKBOX

```
<form>
  <p>Which exams will you take?</p>
  <input type="checkbox" name="exams" value="web" id="web_tech">
  <label for="web_tech">Web Technologies</label><br>
  <input type="checkbox" name="exams" value="pl2" id="pl2">
  <label for="pl2">Programming Languages II</label><br><br>
  <input type="submit">
</form>
```

The screenshot shows a browser window titled "Forms" with the URL <http://127.0.0.1:3000/02-HTML/forms.html>. The page content is:

Which exams will you take?

Web Technologies
 Programming Languages II

An arrow labeled "On submit" points from the browser window to the text "exams=web&exams=pl2".

"exams=web&exams=pl2"

MORE INPUTS: CHECKBOX

```
<form>
  <p>Which exams will you take?</p>
  <input type="checkbox" name="exams" value="web" id="web_tech">
  <label for="web_tech">Web Technologies</label><br>
  <input type="checkbox" name="exams" value="pl2" id="pl2">
  <label for="pl2">Programming Languages II</label><br><br>
  <input type="submit">
</form>
```

Forms

← → ⚡ http://127.0.0.1:3000/02-HTML/forms.html ⏹

...

Which exams will you take?

Web Technologies

Programming Languages II

Invia

On submit → "" (empty string)

MORE INPUTS: RADIO BUTTONS

```
<form>
  <p>What's your favourite course?</p>
  <input type="radio" name="fav" value="web" id="web_tech">
  <label for="web_tech">Web Technologies</label><br>
  <input type="radio" name="fav" value="net" id="net">
  <label for="net">Computer Networks</label><br>
  <input type="radio" name="fav" value="se" id="se">
  <label for="se">Software Engineering</label><br><br>
  <input type="submit" value="Submit your opinion">
</form>
```

Forms

What's your favourite course?

Web Technologies
 Computer Networks
 Software Engineering

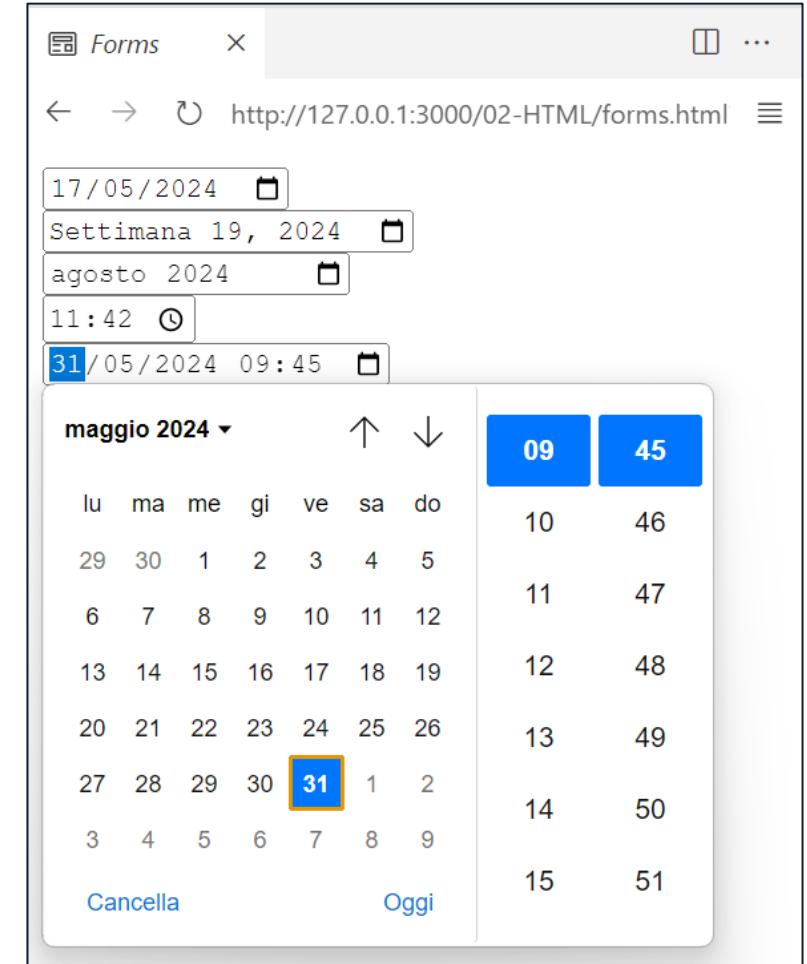
Submit your opinion

On submit → "fav=web"

MORE INPUTS: DATES

Esistono tipi di input dedicati alle date e al tempo.

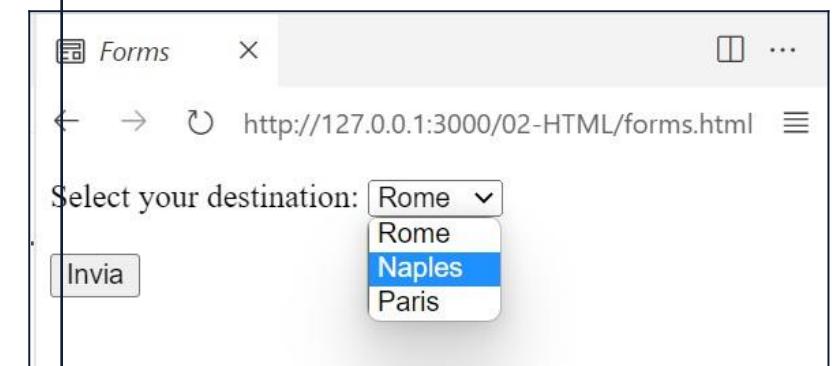
```
<form>
  <input type="date"><br>
  <input type="week"><br>
  <input type="month"><br>
  <input type="time"><br>
  <input type="datetime-local"><br>
</form>
```



MORE INPUTS: SELECT

<select> può essere usato per definire scelte multiple a cascata. Gli attributi multipli possono essere usati per permettere la selezione di più di un'opzione.

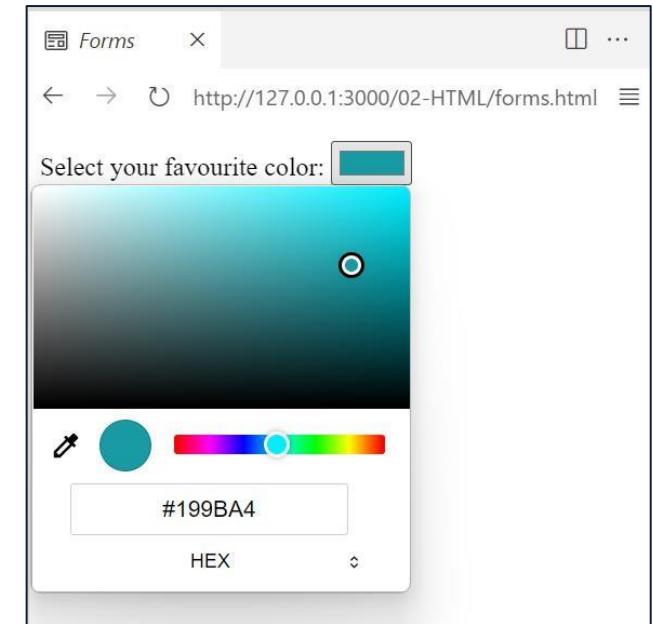
```
<form>
  <label for="dest">Select your destination:</label>
  <select name="destination" id="dest">
    <option value="Rome">Rome</option>
    <option value="Naples">Naples</option>
    <option value="Paris">Paris</option>
  </select><br><br>
  <input type="submit">
</form>
```



THERE'S MORE TO INPUTS

- There's more to inputs! (e.g.: color picker, file picker, datalists...)

```
<form method="POST">
  <label for="col">Select your favourite color:</label>
  <input name="color" id="col" type="color"><br><br>
  <input type="submit">
</form>
```



- Check out [MDN web docs](#) for a complete reference

FORMS: GROUPING INPUTS

```
<form>
  <fieldset>
    <legend>Personal data:</legend>
    <label for="fname">First name:</label><br>
    <input type="text" id="fname" name="fname"><br>
    <label for="lname">Last name:</label><br>
    <input type="text" id="lname" name="lname">
  </fieldset>
  <fieldset>
    <legend>Exam Registration:</legend>
    <label for="grade">Grade:</label><br>
    <input type="number" id="grade" name="grade"><br>
    <label for="date">Date:</label><br>
    <input type="date" id="date" name="date">
  </fieldset><br>
  <input type="submit" value="Submit">
</form>
```

Personal data:

First name:
John

Last name:
Doe

Exam Registration:

Grade:
30

Date:
30/09/2024

Submit

fname=John&lname=Doe&grade=30&date=2024-09-30

GROUPING AND ORGANIZING CONTENT

Il contenuto di una pagina web può essere organizzato in parti, usando le divisioni come `<div>` o con tag semantici come `<header>`, `<nav>`, `<main>`, `<article>`, `<section>`, `<aside>`, `<footer>` , e altri.

DIVISIONS

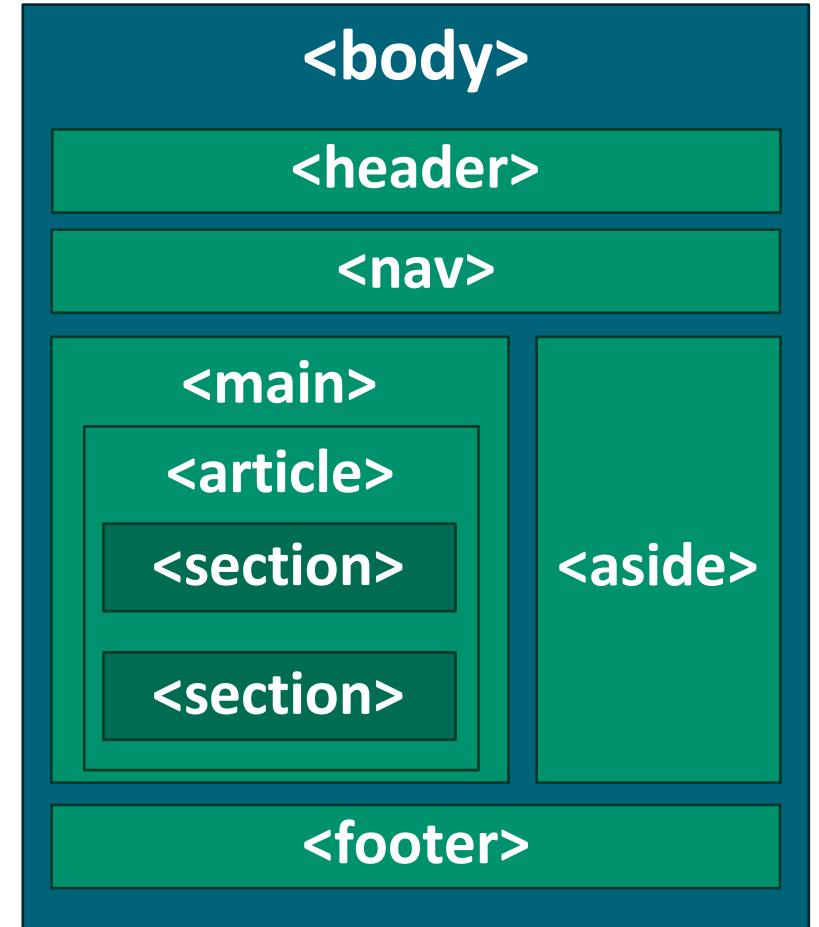
Le <div> erano il modo principale di raggruppare il contenuto in modo ordinato, questo finché non sono stati introdotti i tag semantici. Non supportano alcun tipo di semantica specifica, oltre che raggruppare i contenuti sono in qualche modo relazionati tra di loro.

```
<div>
    <h1>Section 1</h1>
    <p>Content of Section 1</p>
</div>
<div>
    <h1>Section 2</h1>
    <p>Content of Section 2</p>
</div>
```



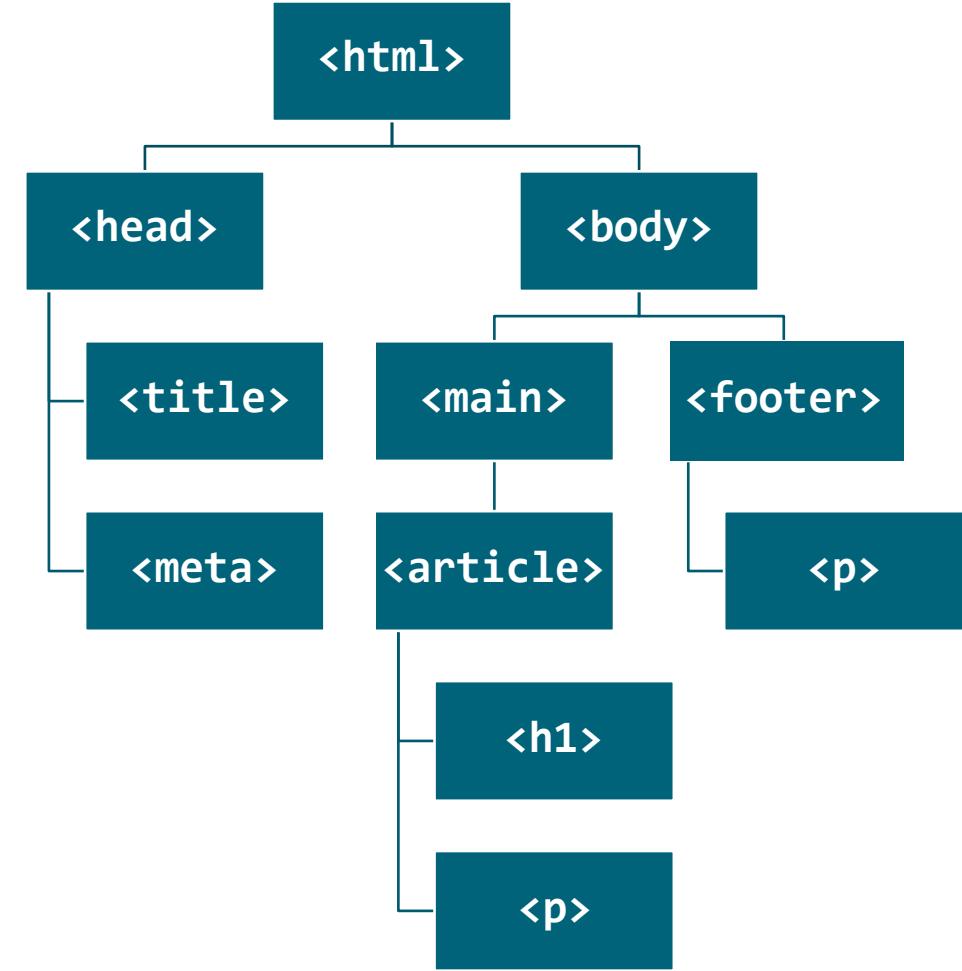
SEMANTIC TAGS

Descrivono il significato del loro contenuto al browser, developer e software. I `<nav>` contengono i link da navigare. I `<main>` indica il contenuto principale. `<article>` è usato per il contenuto indipendente e autocontenuto. `<aside>` è usato per essere al lato del contenuto a cui è legato. `<header>`, `<footer>`, `<section>` sono autoesplicativi



HTML DOCUMENTS AS TREES

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>The Book of Programming</title>
  <meta charset="UTF-8">
</head>
<body>
  <main>
    <article>
      <h1>Title</h1>
      <p>Body</p>
    </article>
  </main>
  <footer>
    <p>&copy; Web Technologies 2024</p>
  </footer>
</body>
</html>
```



```
<header>
  <h1>Web Technologies!</h1>
  <p>This page contains some contents on Web Technologies.</p>
</header>
<main>
  <article>
    <h2>Semantic elements are good</h2>
    <p>Let's discuss semantic elements.</p>
    <section>
      <h3>Pros</h3>
      <p>They convey more information.</p>
    </section>
    <section>
      <h3>Cons</h3>
      <p>Literally none.</p>
    </section>
  </article>
  <article>
    <h2>HTML is nice</h2>
    <p>And you ain't seen styling and scripting yet!</p>
  </article>
</main>
<footer> © Web Technologies course, 2024. </footer>
```



The screenshot shows a browser window with the title "Semantic Elements". The URL in the address bar is "http://127.0.0.1:3000/02-HTTP/semantics.html". The page content is as follows:

Web Technologies!

This page contains some contents on Web Technologies.

Semantic elements are good

Let's discuss semantic elements.

Pros

They convey more information.

Cons

Literally none.

HTML is nice

And you ain't seen styling and scripting yet!

© Web Technologies course, 2024.

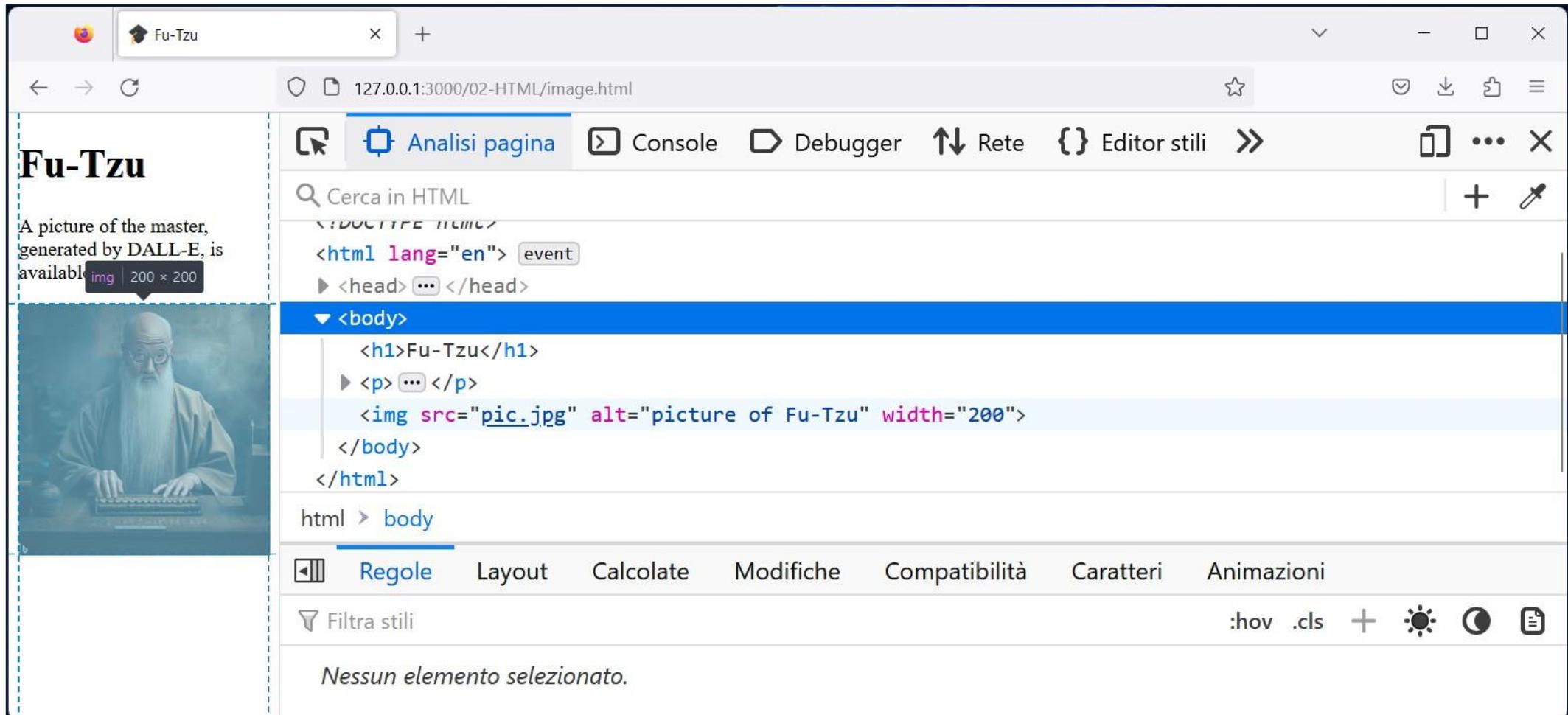
BROWSER DEV TOOLS

BROWSER DEV TOOLS

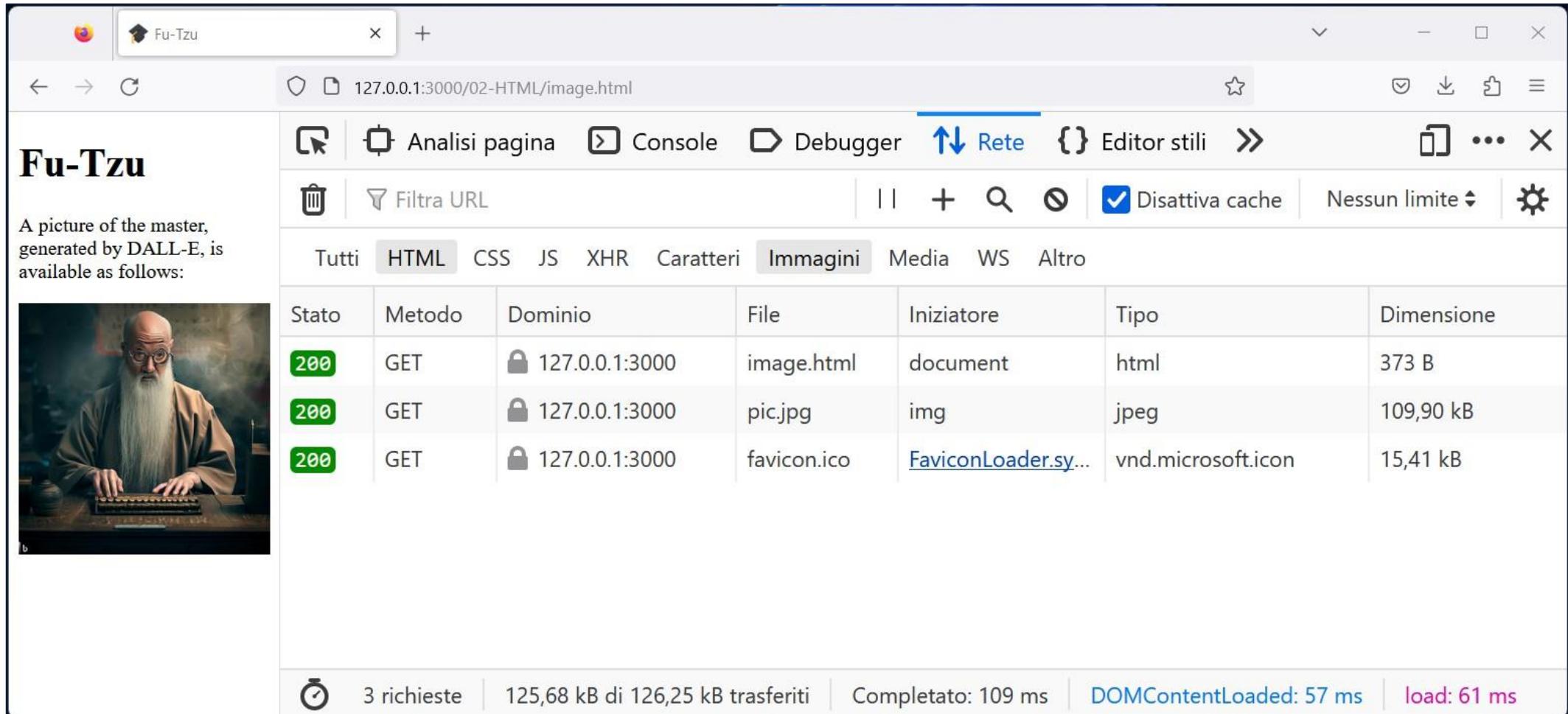
I browser moderni includono molte caratteristiche per supportare i developer web, questi attrezzi per sviluppatori sono acceduti tramite il pulsante F12 e includono:

- La possibilità d'ispezionare i documenti HTML
- L'analisi di rete (e anche di come vanno le richieste/risposte HTTP)
- Profiling (misurare le performance dei tempi di caricamento)
- Debugging sia gli elementi di stile che di scripting

BROWSER DEV TOOLS: INSPECT PAGE



BROWSER DEV TOOLS: NETWORK ANALYSIS



The screenshot shows the Firefox Developer Tools Network tab for the URL `127.0.0.1:3000/02-HTML/image.html`. The Network tab is active, displaying a list of network requests. The requests are:

Stato	Metodo	Dominio	File	Iniziatore	Tipo	Dimensione
200	GET	127.0.0.1:3000	image.html	document	html	373 B
200	GET	127.0.0.1:3000	pic.jpg	img	jpeg	109,90 kB
200	GET	127.0.0.1:3000	favicon.ico	FaviconLoader.js...	vnd.microsoft.icon	15,41 kB

At the bottom of the Network tab, there are performance metrics: 3 richieste, 125,68 kB di 126,25 kB trasferiti, Completato: 109 ms, DOMContentLoaded: 57 ms, and load: 61 ms.

ASSIGNMENT

Today's lecture comes with the very first course assignment!

- The assignment will guide you in setting up a development environment with **VS Code**, including a development HTTP server
- You will build a static website by authoring and linking together HTML documents
- You will work with HTML Forms
- You will learn to deploy a production-grade HTTP server

REFERENCES (1/2)

- **Learn HTML**

web.dev

<https://web.dev/learn/html>

Sections: 1 to 10 and 12 to 14

- **Learn Forms**

web.dev

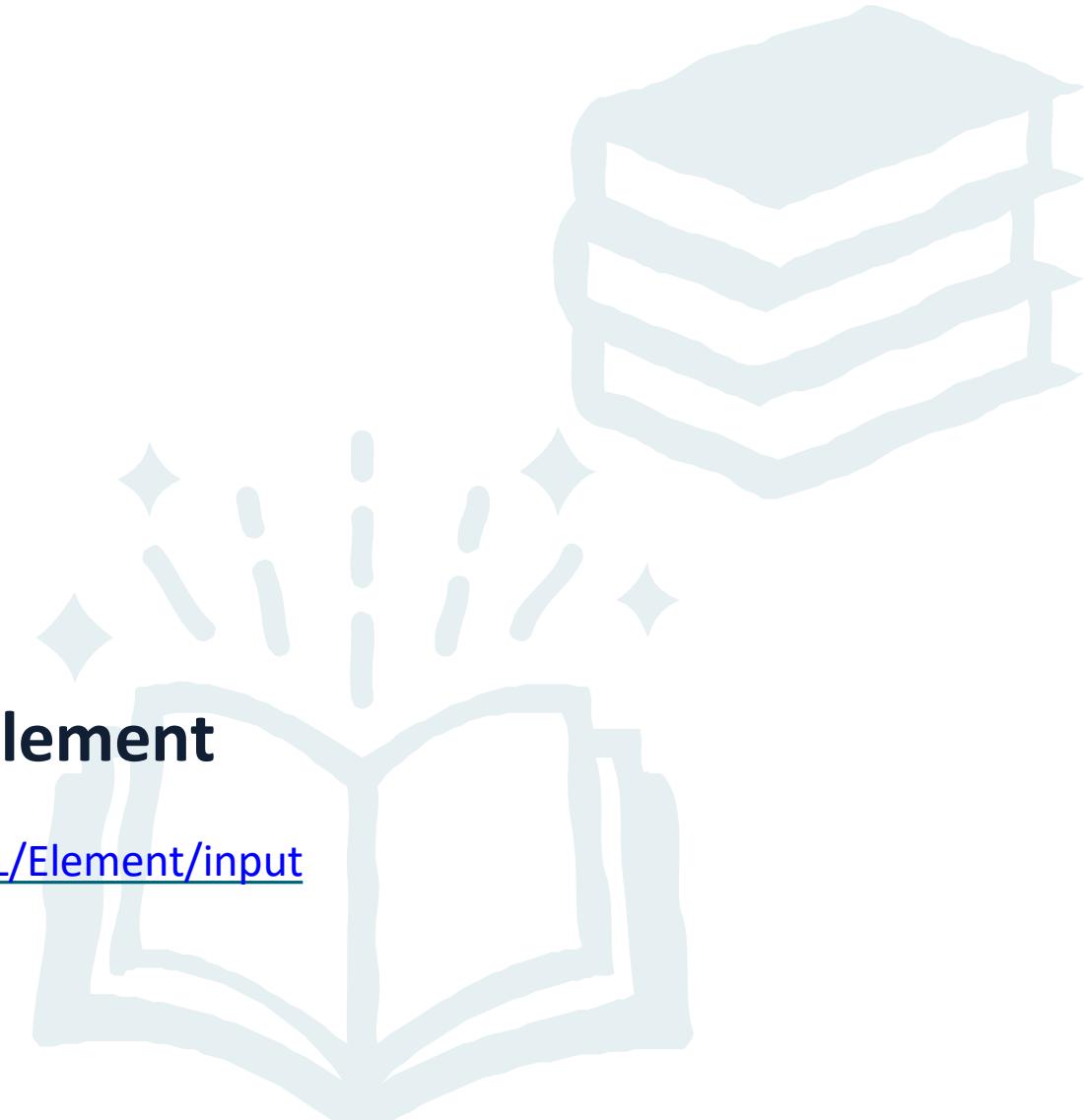
<https://web.dev/learn/forms>

Sections: 1 to 3

- **<input>: The Input (Form Input) element**

MDN web docs

<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input>



REFERENCES (2/2)

- **HTML Forms (overview)**

W3Schools

https://www.w3schools.com/html/html_forms.asp

- **HTML Living Standard**

WHATWG

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

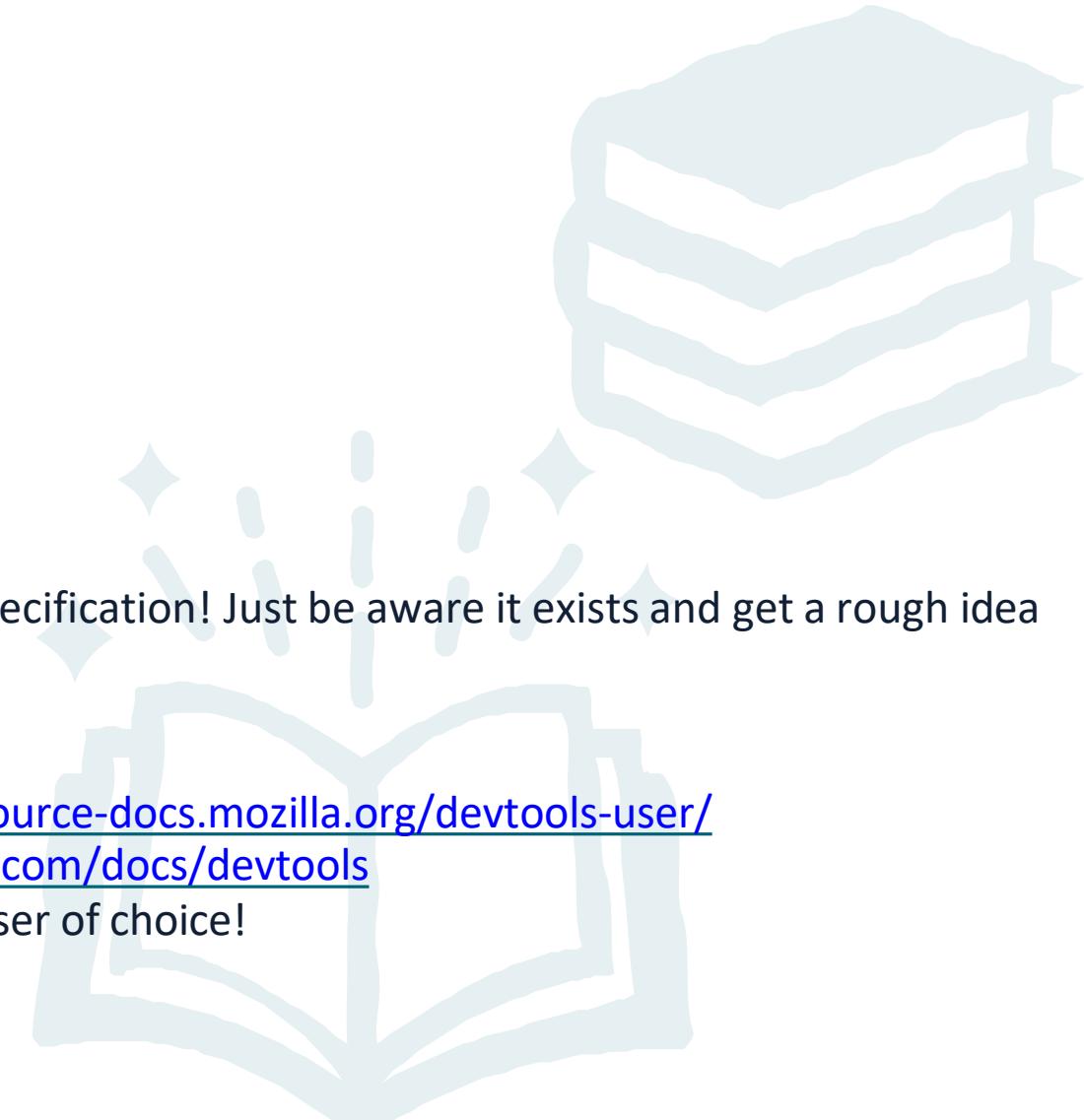
⚠ You are **not** required to learn the entire HTML specification! Just be aware it exists and get a rough idea of how it is structured.

- **Browser DevTools**

Mozilla Firefox DevTools User Docs: <https://firefox-source-docs.mozilla.org/devtools-user/>

Google Chrome DevTools: <https://developer.chrome.com/docs/devtools>

ℹ Get familiar with the DevTools in your web browser of choice!



UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
WEB TECHNOLOGIES – LECTURE 03

CSS: CASCADING STYLE SHEETS

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



PREVIOUSLY, ON WEB TECHNOLOGIES

- We have learned how to write HTML documents
- HTML is concerned with **structure** and **semantics** of documents
- HTML is saying nothing at all on the **appearance** of documents
- An **** element specifies that its content should be emphasized
- It's not saying **how** the emphasizing part should be done
 - Emphasis might be conveyed using *italics*, **different colors** or **backgrounds**.

CSS: CASCADING STYLE SHEETS

- Un linguaggio dichiarativo basato su regole per specificare la modalità di presentazione dei documenti agli utenti.
- Un foglio di stile è un insieme di regole, ciascuna definita come segue
- Il **selettore** specifica quali elementi HTML sono interessati dalla regola
- Le regole contengono un insieme di dichiarazioni, sotto forma di coppie proprietà-valore, che specificano lo stile da applicare

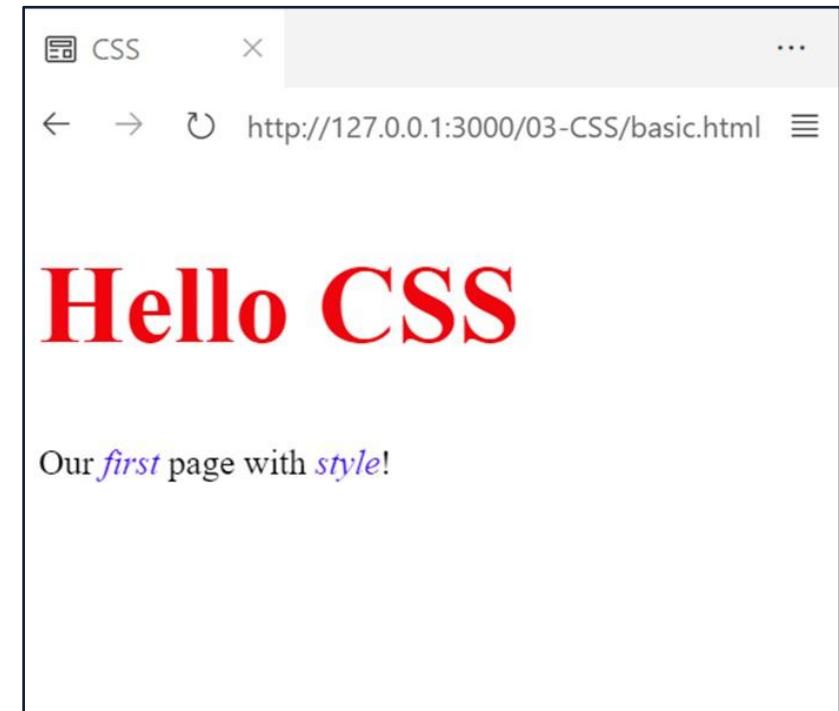
```
selector {  
    property: value;  
    property: value;  
}
```

CSS: FIRST EXAMPLE

```
<h1>Hello CSS</h1>
<p>
  Our <em>first</em> page
  with <em>style</em>!
</p>
```

```
h1 {
  color: red;
  font-size: 50px;
}

em {
  color: blue;
}
```



DEFAULT USER AGENT STYLES

- Ma le prime pagine web che abbiamo sviluppato hanno un po' di stile!
- Le intestazioni sono più grandi e mostrate con una faccia in grassetto...
- <p> inizia su nuove righe, sono visualizzati in corsivo, in grassetto,
- <a> sono sottolineati e blu, hanno punti elenco e così via...
- Questo perché i browser applicano i propri stili di base a ogni pagina!
- Sono spesso indicati come stili di agente utente
- Queste impostazioni predefinite sono più o meno le stesse nei diversi browser, ma esistono alcune differenze

INCLUDING STYLESHEETS IN WEB PAGES

Lo stile può essere incluso nei documenti HTML in diversi modi

Utilizzo <link> degli elementi nel <head> del documento

L'attributo rel="stylesheet" specifica la relazione tra il documento corrente e il documento collegato

L'attributo href="style.css" specifica l'URL del foglio di stile da caricare

Stesso meccanismo di : il browser effettuerà una richiesta HTTP aggiuntiva per recuperare il foglio di stile prima di eseguire il rendering della pagina

```
<head>
  <meta charset="UTF-8">
  <title>CSS</title>
  <link rel="stylesheet" href="style.css">
</head>
```

INCLUDING STYLESHEETS IN WEB PAGES

- Le regole CSS possono anche essere definite in `<style>` elementi nel `<head>` campo
- In genere è preferibile utilizzare fogli di stile esterni e `<link>`

```
<head>
  <meta charset="UTF-8">
  <title>CSS</title>
  <style>
    h1 {
      color: red;
      font-size: 50px;
    }
  </style>
</head>
```

STYLING HTML ELEMENTS

- Gli elementi HTML possono anche essere formattati in linea, utilizzando l'attributo style
- Il valore dell'attributo style è una sequenza di dichiarazioni, separate da «;»
- Queste dichiarazioni di stile si applicano solo all'elemento specifico che riporta l'attributo

```
<em style="color: fuchsia; font-weight: bold;">inline style</em>
```

CSS: INLINE STYLES

```
<h1>Hello CSS</h1>
<p>
  Our <em style="color:fuchsia;font-weight: bold;">first</em>
  page with <em>style</em>!
</p>
```

```
h1 {
  color: red;
  font-size: 50px;
}

em {
  color: blue;
}
```



SELECTORS



SELECTORS

- I selettori sono una parte fondamentale di CSS
- Specificano a quali elementi si applica una regola CSS
- I selettori CSS non vengono utilizzati solo per lo styling!
- Quando si utilizza JavaScript per rendere dinamiche le pagine Web, è possibile utilizzarlo per
 - Seleziona gli elementi con cui interagire
 - Quando si eseguono test Web automatizzati, possono essere utilizzati per determinare con quali elementi il test deve interagire
 - Quando si esegue lo scraping/crawling, possono essere utilizzati per selezionare gli elementi che
 - contengono le informazioni che vogliamo estrarre

SIMPLE CSS SELECTORS

Esistono cinque tipi di selettori semplici:

Selettore universale (noto anche come carattere jolly). Corrisponde a qualsiasi elemento.

```
* {  
    color: hotpink;  
}
```

```
<p>Here's a list:</p>  
<ul>  
    <li>Ann</li>  
    <li>Bob</li>  
    <li><a href="/car/">Carl</a></li>  
    <li>Dave</li>  
</ul>  
<a href="/">Back to homepage</a>
```

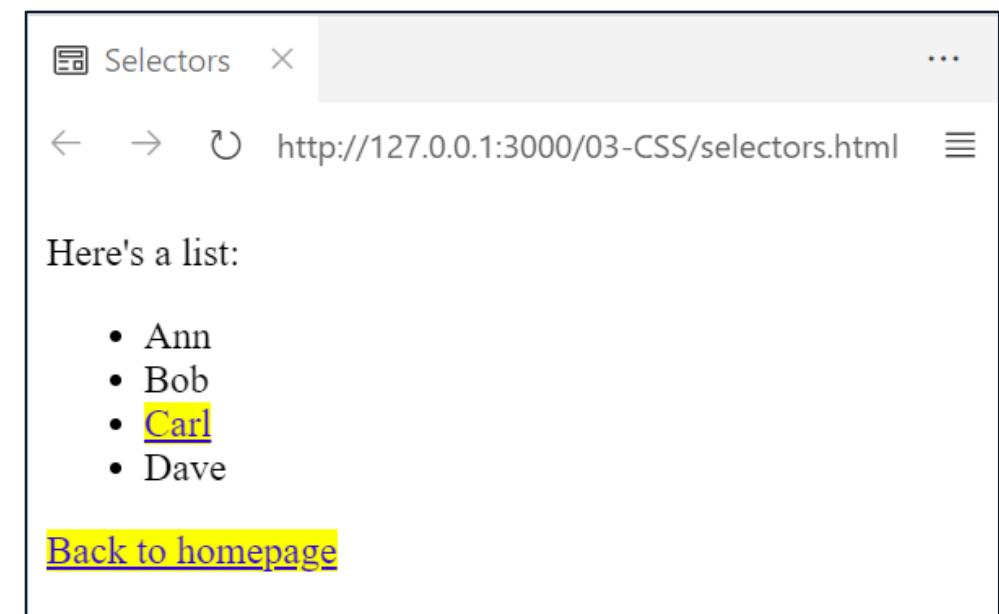


SIMPLE CSS SELECTORS

- Selettore di tipo. Corrisponde a tutti gli elementi di un determinato tipo (ad esempio, il nome del tag)
- Il selettore è semplicemente il nome del tag da abbinare

```
a {  
    background: yellow;  
}
```

```
<p>Here's a list:</p>  
<ul>  
    <li>Ann</li>  
    <li>Bob</li>  
    <li><a href="/car/">Carl</a></li>  
    <li>Dave</li>  
</ul>  
<a href="/">Back to homepage</a>
```

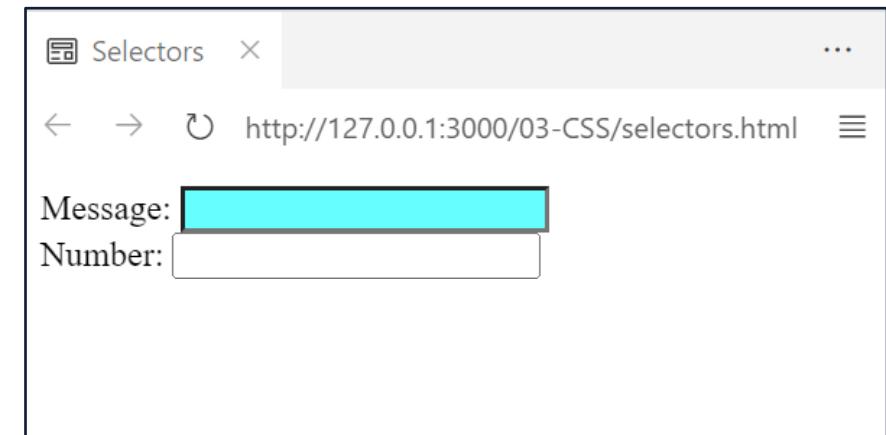


SIMPLE CSS SELECTORS

- Selettore id. Abbina l'elemento con l'attributo id specificato.
- Il selettore ha la forma #ElementId

```
#msg {  
    background: cyan;  
}
```

```
<form>  
    <label for="msg">Message: </label>  
    <input id="msg" type="text" name="msg"><br>  
    <label for="num">Number: </label>  
    <input id="num" type="number" name="num">  
</form>
```

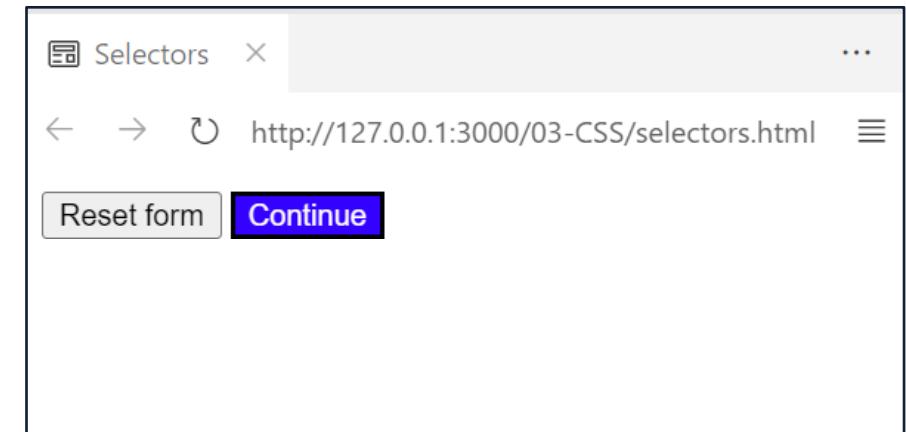


SIMPLE CSS SELECTORS

- Selettore di classe. Corrisponde all'elemento con l'attributo class specificato.
- Il selettore ha il formato .classname

```
.primary {  
    background: blue;  
    color: white;  
}
```

```
<button>Reset form</button>  
<button class="primary btn">Continue</button>
```

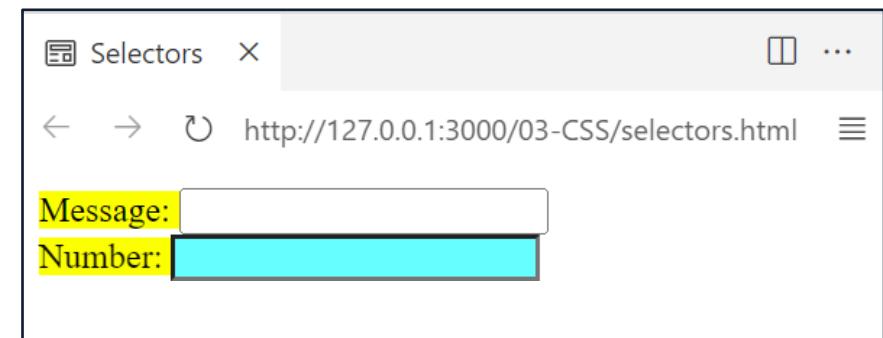


SIMPLE CSS SELECTORS

- Selettore di attributi. Corrisponde all'elemento con un determinato attributo.
- Il selettore ha la forma [attribute] o [attribute='value']

```
[for]{ /*all elems with a for attribute*/
  background: yellow;
}
[type='number']{ /*all elems with type=number*/
  background: cyan;
}
```

```
<form>
  <label for="msg">Message: </label>
  <input id="msg" type="text" name="msg"><br>
  <label for="num">Number: </label>
  <input id="num" type="number" name="num">
</form>
```



SIMPLE CSS SELECTORS

- Gli operatori aggiuntivi (*=, ^=, \$=) consentono la corrispondenza parziale con i valori degli attributi

```
[href*='programming']{ /*contains 'programming'*/
    text-decoration: overline;
}
[href^='https']{ /*start with 'https'*/
    color: red;
}
[href$='.it/']{ /*ends with '.it/'*/
    color: green;
}
```

```
<a href="http://bookofprogramming.com/">Link 1</a>
<a href="https://programming.net/">Link 2</a>
<a href="http://webtechnologies.it/">Link 3</a>A
```



COMPLEX CSS SELECTORS: COMPOUNDS

- È possibile combinare i selettori per ottenere un controllo a grana fine
- Questa operazione viene eseguita concatenando i selettori
- Fondamentalmente seleziona l'intersezione dei selettori coinvolti

```
a[target='_blank'] {  
    color: red;  
}  
  
a.my-class{  
    color: green;  
}  
  
a[href*='programming'].my-class {  
    background: yellow;  
}  
  
<a href="http://bookofprogramming.com/" target="_blank">Link 1</a>  
<a class="my-class" href="https://programming.net/">Link 2</a>  
<em class="my-class">Hello</em>
```



COMPLEX CSS SELECTORS: COMBINATORS

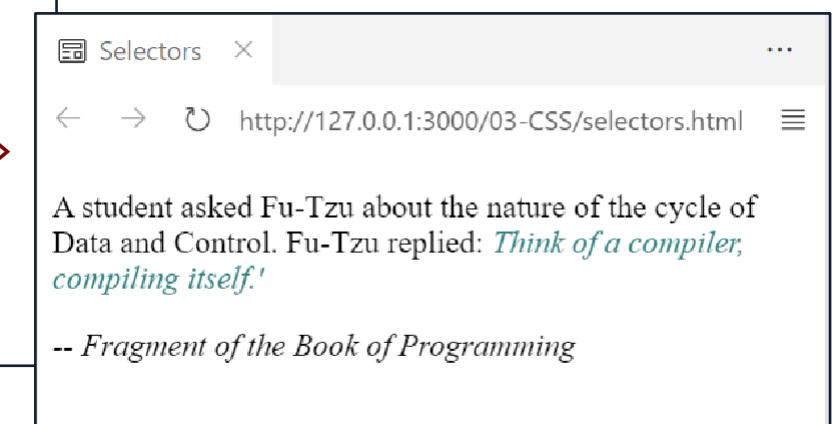
- I combinatori vengono utilizzati per selezionare gli elementi in base alla loro posizione nel documento (ricorda che i documenti HTML possono essere visti come alberi!)
- La sintassi è: selettore1 combinatore selettore2
-
- Nei CSS esistono quattro diversi combinatori:
- Selettore discendente (spazio)
- Selettore bambino (>)
- Selettore di pari livello adiacente (+)
- Selettore di pari livello generale (~)

COMBINATORS: DESCENDANT SELECTOR

- Sintassi: selettoreA selettoreB
- Semantica: corrisponde a tutti gli elementi che corrispondono al selettoreB e sono contenuti all'interno (cioè sono un discendente di) un elemento corrispondente al selettoreA (**em** è contenuto in **section**)

```
<section>
  <p>
    A student asked Fu-Tzu about the nature of
    the cycle of Data and Control. Fu-Tzu replied:
    <em>Think of a compiler, compiling itself.'</em>
  </p>
</section>
<em>-- Fragment of the Book of Programming</em>
```

```
section em {
  color: teal;
}
```



COMBINATORS: CHILD SELECTOR

- Sintassi: selettoreA > selettoreB
- Semantica: corrisponde a tutti gli elementi che corrispondono al selettore B e sono direttamente contenuti all'interno di un elemento che corrisponde al selettore A (cioè sono figli diretti di) un elemento che corrisponde al selettore A (nel codice c'è una sequenza main->em->p->em, lo stile si applica solo al caso main->em, se non ci fosse stato il paragrafo di mezzo sarebbe stato tutto stilizzato)

```
main > em {  
    color: teal;  
    font-variant: small-caps;  
}
```

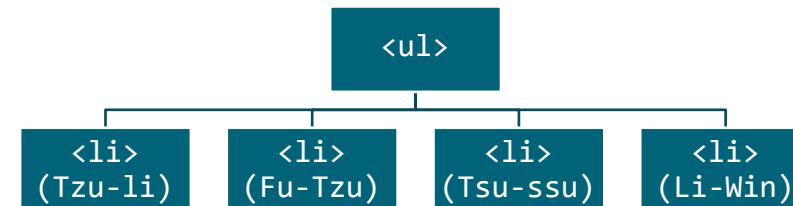
```
<main>  
CSS <em>selectors</em>:  
  <p>We <em>like</em> 'em.</p>  
</main>
```



COMBINATORS: ADJACENT SIBLINGS

- Sintassi: selettoreA + selettoreB
- Semantica: abbina tutti gli elementi che corrispondono al selettore B e sono elementi successivi adiacenti a un elemento che corrisponde al selettoreA (in questo caso si deve applicare lo stile al primo li dopo la classe master)

```
.master + li {  
    color:red;  
}
```



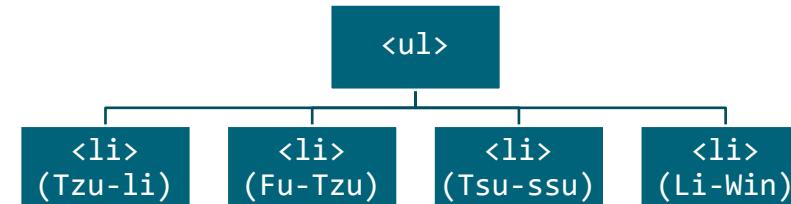
```
<ul>  
    <li>Tsu-li</li>  
    <li class="master">Fu-Tzu</li>  
    <li>Tsu-ssu</li>  
    <li class="disciple">Li-Win</li>  
</ul>
```



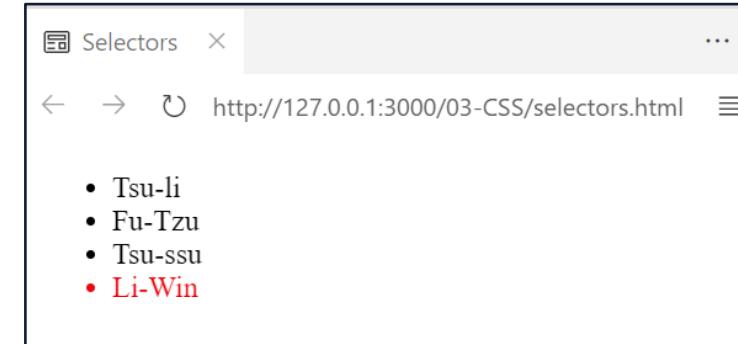
COMBINATORS: GENERAL SIBLINGS

- Sintassi: selectorA ~ selectorB
- Semantica: abbina tutti gli elementi che corrispondono al selettore B e sono fratelli successivi di un elemento che corrisponde al selettoreA (si applica lo stile al primo li con classe disciple dopo la classe master)

```
.master ~ li.disciple {  
    color:red;  
}
```



```
<ul>  
    <li>Tsu-li</li>  
    <li class="master">Fu-Tzu</li>  
    <li>Tsu-ssu</li>  
    <li class="disciple">Li-Win</li>  
</ul>
```



CSS SELECTORS: PSEUDO-CLASSES

Gli elementi HTML possono trovarsi in stati diversi, ad esempio a causa delle interazioni dell'utente o della loro relazione con altri elementi.

I selettori di pseudo-classi iniziano con «:» e consentono di stilizzare gli elementi in base al loro stato:

- Stati interattivi (risultanti dall'interazione dell'utente)
- Stati storici (utilizzati per «ricordare» quali link sono stati visitati)
- Stati della forma (specifici dell'interazione con le forme)
- Stati membri derivanti da relazioni con altri elementi

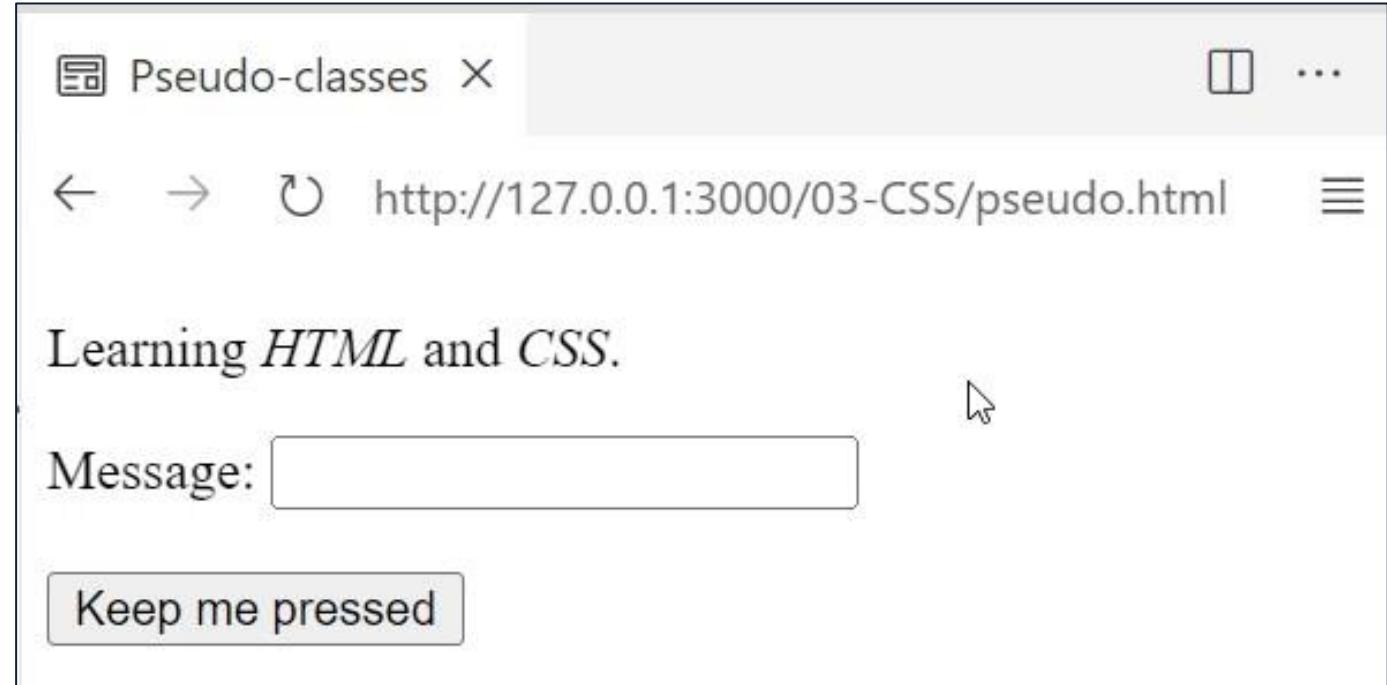
PSEUDO-CLASSES: INTERACTIVE STATES

- :hover seleziona gli elementi su cui è posizionato un dispositivo di puntamento (ad esempio: il mouse)
- :active corrisponde allo stato in cui si interagisce attivamente con un elemento (ad esempio: viene premuto il pulsante)
- :focus corrisponde allo stato in cui un elemento (ad esempio: un collegamento o un campo di input) ha il focus (ad esempio: è attualmente selezionato) nella pagina web

PSEUDO-CLASSES: INTERACTIVE STATES

```
<p>Learning <em>HTML</em> and <em>CSS</em>.</p>
Message: <input type="text"><br><br>
<button>Keep me pressed</button>
```

```
em:hover {
    background: yellow;
}
input[type='text']:focus{
    background: cyan;
}
button:active{
    background: blue;
    color: white;
}
```

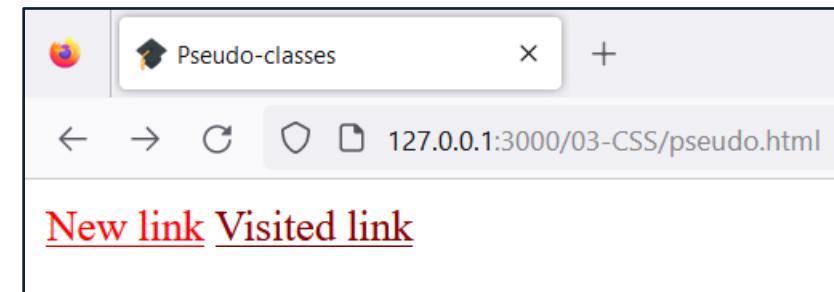


PSEUDO-CLASSES: HISTORIC STATES

- `:link` seleziona i link che non sono ancora stati visitati
- `:visited` seleziona i link che sono già stati visitati

```
:link{  
    color: red;  
}  
:visited{  
    color: darkred;  
}
```

```
<a href=".js/">New link</a>  
<a href=".css/">Visited link</a>
```

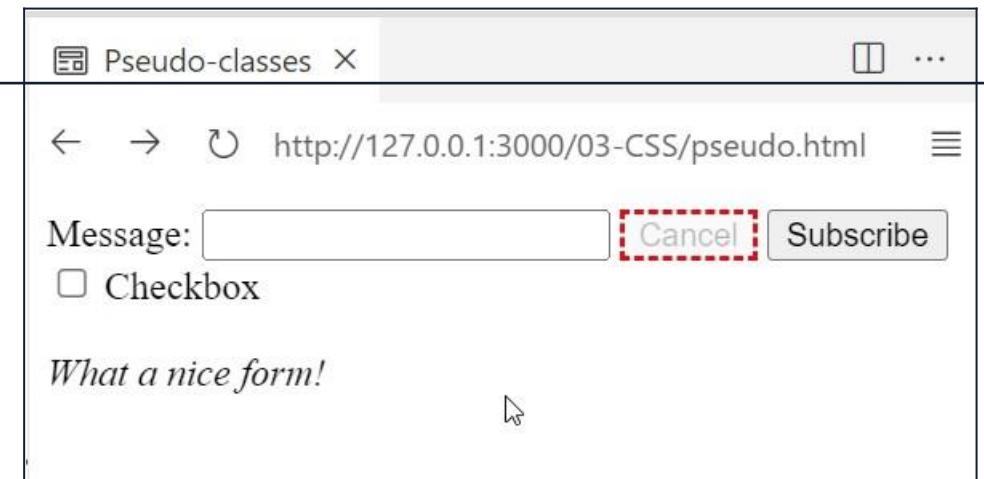


Historic states are **not supported** in the *Live Preview* browser within Visual Studio Code! Open the page in Firefox to check them out.

PSEUDO-CLASSES: FORM STATES

```
<div>
  <label for="mail">Message:</label><input id="mail" type="email">
  <button disabled>Cancel</button><button>Subscribe</button>
</div>
<input type="checkbox"> Checkbox<br><br>
<em>What a nice form!</em>
```

```
:disabled {
  border: 2px dashed red;
}
:invalid {
  color: red;
}
:checked ~ em {
  color: deeppink; font-weight: bold;
}
```



Technically, there can be email address without a dot. For example, user@localhost or user@com are valid addresses!

PSEUDO-CLASSES: POSITION RELATIONS

- :first-child e :last-child selezionano il primo (ultimo) figlio tra un insieme di fratelli.
- :only-child può essere utilizzato per selezionare elementi che non hanno fratelli.
- :first-of-type e :last-of-type possono essere utilizzati per selezionare gli elementi che sono il primo (ultimo) figlio di un insieme di fratelli, considerando solo gli elementi dello stesso tipo.
- :nth-child(n) e :nth-of-type(n) possono essere utilizzati per selezionare elementi che si trovano nella posizione n-esima tra i loro fratelli. L'indicizzazione in CSS inizia da 1!

PSEUDO-CLASSES: POSITION RELATIONS

```
<p>
  <em>Ann</em> <strong>Bob</strong> <em>Carl</em>
</p>
<p>
  <strong>Ann</strong> <em>Bob</em> <strong>Carl</strong>
</p>
```

Ann Bob Carl
Ann Bob Car

PSEUDO-CLASSES: POSITION RELATIONS

```
<p>
  <em>Ann</em> <strong>Bob</strong> <em>Carl</em>
</p>
<p>
  <strong>Ann</strong> <em>Bob</em> <strong>Carl</strong>
</p>
```

Ann Bob Carl
Ann Bob Car

```
em:last-child {
  color: red;
}
```

Ann Bob **Carl**
Ann Bob Car

PSEUDO-CLASSES: POSITION RELATIONS

```
<p>
  <em>Ann</em> <strong>Bob</strong> <em>Carl</em>
</p>
<p>
  <strong>Ann</strong> <em>Bob</em> <strong>Carl</strong>
</p>
```

Ann Bob Carl
Ann Bob Car

```
em:last-of-type {
  color: red;
}
```

Ann Bob Carl
Ann Bob Car

PSEUDO-CLASSES: POSITION RELATIONS

```
<p>
  <em>Ann</em> <strong>Bob</strong> <em>Carl</em>
</p>
<p>
  <strong>Ann</strong> <em>Bob</em> <strong>Carl</strong>
</p>
```

Ann Bob Carl
Ann Bob Car

```
em:first-child {
  color: red;
}
```

Ann Bob Carl
Ann Bob Car

PSEUDO-CLASSES: POSITION RELATIONS

```
<p>
  <em>Ann</em> <strong>Bob</strong> <em>Carl</em>
</p>
<p>
  <strong>Ann</strong> <em>Bob</em> <strong>Carl</strong>
</p>
```

Ann Bob Carl
Ann Bob Car

```
em:first-of-type {
  color: red;
}
```

Ann Bob Carl
Ann Bob Car

PSEUDO-CLASSES: POSITION RELATIONS

```
<p>
  <em>Ann</em> <strong>Bob</strong> <em>Carl</em>
</p>
<p>
  <strong>Ann</strong> <em>Bob</em> <strong>Carl</strong>
</p>
```

Ann **Bob** Carl
Ann **Bob** Car

```
em:nth-child(2) {
  color: red;
}
```

Ann **Bob** Carl
Ann **Bob** Car

PSEUDO-CLASSES: POSITION RELATIONS

```
<p>Lectures:</p>
<ol>
  <li>Introduction</li>
  <li>HTML</li>
  <li>CSS (basics)</li>
  <li>CSS (frameworks + Sass)</li>
  <li>JavaScript</li>
</ol>
```

```
li:nth-child(even){
  color: red;
}
li:nth-child(odd){
  color: blue;
}
```



PSEUDO-ELEMENTS

Gli pseudo-elementi possono essere utilizzati per indirizzare parti specifiche del contenuto di un determinato elemento HTML, senza aggiungere ulteriore markup HTML

La sintassi dei selettori di pseudo-elementi è selector::pseudo-element
selector è un selettore CSS per l'elemento di destinazione

pseudo-element è uno dei selettori di pseudo-elementi supportati:

::first-letter: prende di mira la prima lettera del contenuto di un elemento a livello di blocco

::first-line: si rivolge alla prima riga del contenuto di un elemento a livello di blocco

::selection: si rivolge al contenuto attualmente selezionato dall'utente

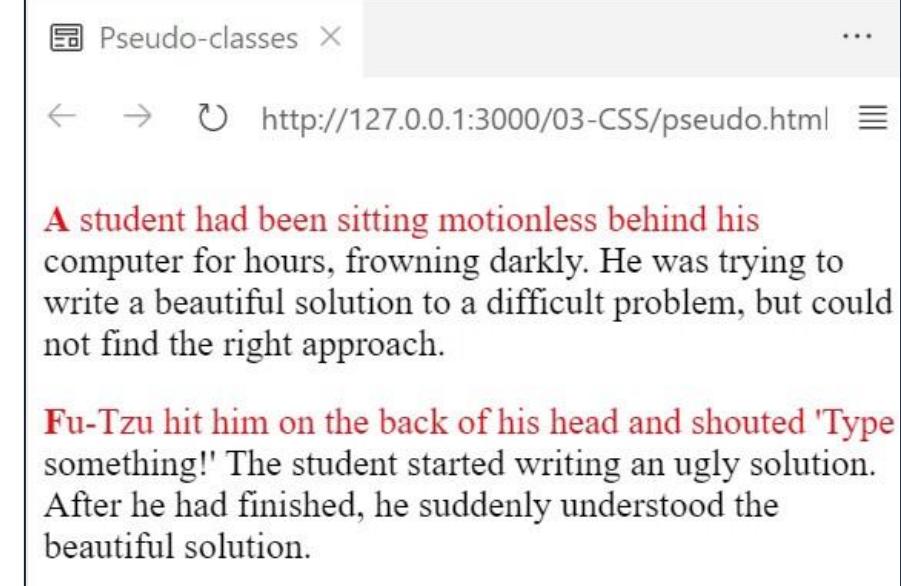
::before: crea un elemento che è il primo figlio dell'elemento selezionato

::after: crea un elemento che è l'ultimo figlio dell'elemento selezionato

PSEUDO-ELEMENTS: EXAMPLES

```
<p>A student had been sitting motionless behind his computer for hours,  
frowning darkly. He was trying to write a beautiful solution to a difficult  
problem, but could not find the right approach.</p>  
<p>Fu-Tzu hit him on the back of his head and shouted 'Type something!' The  
student started writing an ugly solution. After he had finished, he suddenly  
understood the beautiful solution.</p>
```

```
p::first-letter {  
    font-weight:bold;  
}  
  
p::first-line{  
    color: red;  
}  
  
p:last-child::selection {  
    background: red;  
    color: white;  
}
```



PSEUDO-ELEMENTS: EXAMPLES

```
<p class="narrator">A hermit spent ten years writing the perfect program. He  
proudly announced: </p>  
<p class="hermit">'My program can compute the motion of the stars on a 286-  
computer running MS DOS'.</p>  
<p class="narrator">Fu-Tzu responded:</p>  
<p class="fu-tzu">'Nobody owns a 286-computer or uses MS DOS anymore.'</p>
```

```
.narrator::before {  
    content: "Narrator » "; font-weight: bold;  
}  
.narrator::after {  
    content: " «"; font-weight: bold;  
}  
.hermit::before {  
    content: " 🧑 "; font-size: 24px;  
}  
.fu-tzu::before {  
    content: " 🧑 "; font-size: 24px;  
}
```

Narrator » A hermit spent ten years writing the perfect program. He proudly announced: «

🧑́ : 'My program can compute the motion of the stars on a 286-computer running MS DOS'.

Narrator » Fu-Tzu responded: «

🧑́ : 'Nobody owns a 286-computer or uses MS DOS anymore.'

THE CASCADE



THE CASCADE IN CASCADING STYLE SHEETS

- A volte, due o più regole possono essere applicate allo stesso elemento
- Queste regole potrebbero essere in conflitto, ad esempio assegnare valori diversi alla stessa proprietà (ad esempio: colore)
- La cascade è l'algoritmo utilizzato per risolvere tali conflitti
- Input: un insieme di proprietà in conflitto che si applicano a un determinato elemento
- Output: una singola proprietà a cascade da applicare effettivamente
- La cascade considera 4 aspetti chiave, nell'ordine:
- Origine e importanza
- Strati
- Specificità
- Posizione e ordine di apparizione della regola

THE CASCADE: ORIGIN

- The CSS we write (a.k.a. **authored CSS**) is not the only one being applied to a web page
- We've already mentioned that **user agent styles** exist
 - The stylesheets that are included by browsers by default
- Other styles (a.k.a. **local user styles**) might be added by specific browser extensions or from the operating system level
 - For example, for accessibility purposes
 - Visually-impaired persons might want to use high-contrast color schemes, with larger fonts, etc.

THE CASCADE: IMPORTANCE

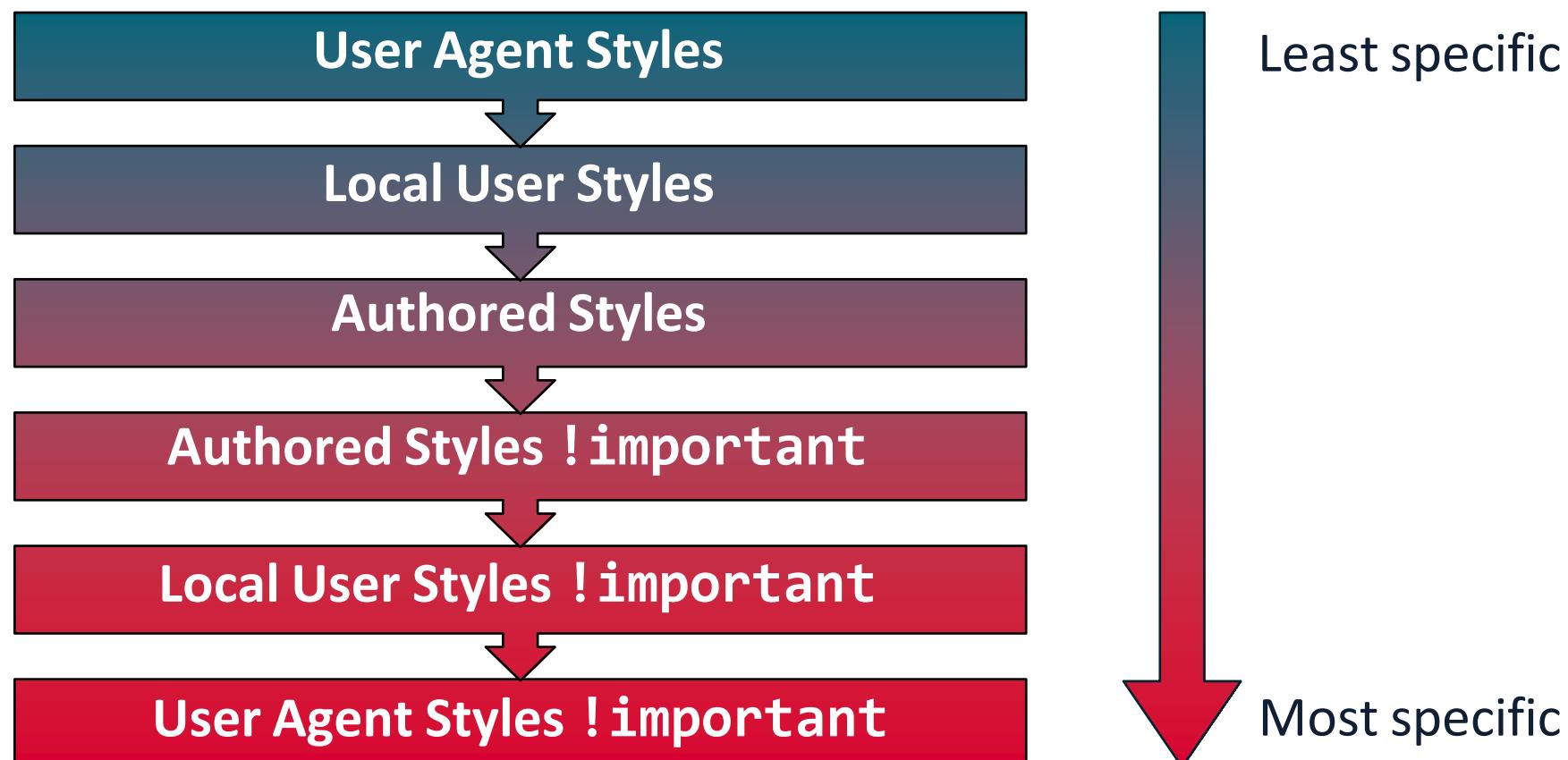
- La regola !important può essere utilizzata per aggiungere maggiore importanza a una proprietà all'interno di una regola CSS
- !important viene semplicemente aggiunto alla fine della dichiarazione di proprietà

```
h1 {  
    color: red !important;  
}
```

- L'importanza gioca un ruolo significativo nella cascade

THE CASCADE: ORIGIN-IMPORTANCE

From the least specific origin to the most specific one



THE CASCADE: LAYERS

All'interno di ogni bucket di origine/importanza, possono essere presenti più livelli a cascata

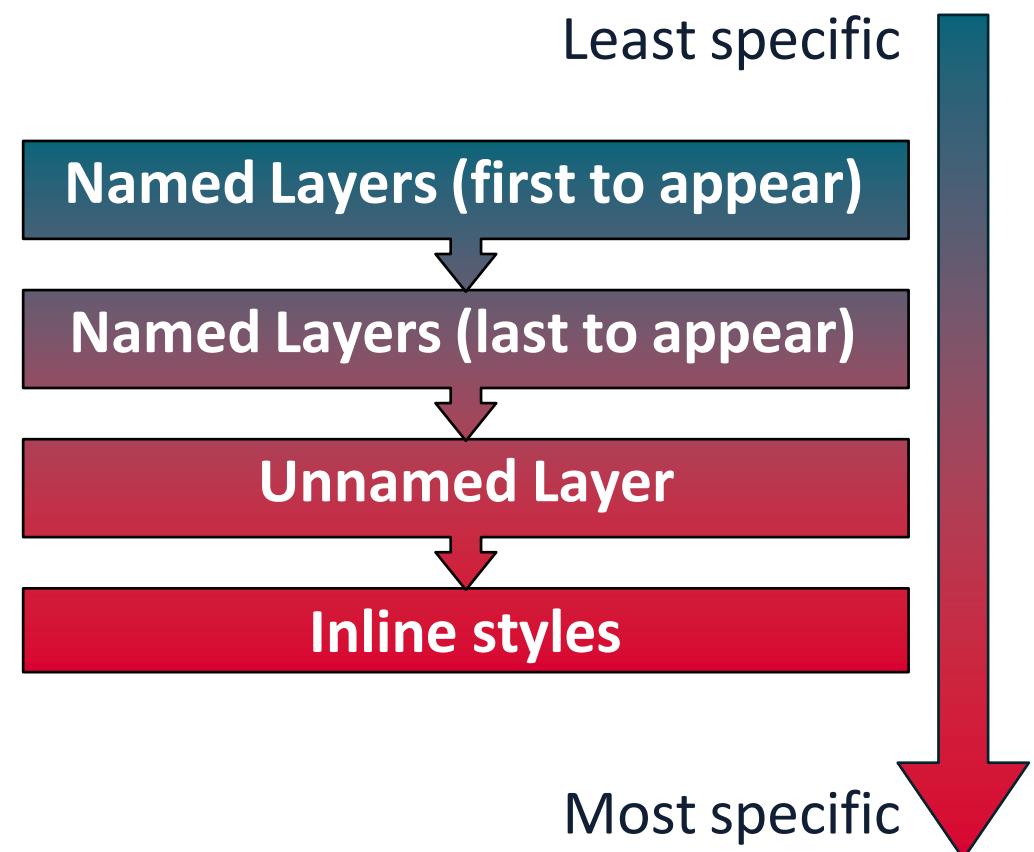
All'interno degli stili creati:

I livelli personalizzati possono essere definiti utilizzando la regola @layer (non lo vedremo)

I livelli personalizzati dichiarati in un secondo momento hanno una priorità più alta

Tutti gli altri CSS (in <style> o importati con <link>) appartengono a un livello senza nome

Gli stili in linea appartengono a un livello separato e hanno la priorità più alta



THE CASCADE: SPECIFICITY

Quando due regole in conflitto:

Appartengono allo stesso bucket di importanza origine e

Appartiene allo stesso livello

Viene presa in considerazione la specificità.

L'idea è che vinca il selezionatore più specifico

```
.primary {  
    color: blue;  
}
```



```
h1 {  
    color: red;  
}
```

```
<h1 class="primary">Cascading!</h1>
```

THE CASCADE: SPECIFICITY

I CSS definiscono come calcolare la specificità di un selettore:

Ignorare il selettore universale

Conta il numero di selettori id (=A)

Conta il numero di selettori di classe, attributo e pseudo-classi (=B)

Contare il numero di selettori di tipo e pseudo-elemento (=C)

La specificità è una tripla numerica (A, B, C) calcolata come sopra

THE CASCADE: SPECIFICITY EXAMPLES

- A: Number of id selectors
- B: Number of class, attribute and pseudo-classes selectors
- C: Number of type and pseudo-element selectors

Selector	Specificity (A, B, C)
#id	(1, 0, 0)
em.master[target]	(0, 2, 1)
#navbar ul li a.nav-link[href*='/']	(1, 2, 3)
article.item section p::first-letter	(0, 1, 4)
a:hover	(0, 1, 1)
*	(0, 0, 0)

THE CASCADE: COMPARING SPECIFICITIES

I confronti vengono effettuati considerando le tre componenti nell'ordine:

- 1) la specificità con una A più grande è più specifica;
- 2) se le due A sono in parità, allora vince la specificità con una B più grande;
- 3) se anche le due B sono in parità, allora vince la specificità con una C più grande;
- 4) Se tutti i valori pareggiano, le due specificità sono uguali.

THE CASCADE: SPECIFICITY



Selector #1	Specif. #1	Selector #2	Specif. #2	Winner
a[target]	(0, 1, 1)	.list a	(0, 1, 1)	Draw
#msg	(1, 0, 0)	input[type].inp	(0, 2, 1)	#1
#nav > #brd a.1k	(2, 1, 1)	em.foo.bar.light	(0, 3, 1)	#1
[id='nav'] a	(0, 1, 1)	#nav a	(1, 0, 1)	#2

THE CASCADE: POSITION AND APPEARANCE

Quando due proprietà:

Appartengono allo stesso bucket di origine/importanza

Appartengono allo stesso livello

Avere la stessa specificità

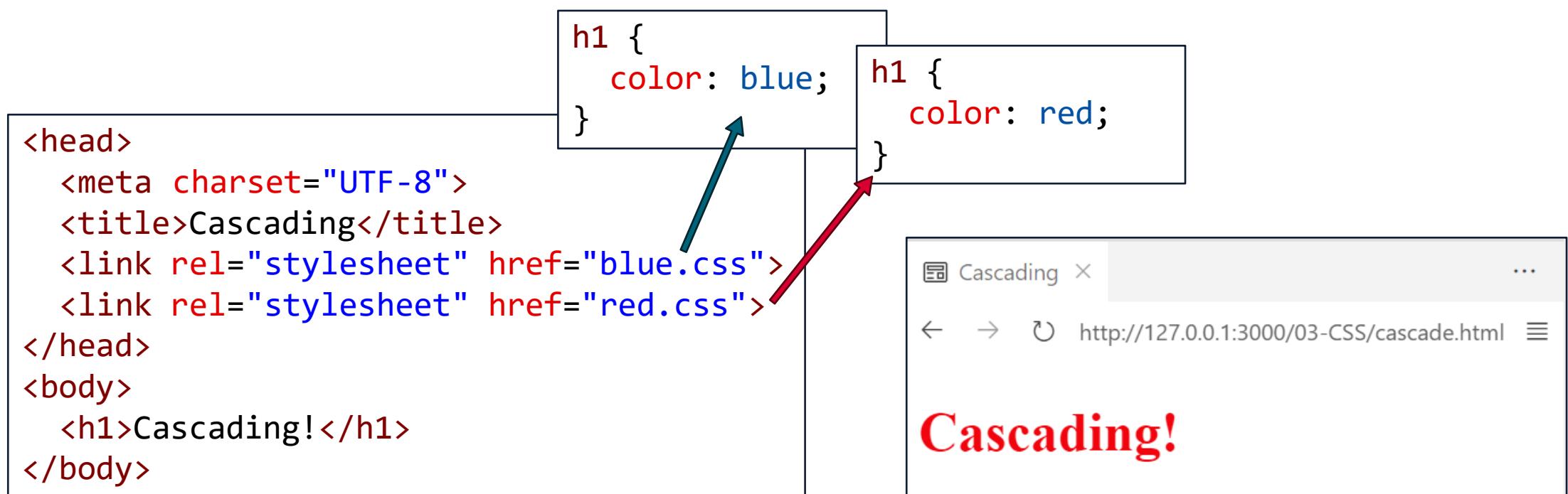
L'ultima regola che viene visualizzata ha la priorità più alta

```
h1 {  
    color: red;  
}  
h1 {  
    color: blue;  
}
```

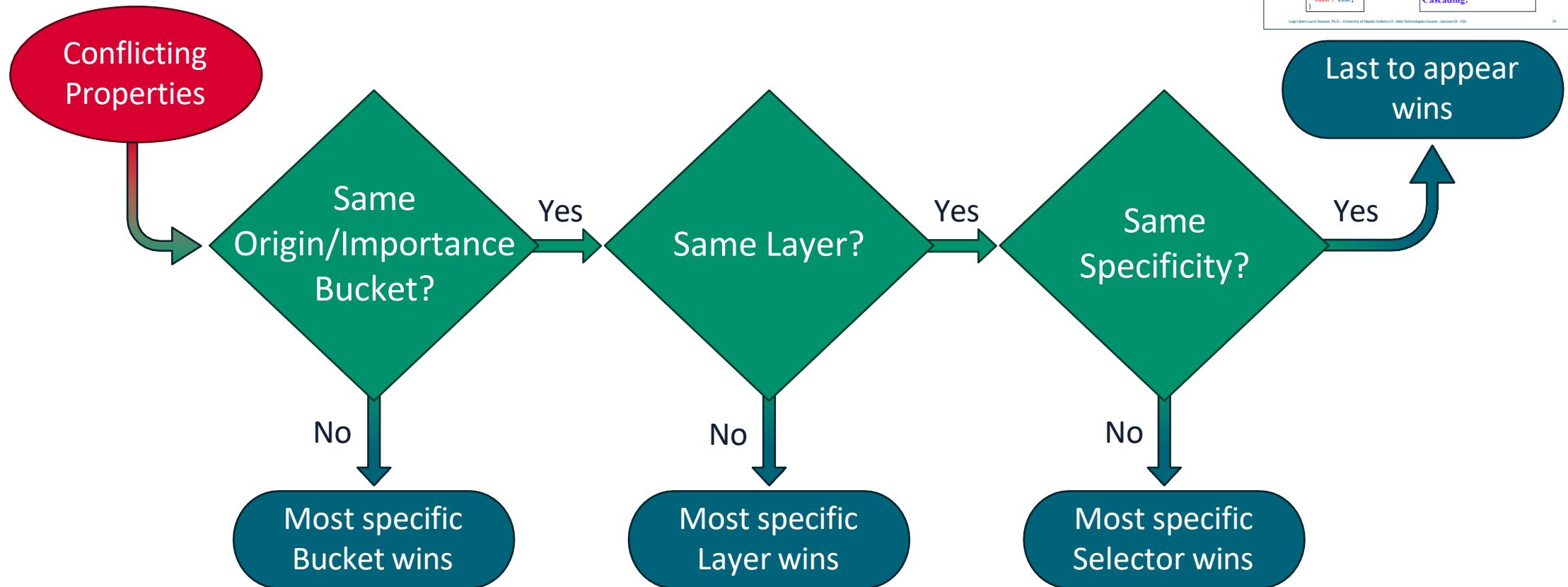


THE CASCADE: POSITION AND APPEARANCE

- Questa regola si applica all'interno dello stesso foglio di stile e nell'ordine in cui appaiono i fogli di stile



THE CASCADE: OVERVIEW



THE CASCADE: ORIGIN-IMPORTANCE

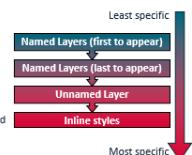
From the least specific origin to the most specific one



THE CASCADE: LAYERS

Within each origin/importance bucket, there can be multiple cascade layers

- Within authored styles:
 - Custom layers can be defined using `@layer` rule (we won't see that)
 - The layers that are declared later have higher priority
- All other CSS (in `<style>` or imported with `<link>`) belongs to a unnamed layer
- Inline styles belong to a separate layer and have the highest priority



THE CASCADE: SPECIFICITY

CSS defines how to calculate the specificity of a selector:

- Ignore the universal selector
- Count the number of id selectors (=A)
- Count the number of class, attribute and pseudo-classes selectors (=B)
- Count the number of type and pseudo-element selectors (=C)

The specificity is a numeric triple (A, B, C) computed as above

THE CASCADE: POSITION AND APPEARANCE

When two properties:

- Belong to the same origin/importance bucket
- Belong to the same layer
- Have the same specificity

The last rule to appear has the highest priority

A screenshot of a browser's developer tools showing the CSS cascade. It shows two rules for `h1`: one with `color: red;` and another with `color: blue;`. The second rule is highlighted, indicating it is the last to appear and therefore has the highest priority.

THE CASCADE IN BROWSER DEV TOOLS

The screenshot shows a browser window with the title "Cascading" and the URL "127.0.0.1:3000/03-CSS/cascade.html". The page content is "**Cascading!**". The developer tools are open, specifically the "Inspector" tab. On the left, the DOM tree shows the HTML structure:

```
<!DOCTYPE html>
<html lang="en"> event
  <head>
    <meta charset="UTF-8">
    <title>Cascading</title>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <h1 class="primary">Cascading!</h1>
  </body>
</html>
```

The "Inspector" tab displays the CSS rules for the selected element, an **h1** with the class **primary**. The rules are listed from most specific to least specific:

- element { inline }
- h1.primary { color: teal; } (style.css:1)
- .primary { color: blue; } (style.css:4)
- h1 { color: red; } (style.css:7)
- h1 { (user agent) html.css:166 display: block; font-size: 2em; font-weight: bold; margin-block-start: .67em; margin-block-end: .67em; }

A large red arrow on the right points upwards, labeled "Most Specific" at the top and "Least Specific" at the bottom, indicating the direction of increasing specificity.

User agent styles are **hidden** in Dev Tools by default. If you want to see them, press F1 in Dev Tools and change the settings.

INHERITANCE

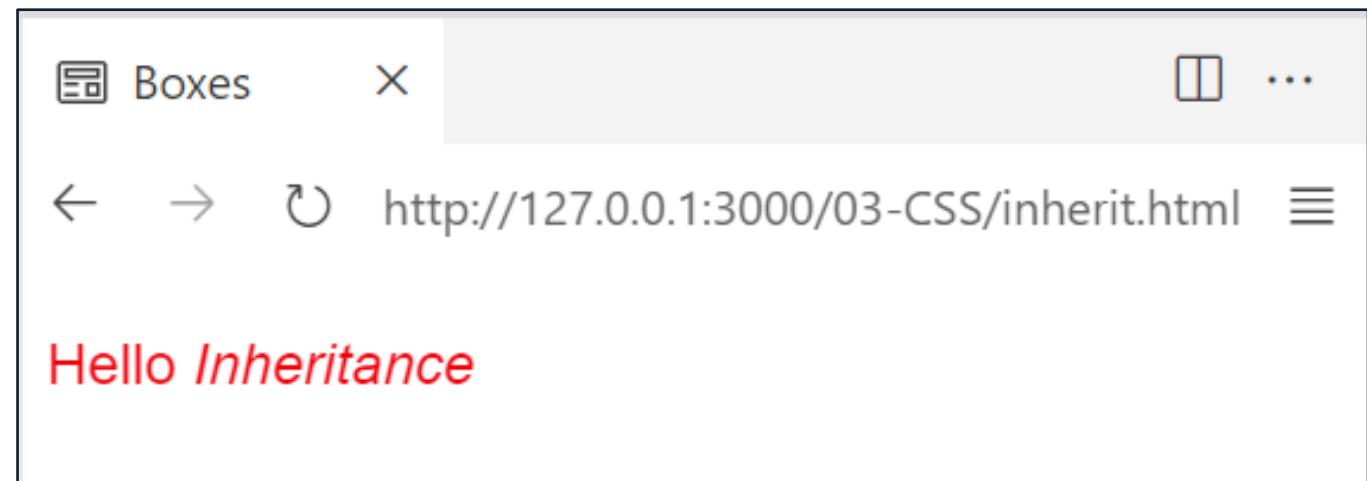


INHERITANCE IN CSS

- Alcune proprietà CSS possono essere ereditate dagli elementi predecessori, se non è impostato alcun valore specifico
- Le proprietà ereditabili includono colore, dimensione del carattere, famiglia di caratteri, peso del carattere, stile del carattere

```
p {  
    font-family: sans-serif;  
    color: red;  
    font-style: normal;  
}
```

```
<p>  
    Hello <em>Inheritance</em>  
</p>
```



INHERITANCE IN CSS

The screenshot shows the Chrome DevTools Inspector with the title bar "Boxes". The URL is "127.0.0.1:3000/03-CSS/inherit.html". The left sidebar shows the HTML structure:

```
<!DOCTYPE html>
<html lang="en"> event
  <head> ...
  <body>
    <p>Hello<br/>
      <em>Inheritance</em>
    </p>
  </body>
</html>
```

The element `Inheritance` is selected. The right panel displays the computed styles:

- `element :: { font-style: italic; }` (Inherited from `i, cite, em, var, (user agent) html.css:509`)
- `Inherited from p`
 - `p :: { font-family: sans-serif; color: red; font-style: normal; }` (Inherited from `html`)
- `:root :: { color: CanvasText; }` (Inherited from `(user agent) ua.css:394`)

Inherited properties have the lowest specificity of all styling methods

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
WEB TECHNOLOGIES – LECTURE 04

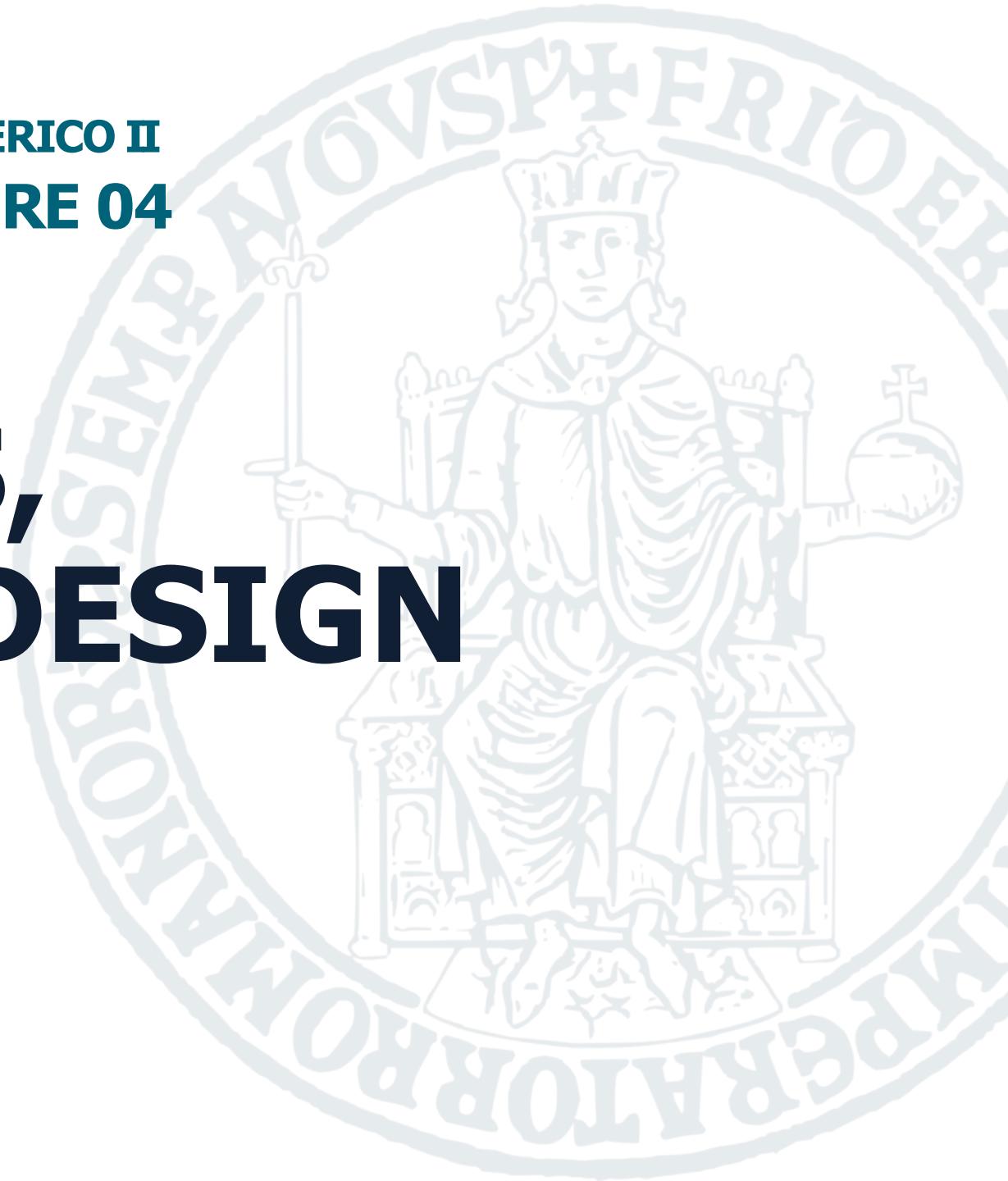
CSS: LAYOUTS, RESPONSIVE DESIGN

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



PREVIOUSLY, ON WEB TECHNOLOGIES

In the last lecture, we have learned:

- What **CSS** is and how to include it in HTML documents
- Some basic **styling properties** (color, background, font-style, ...)
- Selectors
- The Cascade algorithm
- Inheritance

CSS SIZING UNITS

CSS SIZING UNITS OVERVIEW

Alcune proprietà di CSS possono essere usate per modificare la dimensione degli elementi: larghezza, altezza, font-size, padding, bordi...

Quando si ridimensionano gli elementi in CSS è possibile usare:

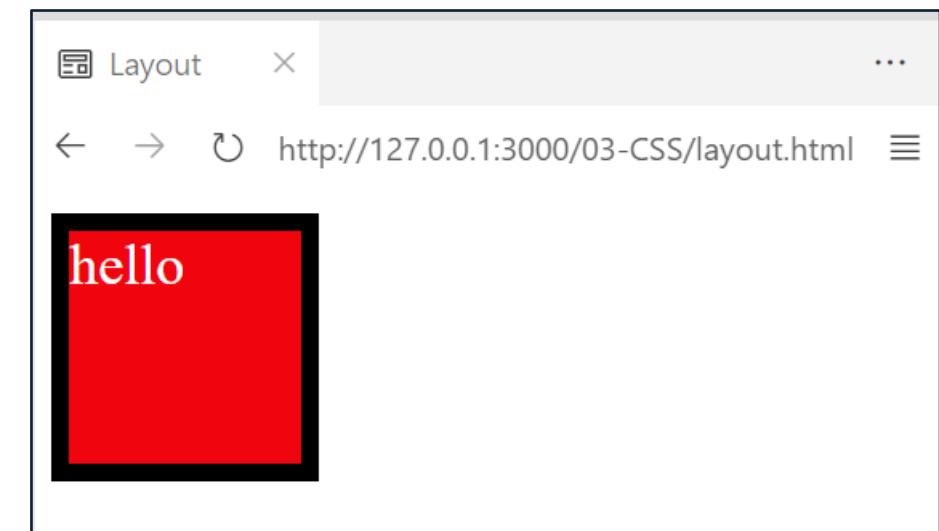
- Lunghezze relative (dipende da altre altezze)
- Lunghezze assolute

CSS: ABSOLUTE SIZING UNITS

- Le lunghezze assolute sono definite usando un numero e una delle unità di lunghezza supportate

```
div {  
    background: red;  
    width: 2.54cm; /* centimeters */  
    height: 1in; /* inches */  
    border: 2mm solid black; /* millimiters */  
    color: white;  
    font-size: 24px; /* pixels */  
}
```

```
<div>hello</div>
```

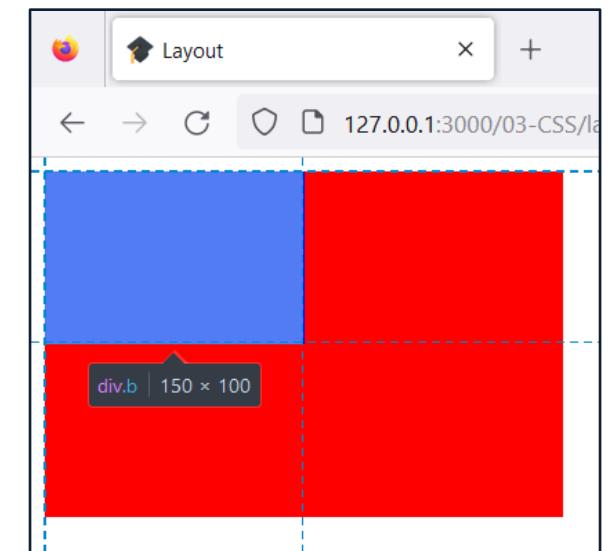


CSS: PERCENTAGE SIZING

- Le percentuali definiscono dimensioni relativi a elementi padre/adiacenti. Possono essere usate con molte proprietà includendo width, height, ...

```
.a {  
    width: 300px; height: 200px;  
    background: red;  
}  
.b {  
    width: 50%; /*150px*/  
    height: 50%; /*100px*/  
    background: blue;  
}
```

```
<div class="a">  
    <div class="b"></div>  
</div>
```



CSS: RELATIVE SIZING UNITS (FONTS)

Le unità di misura relativa del font includono:

- Em: 1em rappresenta l'attuale font-size, 1.5em è il 50% più grande dell'attuale font-size
- Rem: rappresenta la font-size calcolata all'elemento radice (di base 16px)

```
p {  
    font-family: sans-serif;  
    font-size: 1.5rem; //24px  
}  
  
span {  
    font-size: 1.5em; //36px  
}
```

```
<p>HELLO<span>SIZES</span></p>
```

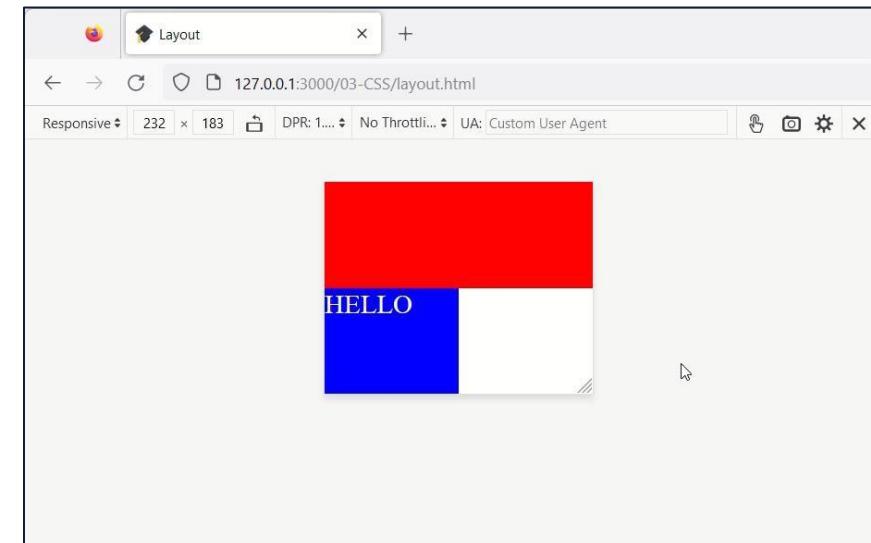


CSS: RELATIVE SIZING UNITS (VIEWPORT)

La viewport è la finestra del browser. Le unità relative alla viewport includono:

- vw: 1vw rappresenta 1% dell'attuale larghezza della viewport
- vh: 1vh rappresenta 1% dell'altezza attuale della viewport

```
div {height: 50vh;} //50% altezza vp
.a {background: red;} //classe a in rosso
.b{ //classe b in blu, con testo bianco
  background: blue; color: white;
  width: 50vw;font-size:10vw;
} //larga il 50% della vp e il font è largo il
10% della vp
<div class="a"></div>
<div class="b">HELLO</div>
```

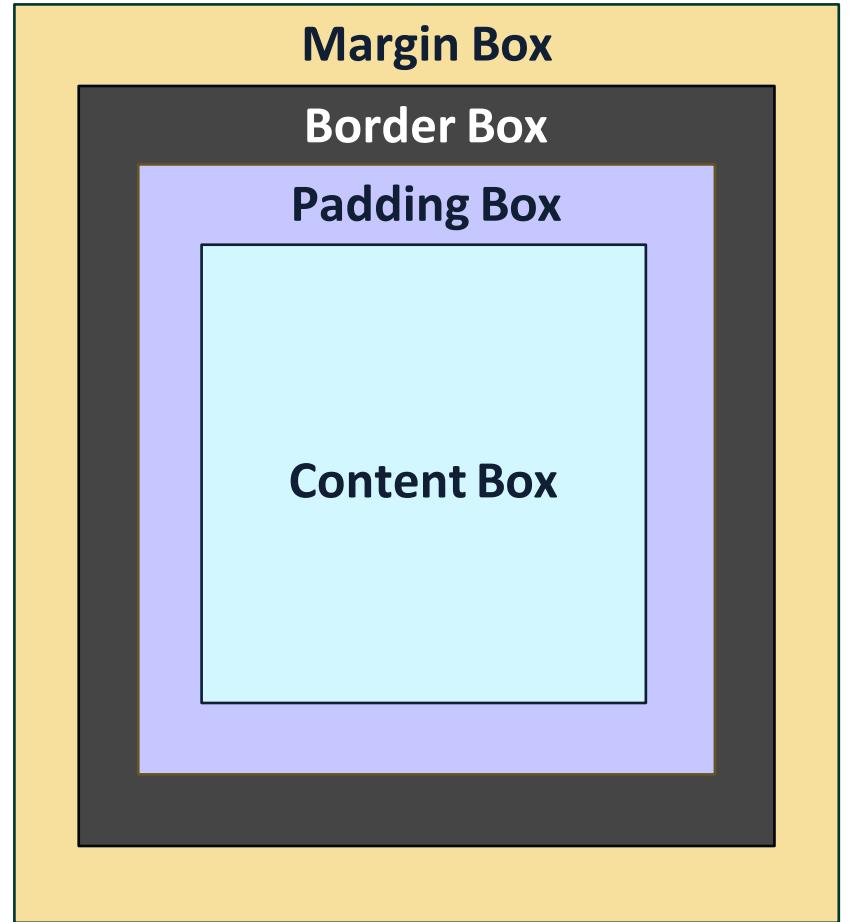


LAYOUTS



THE BOX MODEL

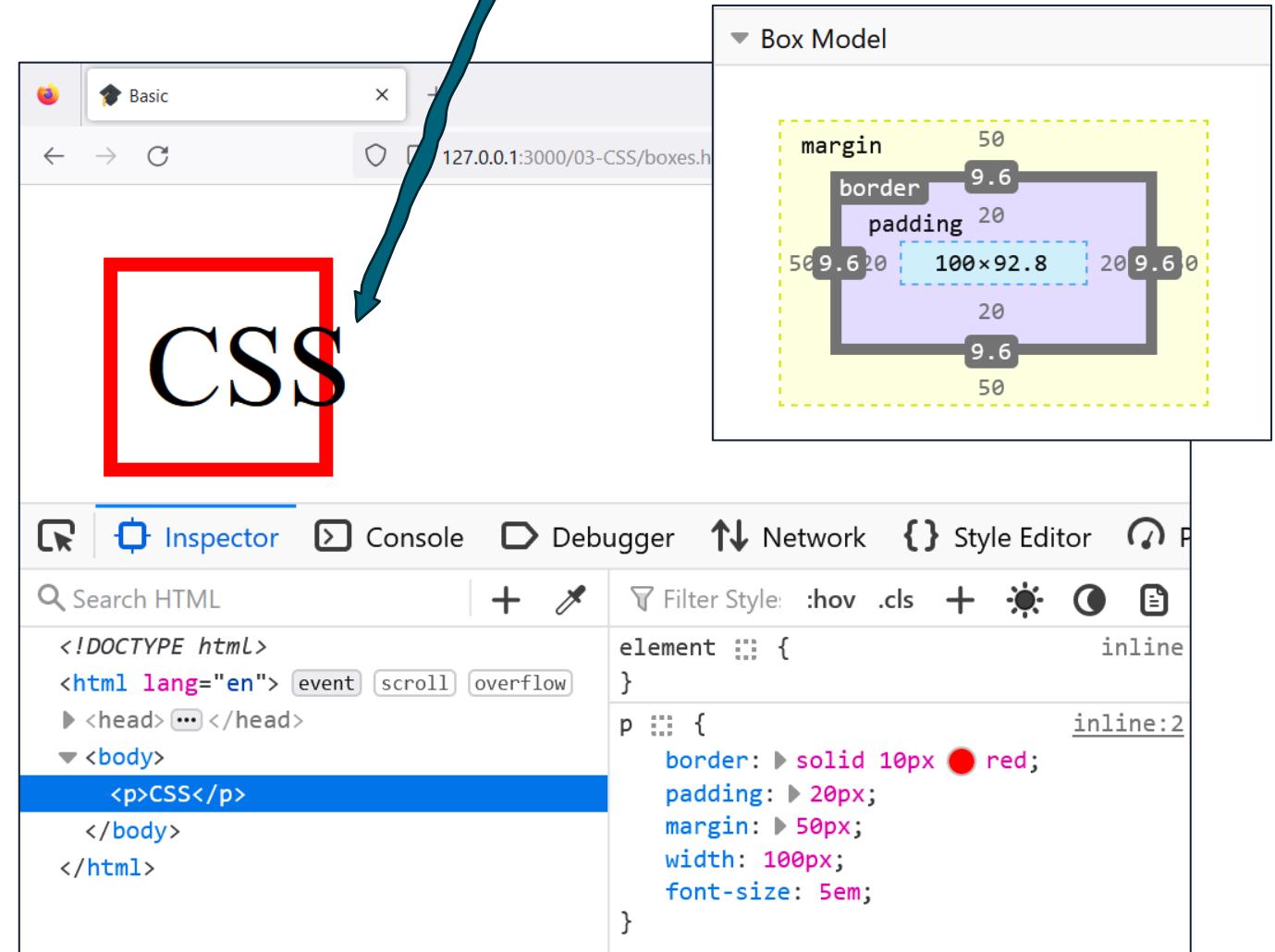
- Il box model stabilisce che ogni elemento di HTML è una box. I box sono fatti di aree distinte, come la content box (contenuti), padding box (separa il contenuto dai bordi), border box e margin box.



THE BOX MODEL

- La dimensione di ogni area può essere definita usando dichiarazioni CSS.
- Il comportamento e come appaiono le box è determinato dalla loro: layout mode, dal contenuto e dalle proprietà del box model.

Content might be too big to fit in its box. The part of content that does not fit is called «**overflow**».

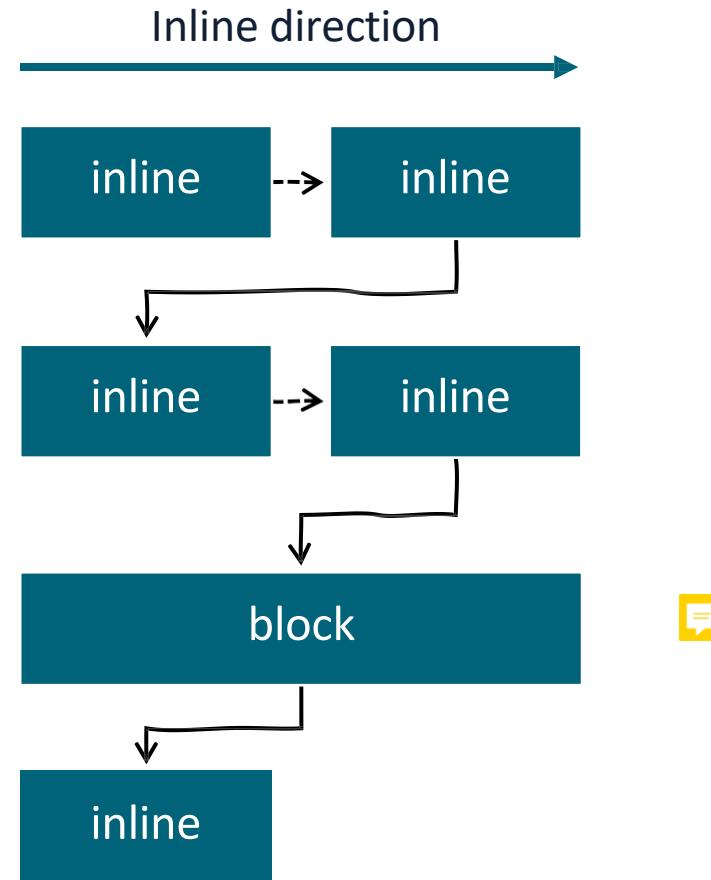


CSS FLOW LAYOUT

- Di base i box sono mostrati usando il flow layout. Gli elementi inline sono mostrati uno dopo l'altro in verticale (quindi seguendo la linea/riga), gli elementi a blocco, invece, sono mostrati sequenzialmente uno sopra l'altro.

Beatrice Morgillo
2024-03-15 13:09:50

I block passano sempre alla riga successiva, rispetto agli inline. Vanno sempre alla riga successiva, nonostante lo spazio

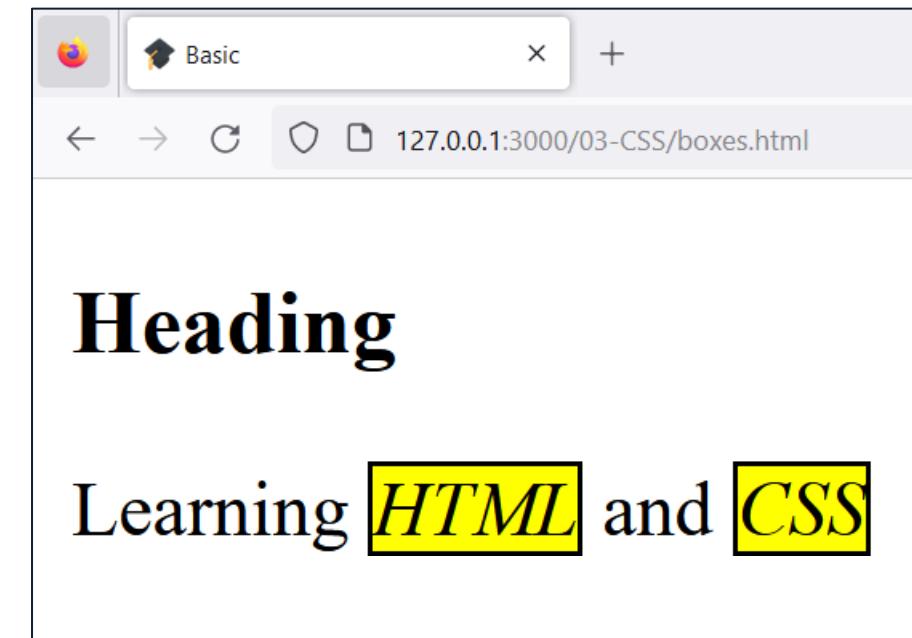


THE DISPLAY PROPERTY: INLINE

- `display: inline` posiziona i box uno dopo l'altro seguendo la linea, come le parole in una frase
- Anchors `<a>`, `` and `` are typically displayed inline in default user agent styles

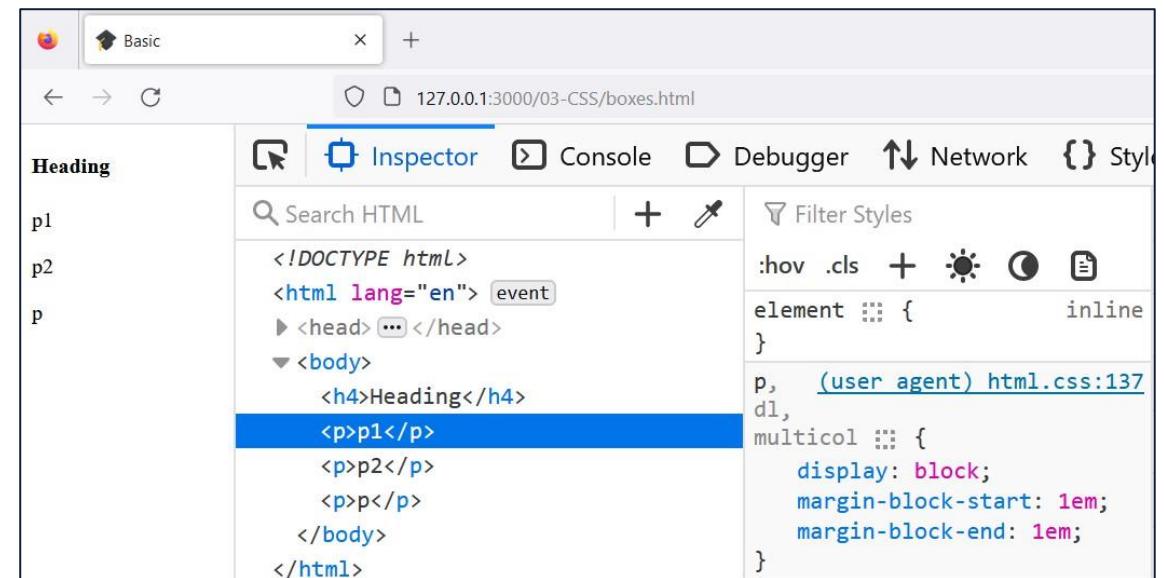
```
em {  
    display: inline; /* default */  
    background: yellow; border: 1px solid;  
    width: 50px; height: 50px; /* ignored */  
}
```

```
<h3>Heading</h3>  
<p>  
    Learning <em>HTML</em> and <em>CSS</em>  
</p>
```

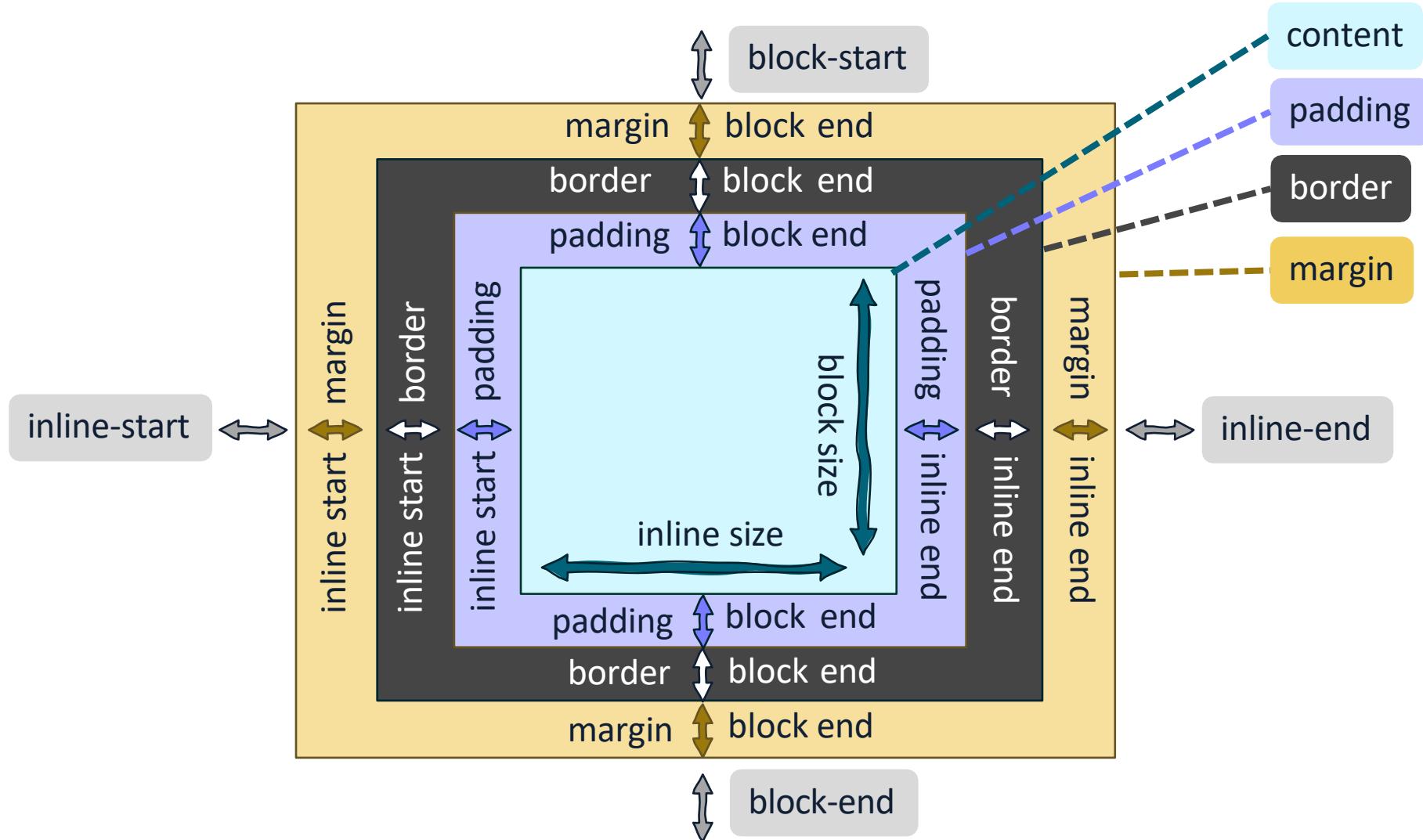


THE DISPLAY PROPERTY: BLOCK

- `display: block` posiziona il box a capo (su una nuova riga/linea)
- Paragraphs `<p>` and headings `<hx>` are typically displayed as blocks in default user agent styles
- Unless specified otherwise, blocks expand to the entire size of the inline dimension



THE BOX MODEL

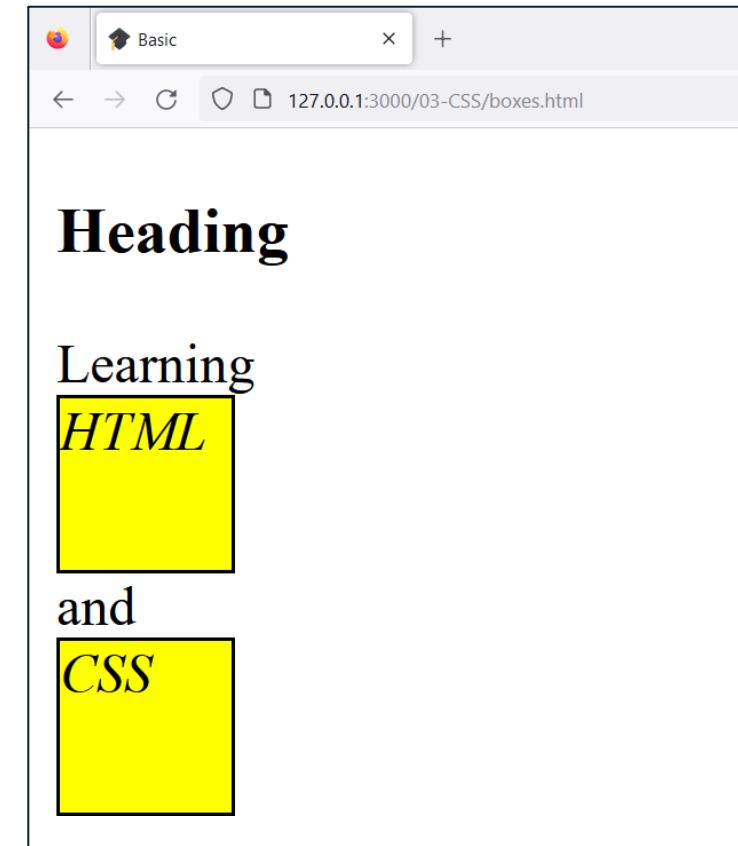


THE DISPLAY PROPERTY: INLINE VS BLOCK

- What happens if we use display **display: block** on the ****s?

```
em {  
    display: block;  
    background: yellow;  
    border: 1px solid;  
    width: 50px; height: 50px;  
}
```

```
<h3>Heading</h3>  
<p>  
    Learning <em>HTML</em> and <em>CSS</em>  
</p>
```

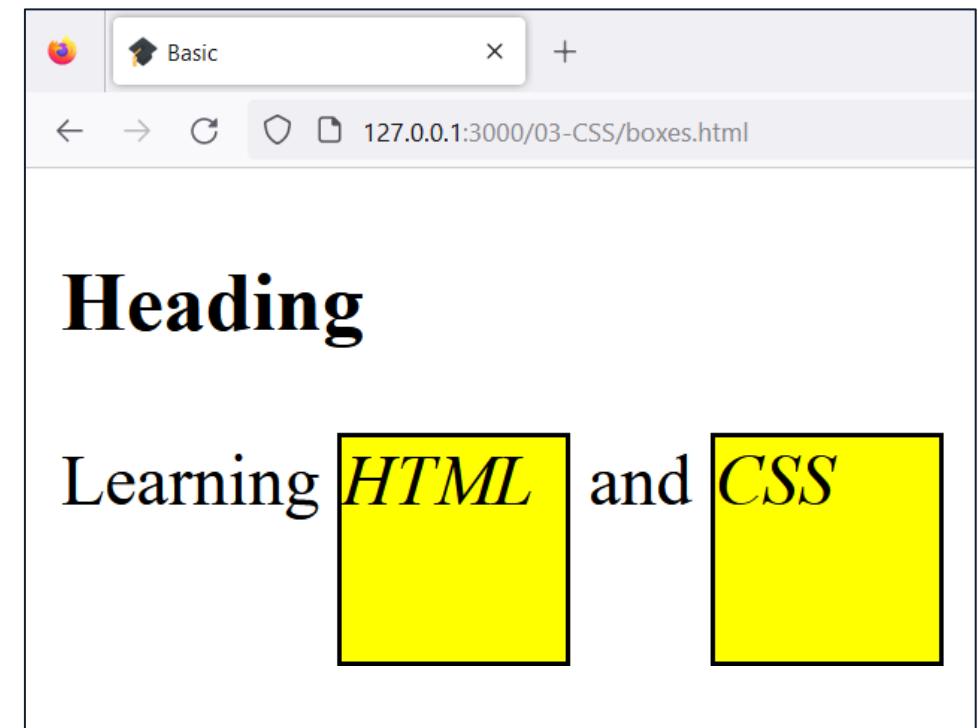


THE DISPLAY PROPERTY: INLINE-BLOCK

- `display: inline-block` is the same as `inline`, but allows also to set a width and a height

```
em {  
    display: inline-block;  
    background: yellow;  
    border: 1px solid;  
    width: 50px; height: 50px;  
}
```

```
<h3>Heading</h3>  
<p>  
    Learning <em>HTML</em> and <em>CSS</em>  
</p>
```

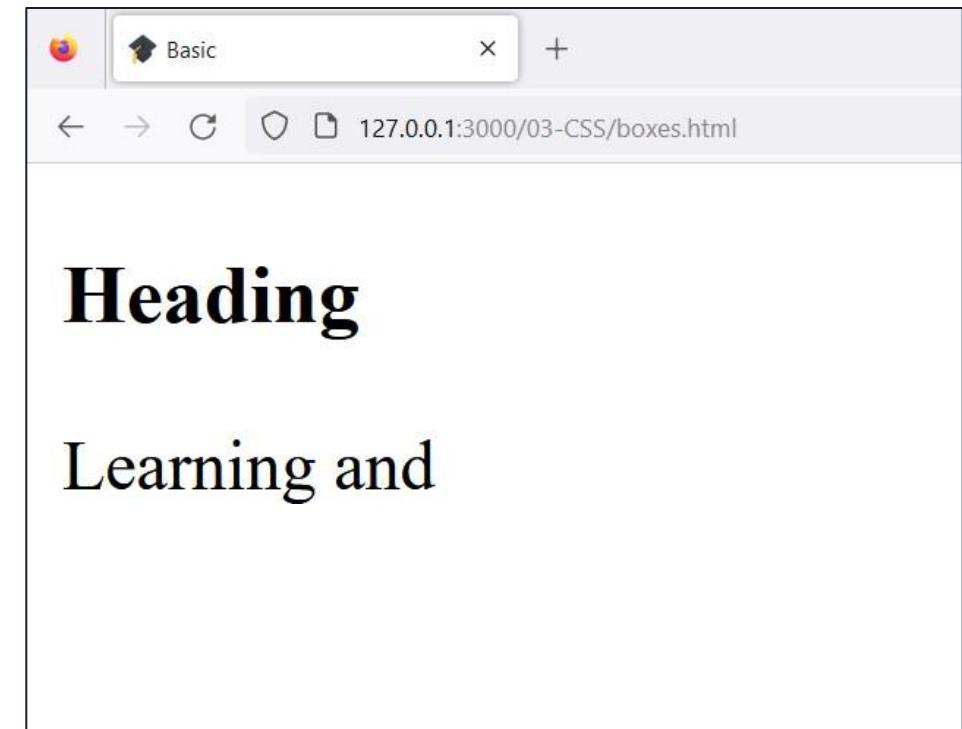


THE DISPLAY PROPERTY: NONE

- `display: none` can be used to completely **remove** the element from the visualization

```
em {  
    display: none;   
    background: yellow;  
    border: 1px solid;  
    width: 50px; height: 50px;  
}
```

```
<h3>Heading</h3>  
<p>  
    Learning <em>HTML</em> and <em>CSS</em>  
</p>
```



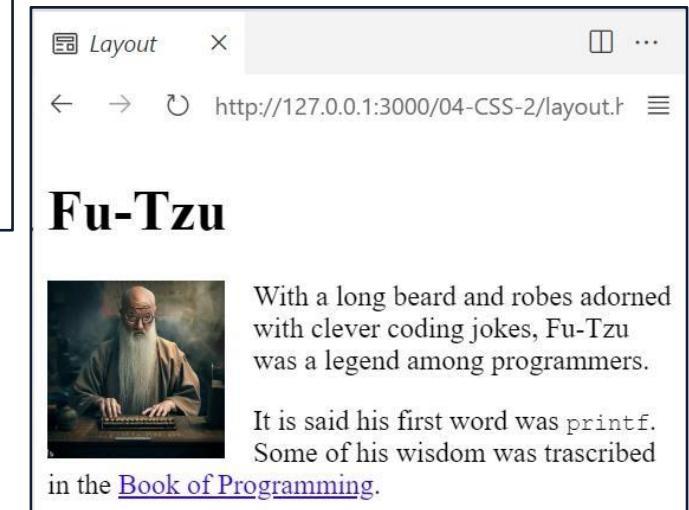
FLOATS

La proprietà float può essere usata per far fluttuare gli elementi in una data direzione e avere i figli successivi che occupano lo spazio circostante

```
<h1>Fu-Tzu</h1>

<p>With a long beard and robes adorned with clever
coding jokes, Fu-Tzu was a legend among programmers.</p>
<p>It is said his first word was <code>printf</code>.
Some of his wisdom was transcribed in the
<a href="index.html">Book of Programming</a>.</p>
```

```
img {
  width: 20vw; min-width: 100px;
  float: left; margin-right: 1rem;
} /7 in questo caso l'immagine «fluttua» tra il testo,
in modo da occupare un suo spazio senza sovrapporsi al
testo
```



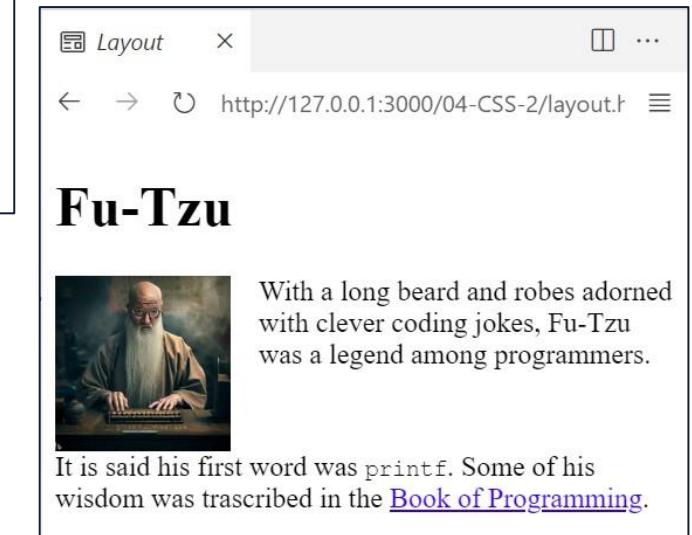
FLOATS

La proprietà clear può essere usata per prevenire questo modo di occupare lo spazio

```
<h1>Fu-Tzu</h1>

<p>With a long beard and robes adorned with clever
coding jokes, Fu-Tzu was a legend among programmers.</p>
<p>It is said his first word was <code>printf</code>.
Some of his wisdom was transcribed in the
<a href="index.html">Book of Programming</a>.</p>
```

```
img {
  width: 20vw; min-width: 100px;
  float: left; margin-right: 1rem;
}
p+p { clear: both; } //Qui mi sta dicendo di far
posizionare il testo sotto l'immagine per il prossimo
paragrafo
```



POSITIONING

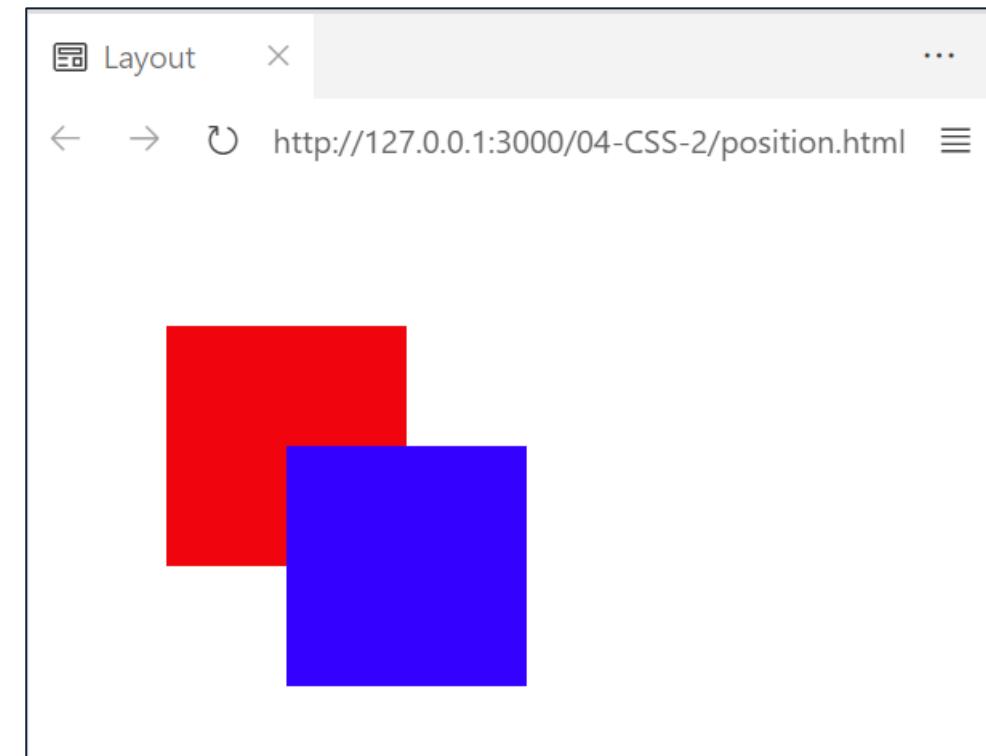
- Le proprietà di posizione possono cambiare il modo in cui un elemento si comporta nel flow del documento. Il valore standard per la proprietà è static.
- Other possible values are:
 - **relative**
 - **absolute**
 - **fixed**
 - **sticky**

POSITION: RELATIVE

- Elementi con **position: relative** sono posizionati in modo relativo rispetto alla loro posizione normale

```
div {  
    position: relative;  
    top: 50px; left: 50px;  
    background: red;  
    width: 100px; height: 100px;  
}  
div div {  
    background: blue;  
}
```

```
<div><div></div></div>
```

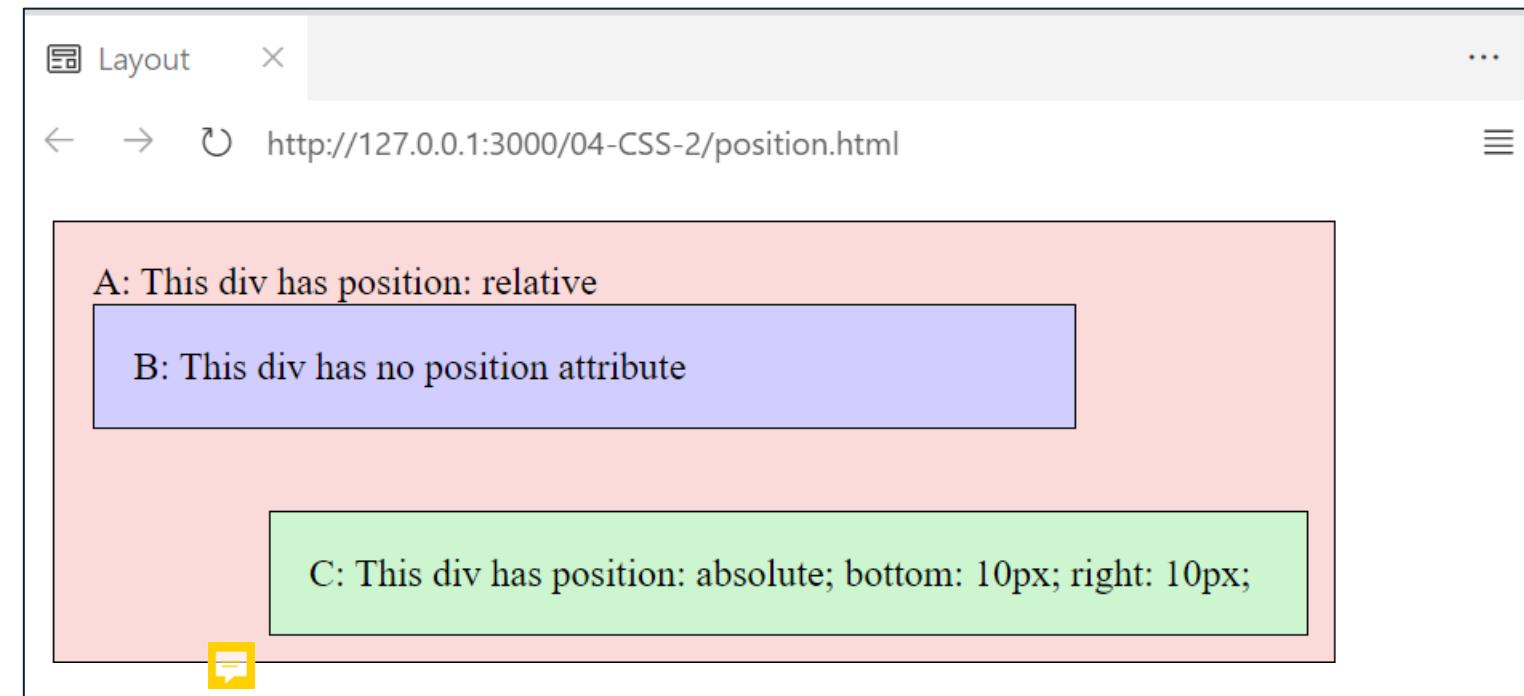


C ha una posizione assoluta rispetto al parente più vicino, che non è B, infatti non ha un attributo di position. C è collegato alla position di A

POSITION: ABSOLUTE

- Elementi con **position: absolute** sono posizionati rispetto al loro antenato più vicino secondo le posizioni relative

```
<div class="a">
  <!-- text -->
  <div class="b">
    <!-- text -->
    <div class="c">
      <!-- text -->
    </div>
  </div>
</div>
```

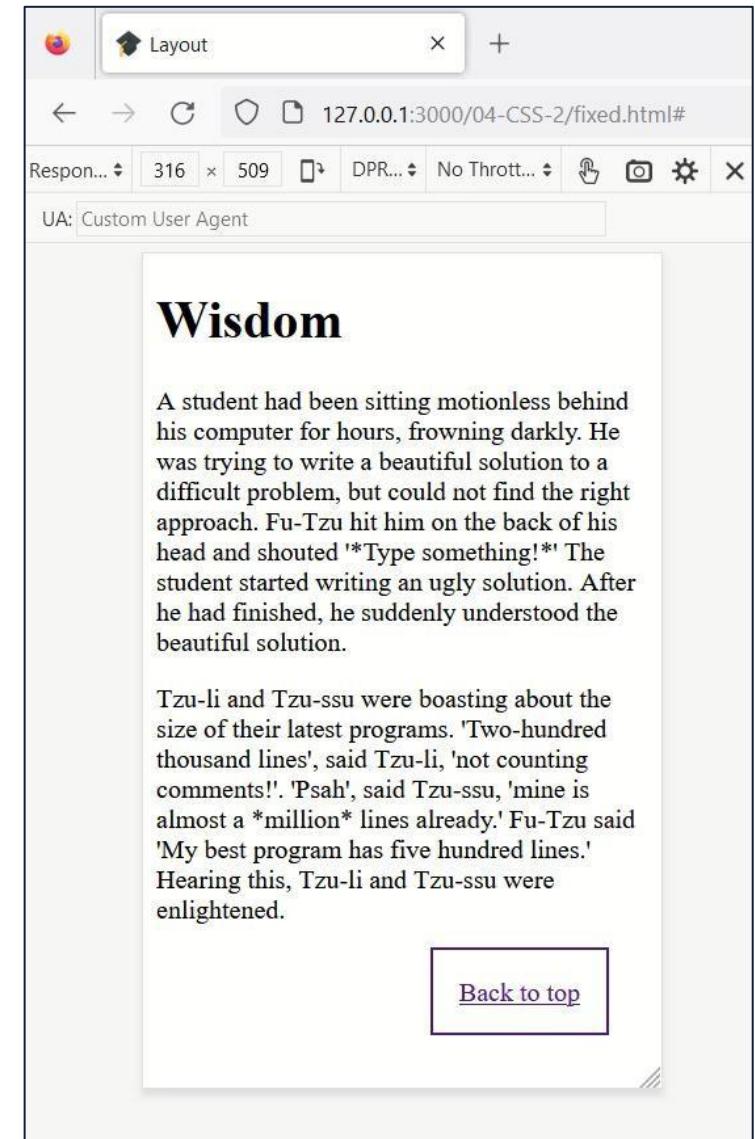


POSITION: FIXED

- Elementi con **position: fixed** sono posizionati rispetto al viewport

```
<h1>Wisdom</h1>
<p><!-- text --></p>
<p><!-- text --></p>
<a href="#">Back to top</a>
```

```
a {
  position: fixed;
  bottom: 2rem; right: 2rem;
  background-color: white;
  border: 2px solid;
  padding: 1em;
}
```



POSITION: STICKY

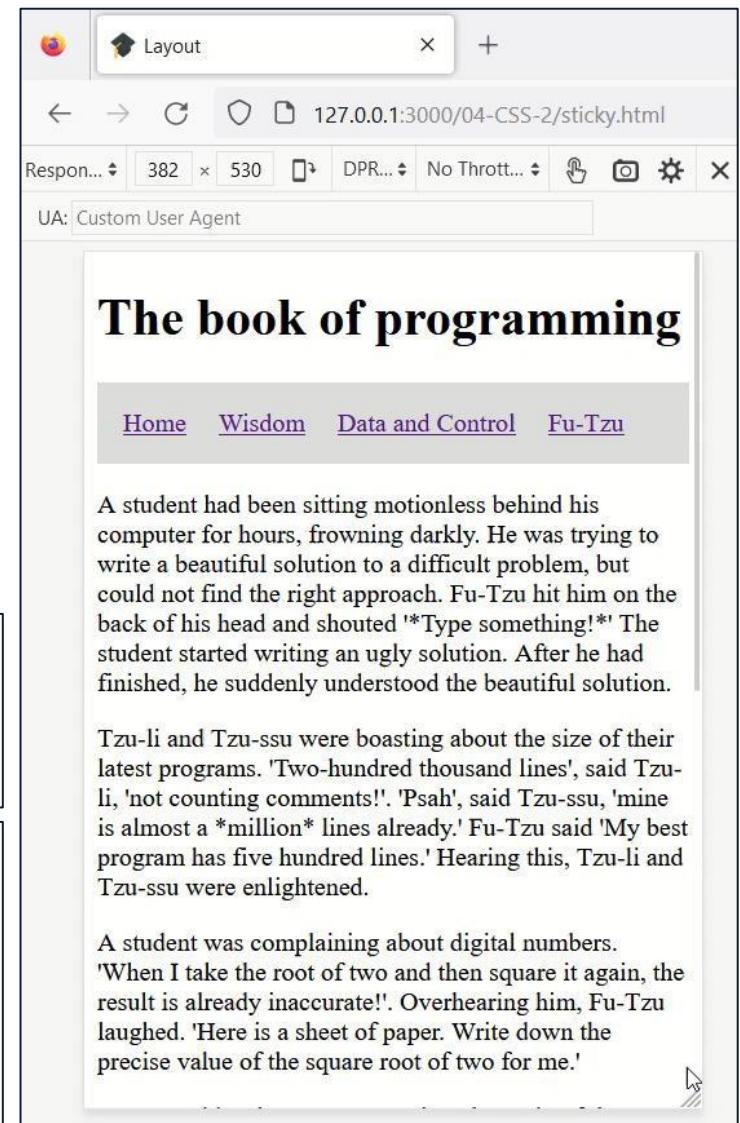
- Elementi con **position: sticky** si posizionano in base allo scroll dell'utente.
- È una sorta di ibrido tra relative/fixed: è relative finché non s'incrocia un threshold, è fixed finché non raggiunge il limite del suo parente

Beatrice Morgillo
2024-03-15 13:34:02

Una position sticky è una position che si adatta, per com'è definita nav, in questo caso, se l'utente non sta scrollando la pagina, ha una position relative, quindi non si adatta nè si muove, appena l'utente scrolla l'elemento compreso in nav va a posizione fissata (top) appena l'utente scrolla.

```
<h1>The book of programming</h1>
<nav><!-- links --></nav>
<p><!-- text --></p><p><!-- text --></p>

nav {
  padding: 1em; background: gainsboro;
  position: sticky; top: 0px;
}
nav a { padding-right: 1em; }
```

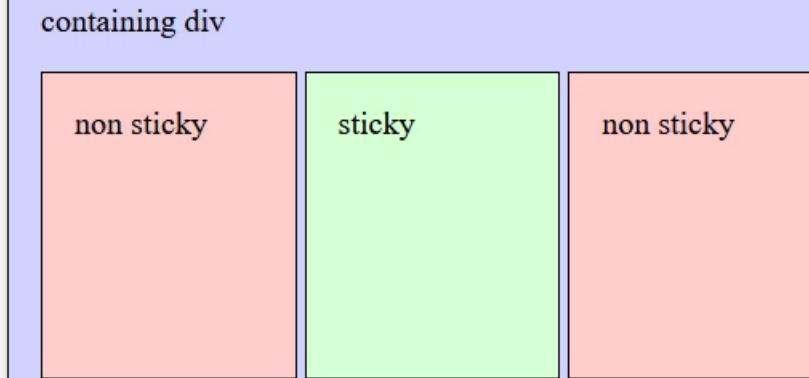


POSITION: STICKY

```
<div class="container">  
  <p>containing div</p>  
  <div class="item"></div>  
  <div class="item sticky"></div>  
  <div class="item"></div>  
</div>
```

```
.sticky {  
  position: sticky;  
  top: 10px;  
  background: rgb(212, 255, 212);  
}
```

Sticky Example



MODERN CSS LAYOUTS: FLEX AND GRID

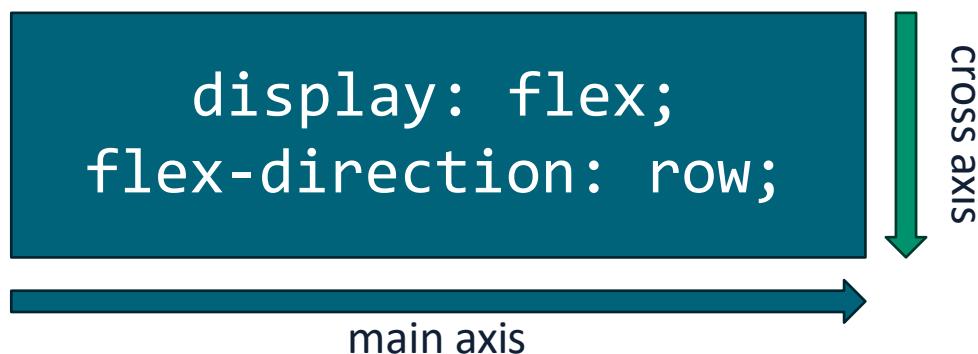
I 2 meccanismi principali per gestire i layout sono Flexbox e Grid.
Flexbox è designato per layout a una dimensione (o verticale od orizzontale).

Il Grid è designato per layout a due dimensioni (verticale e orizzontale)



FLEXBOX: FLEX CONTAINERS

I container flex sono dichiarati usando `display:flex`. Sono elementi a livello di blocco, con figli chiamati flex item. Ogni container ha un asse principale e uno incrociato. L'asse principale è impostato usando la `flex-direction`. L'asse incrociato è ortogonale all'asse principale.

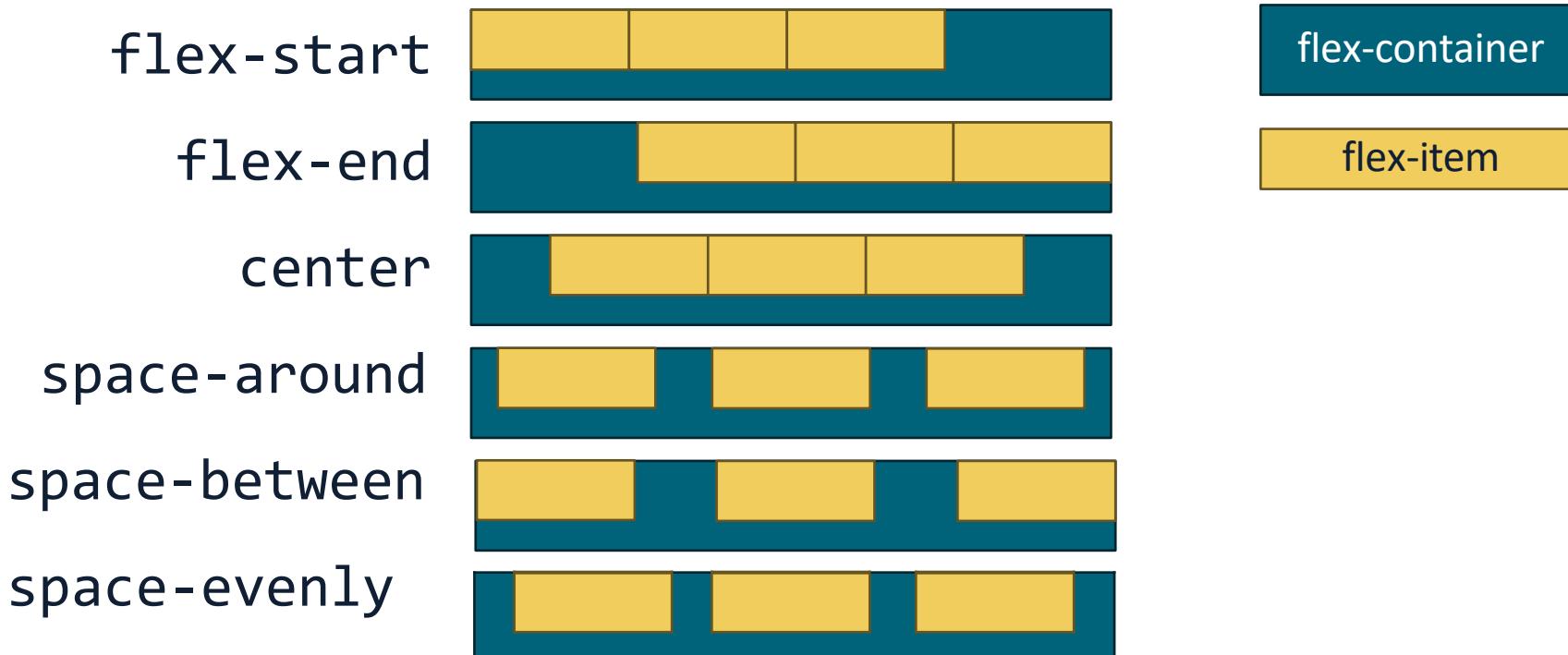


FLEXBOX: FLEX ITEMS

- Il figlio del container flex è convertito in oggetti flex e supportano le proprietà flex, per specificare il loro comportamento nel flex container.
- Sono posizionati lungo l'asse principale. Si può impostare flex-grow in modo che l'oggetto si espanda per tutto lo spazio possibile nell'asse principale.
- Si può usare flex-shrink per controllare dove il flex item può ridursi in base alla dimensione dell'asse principale per fit nel container flex.
- L'overflow è possibile e può essere controllato usando flex-wrap nel container.

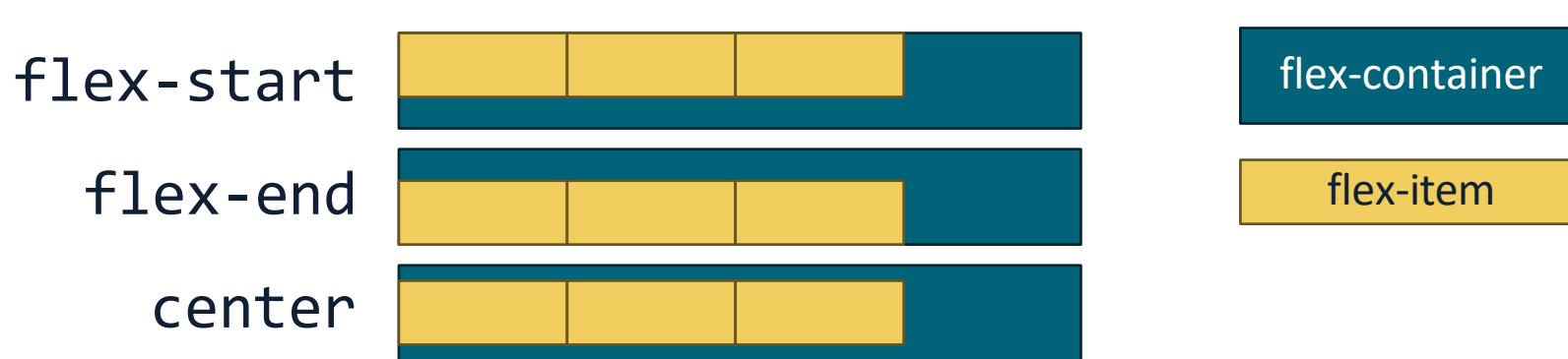
FLEXBOX: JUSTIFY-CONTENT

Il **justify-content** specifica come lo spazio libero dell'asse principale dev'essere gestito.



FLEXBOX: ALIGN-CONTENT

L'align-content specifica come lo spazio libero dell'asse incrociato dev'essere gestito.

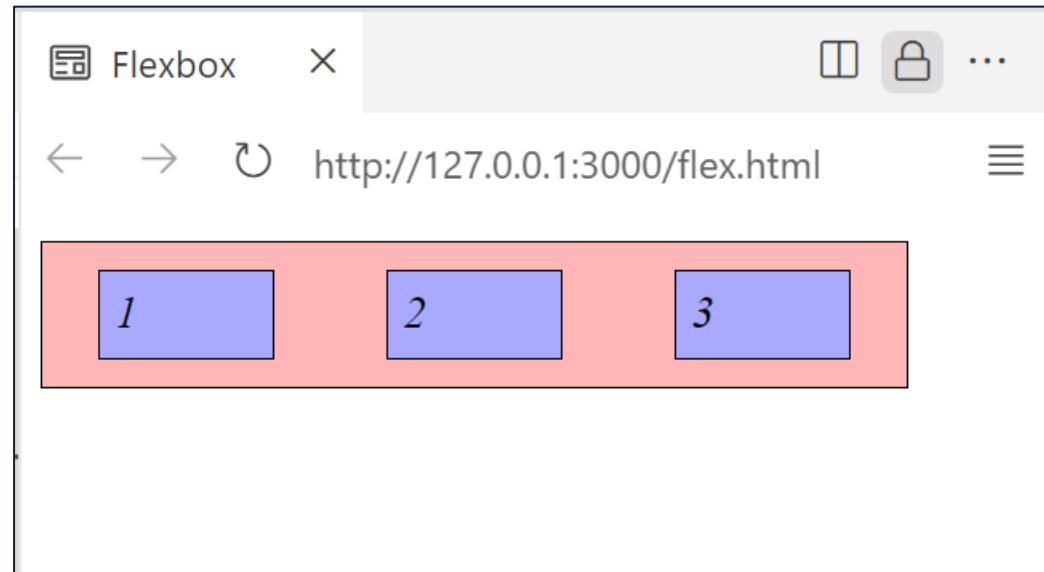


<https://flexbox.help/> is useful to visualize and practice with these properties

FLEXBOX: EXAMPLE

```
.container {  
    width: 300px;  
    height: 50px;  
    background: rgb(255, 182, 182);  
    display: flex;  
    justify-content: space-around;  
    align-items: center;  
}  
  
.item {  
    height: 20px; width: 50px;  
    padding: 5px;  
    background: rgb(169, 169, 255);  
}
```

```
<div class="container">  
    <div class="item special"><em>1</em></div>  
    <div class="item "><em>2</em></div>  
    <div class="item"><em>3</em></div>  
</div>
```

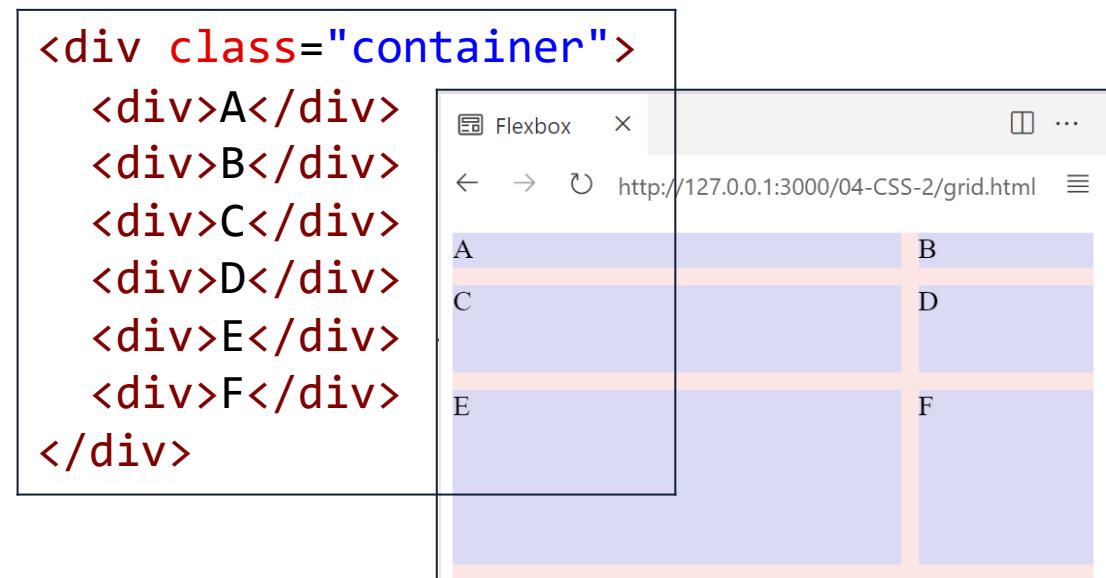


GRID

Grid si basa su righe e colonne, questi container si dichiarano con `display:grid`, il figlio diretto di un grid container è comunque un oggetto grid. I container definiscono numeri e dimensione delle loro righe e colonne.

```
.container {  
  display: grid;  
  /* two columns */  
  grid-template-columns: 1fr 100px;  
  /* three rows */  
  grid-template-rows: 20px 50px 100px;  
  gap: 10px;  
}
```

```
<div class="container">  
  <div>A</div>  
  <div>B</div>  
  <div>C</div>  
  <div>D</div>  
  <div>E</div>  
  <div>F</div>  
</div>
```



GRID: THE FR UNIT

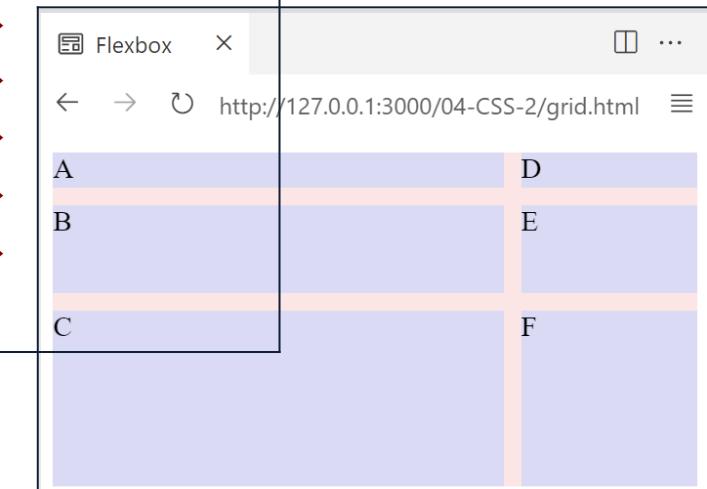
- L'unità fr è un'unità relativa che lavora solo nei grid. Rappresenta la lunghezza flessibile che corrisponde alla condivisione dello spazio occupabile e lavora similmente alla proprietà flex.

GRID: PLACEMENT OF GRID-ITEMS

Di base gli item grid sono posizionati lungo le righe ed è possibile piazzare item lungo le colonne con la proprietà **grid-auto-flow: column**

```
.container {  
  display: grid;  
  /* two columns */  
  grid-template-columns: 1fr 100px;  
  /* three rows */  
  grid-template-rows: 20px 50px 100px;  
  grid-auto-flow: column;  
  gap: 10px;  
}
```

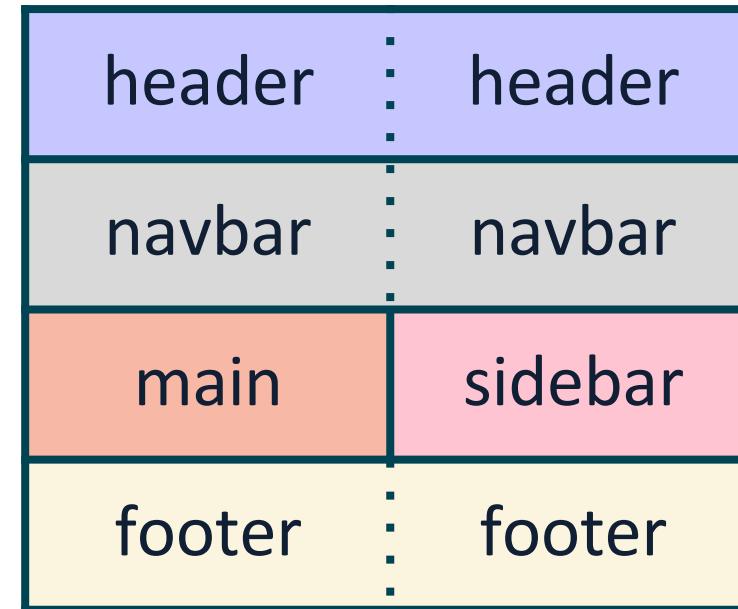
```
<div class="container">  
  <div>A</div>  
  <div>B</div>  
  <div>C</div>  
  <div>D</div>  
  <div>E</div>  
  <div>F</div>  
</div>
```



GRID: TEMPLATE AREAS

- È anche possibile assegnare nomi alle aree del grid e si possono piazzare grid item in una data area con la proprietà **grid-area**

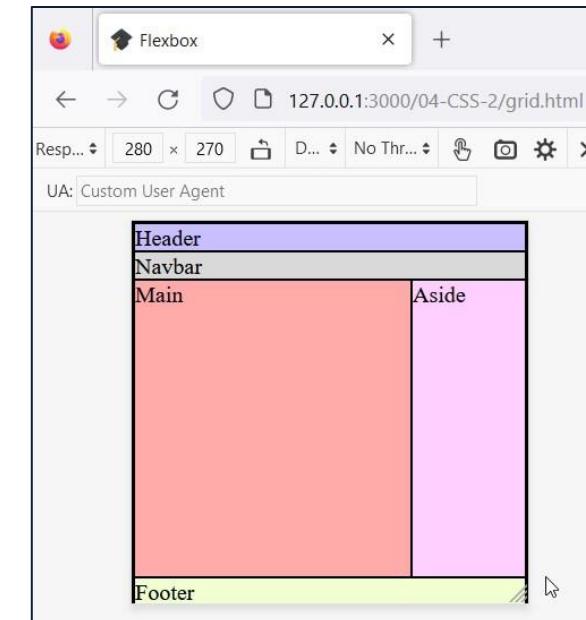
```
display: grid;  
grid-template-columns: 70vw 1fr;  
grid-template-rows: auto auto 1fr auto;  
grid-template-areas:  
  "header header"  
  "navbar navbar"  
  "main sidebar"  
  "footer footer";  
height: 100vh;  
margin: 0;
```



GRID: USING TEMPLATE AREAS

```
* {border: 1px solid black;}  
body {  
    display: grid;  
    grid-template-columns: 70vw 1fr;  
    grid-template-rows: auto auto 1fr auto;  
    grid-template-areas:  
        "header header"  
        "navbar navbar"  
        "main sidebar"  
        "footer footer";  
    height: 100vh; margin: 0;  
} /* background colors omitted */  
header { grid-area: header; }  
nav { grid-area: navbar; }  
main { grid-area: main; }  
aside { grid-area: sidebar; }  
footer { grid-area: footer; }
```

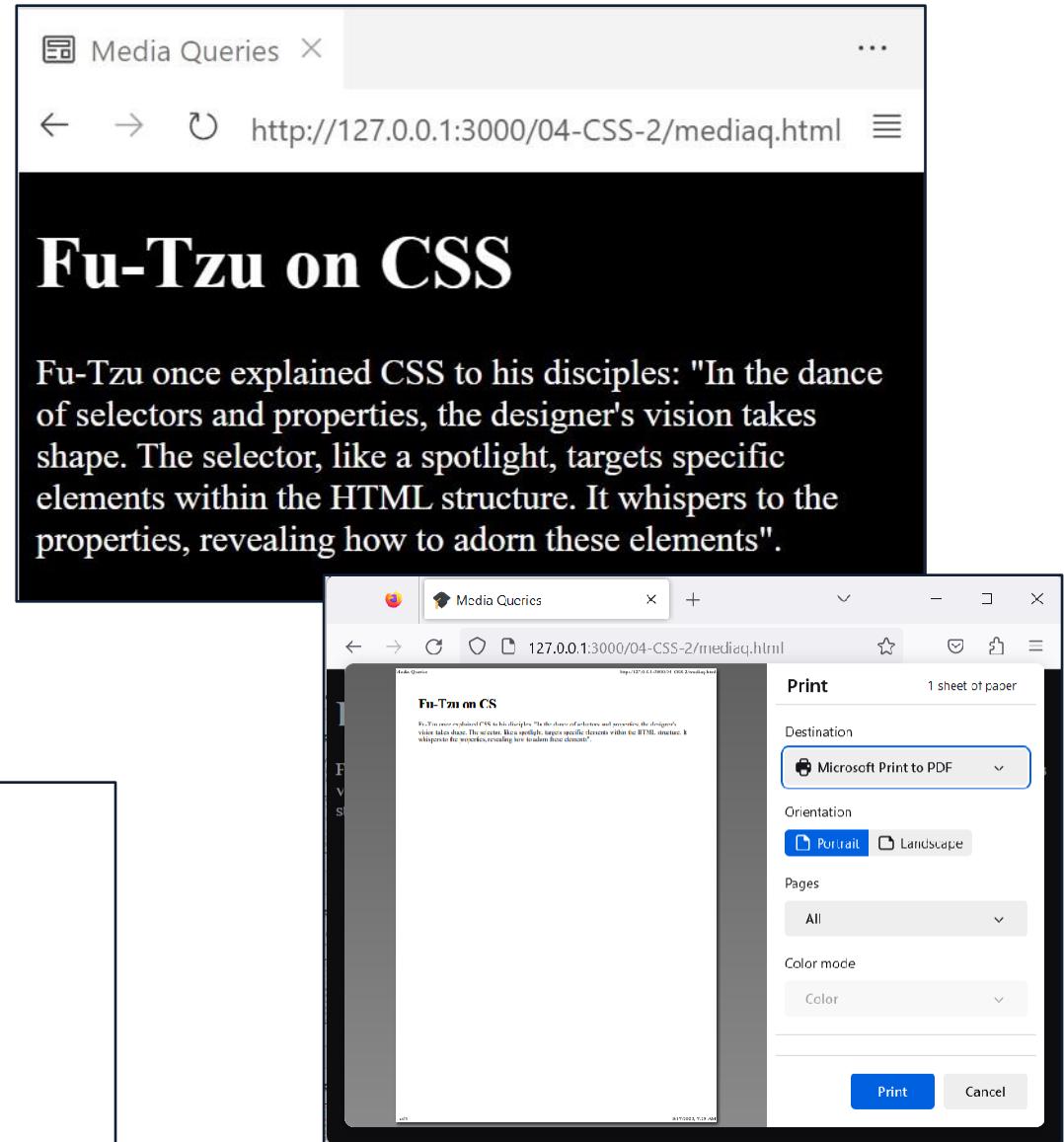
```
<body>  
    <header>Header</header>  
    <nav>Navbar</nav>  
    <main>Main</main><aside>Aside</aside>  
    <footer>Footer</footer>  
</body>
```



MEDIA QUERIES

- Le media query si possono applicare a certi stili CSS. Solo quando il dispositivo che sta visualizzando il contenuto ha caratteristiche specifiche, inizializzate con @media.

```
body {color: white; background: black;}  
@media print {  
    body {  
        color: black; background: transparent;  
    }  
}
```



MEDIA QUERIES: TYPES OF OUTPUT

Il CSS moderno supporta 3 tipi di output in media query:

- Print è inteso per impaginare materiali e documenti visualizzati su uno schermo in modalità di preview
- Screen è inteso per dispositivi che visualizzano il documento su uno schermo
- All si applica a tutti i dispositivi di output

MEDIA QUERIES: MEDIA FEATURES

- Le media query possono anche fare il test di caratteristiche specifiche dell'user agent, il dispositivo di output o ambiente (media features)

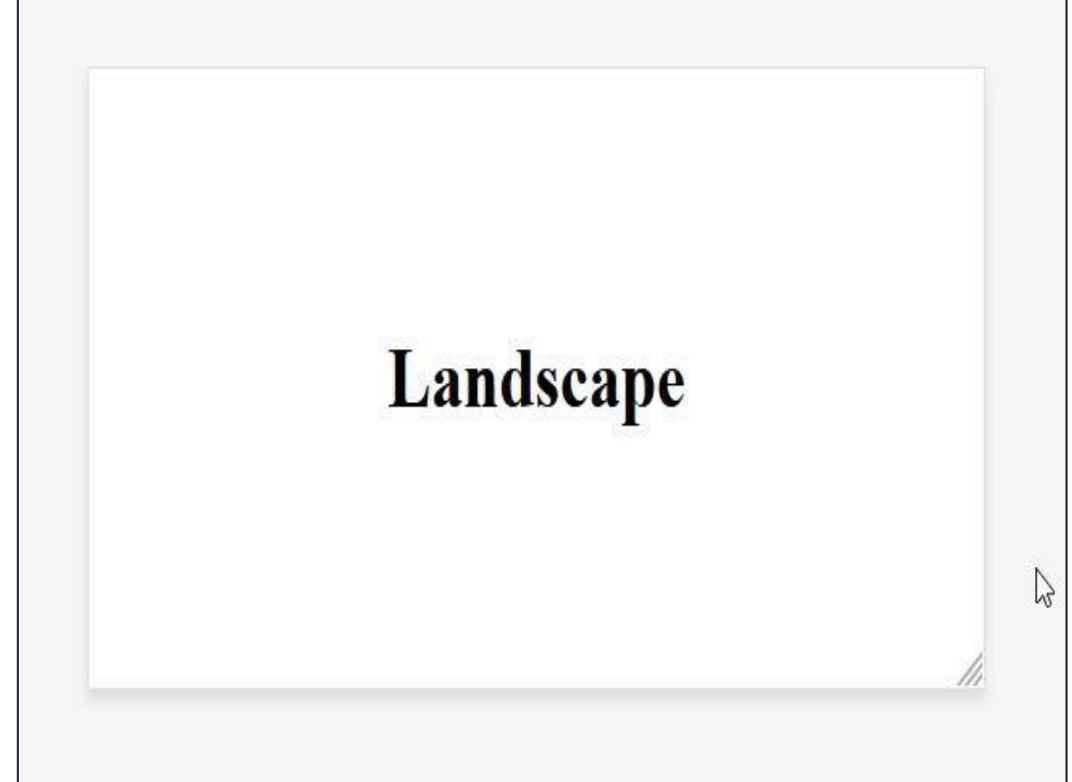
`@media media_type and (media_feature: value) and ...`

- Nella sintassi se `media_type` è omesso, allora verrà applicato all' `all` di default, le `media_features` devono essere incluse tra parentesi e sono opzionali

MEDIA QUERIES: EXAMPLES

```
@media screen and (orientation: landscape) {  
    h1::after {  
        content: "Landscape";  
    }  
}  
  
@media screen and (orientation: portrait) {  
    h1::after {  
        content: "Portrait";  
    }  
}
```

```
<body><h1></h1></body>
```



MEDIA QUERIES: EXAMPLES

```
@media (max-width: 600px) {  
    h1::after {content: "XSmall";}  
}  
  
@media (min-width: 600px) {  
    h1::after {content: "Small";}  
}  
  
@media (min-width: 768px) {  
    h1::after {content: "Medium";}  
}  
  
@media (min-width: 992px) {  
    h1::after {content: "Large";}  
}  
  
@media (min-width: 1200px) {  
    h1::after {content: "XLarge";}  
}
```

```
<body><h1></h1></body>
```



XSmall

RESPONSIVE DESIGN

FIXED-WIDTH LAYOUTS

- In the late 1990s, most monitors were 640px wide and 480px tall
- In the early days of web design, it was a safe bet to design pages with a 640px width.



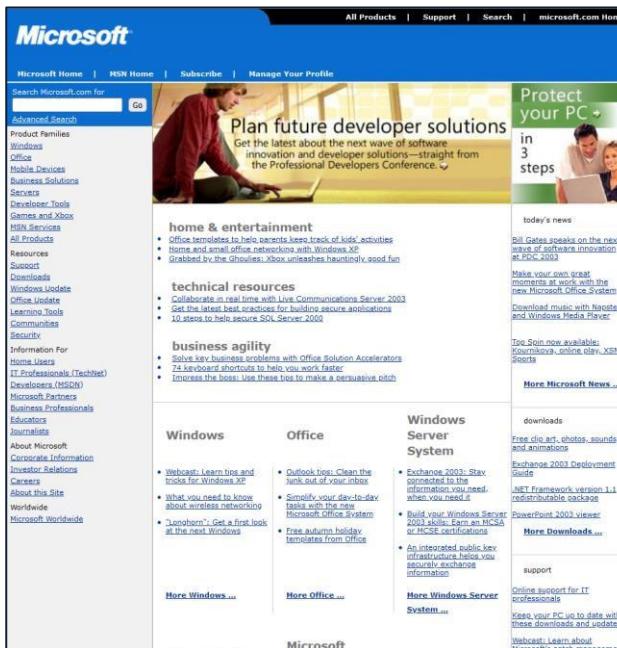
The Microsoft website,
from 1999 (640px wide)



Illustration: Freepik.com

FIXED-WIDTH LAYOUTS

- Then the monitors started getting bigger...
- Most monitors became 800x600, then 1024x768, ...
- Web designers updated their fixed-width design accordingly



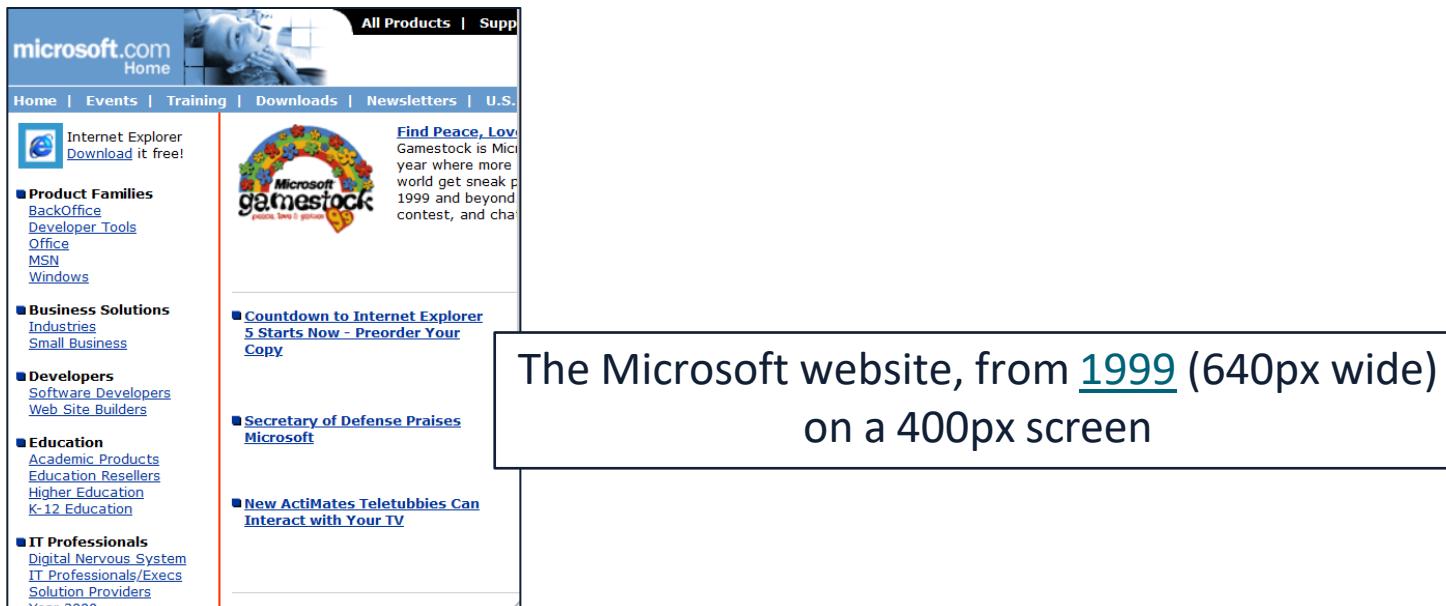
The Microsoft website, from 2003 (800px)



The Microsoft website, from 2005 (1024px)

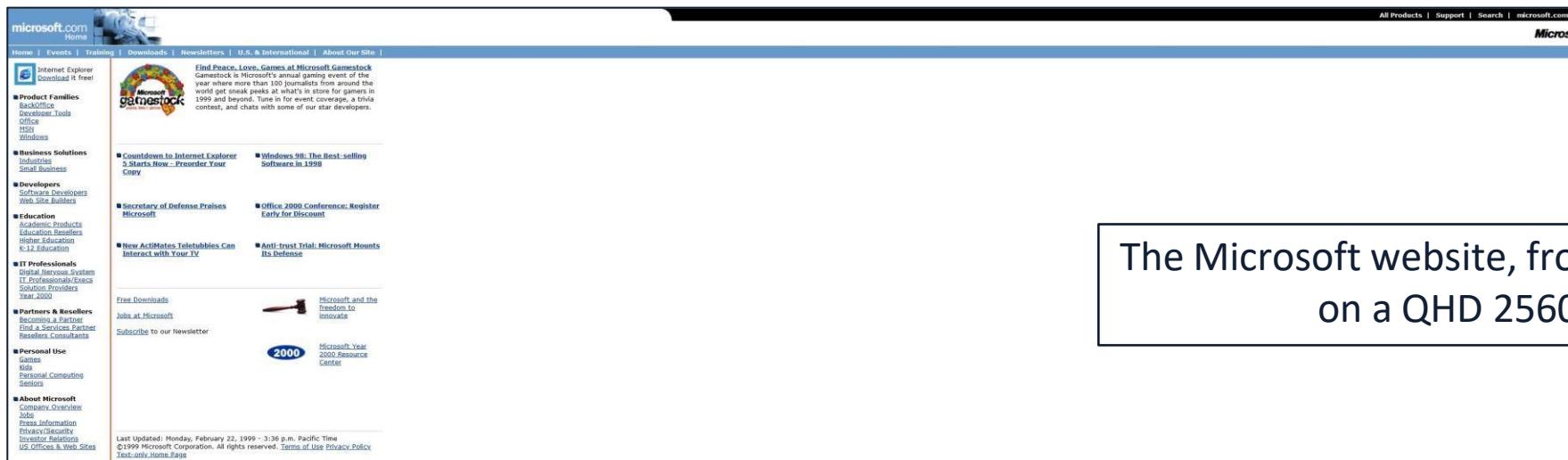
FIXED-WIDTH LAYOUTS

- Whether it is 640, 800, or 1024 pixels, working with a single specific width is called **fixed-width** design
- Fixed-width design will look good only at the considered width.
 - Smaller screens won't be able to see the entire page without horizontal scrolling



FIXED-WIDTH LAYOUTS

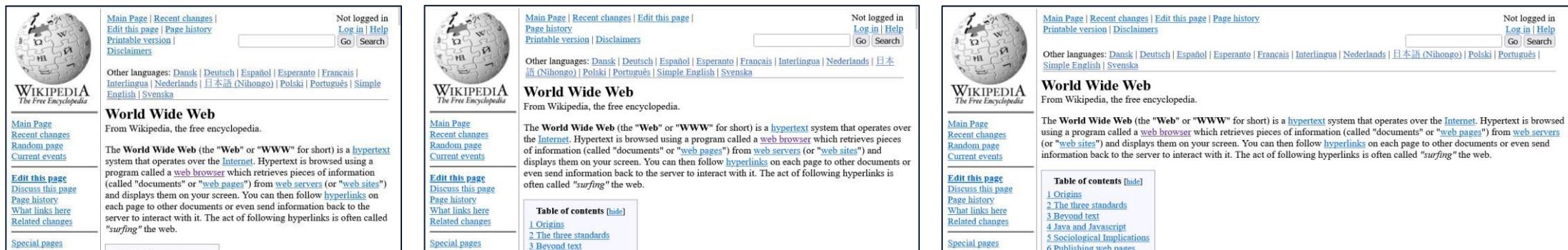
- Whether it is 640, 800, or 1024 pixels, working with a single specific width is called **fixed-width** design
- Fixed-width design will look good only at the considered width.
 - Smaller screens won't be able to see the entire page without horizontal scrolling
 - Larger screens will have a lot of (wasted) space



The Microsoft website, from [1999](#) (640px wide)
on a QHD 2560px screen

LIQUID (FLUID) LAYOUTS

- Mentre molti designer web usavano layout e fixed-width alcuni hanno reso i loro layout più flessibili. Non c'erano flexbox, grid o media query, lo facevano usando percentuali come le larghezze di una colonna. I layout sono chiamati layout liquidi.



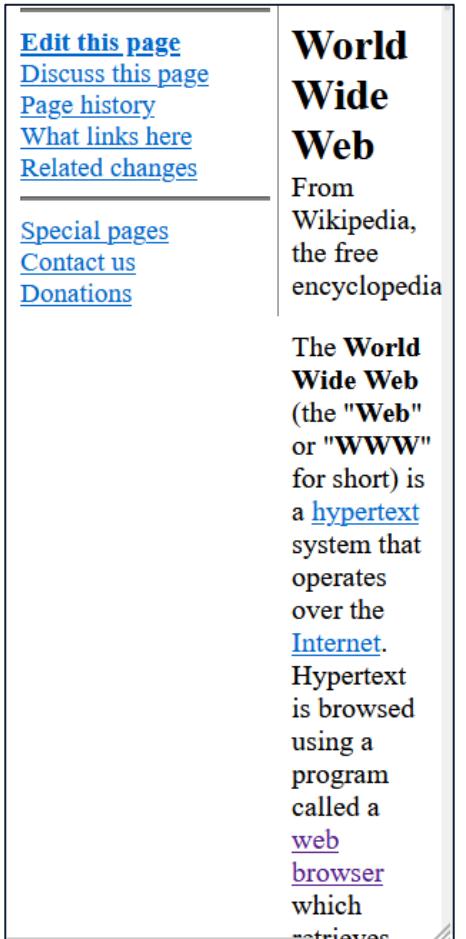
The Wikipedia website from 2004, viewed in a 640px, 800px, and 1024px screens

LIQUID (FLUID) LAYOUTS

- I layout liquidi sembrano buoni tra un'ampia gamma di larghezze, ma è peggiorato agli estremi, o troppo striminzito su schermi piccoli o troppo largo su schermi grandi.



The screenshot shows the Wikipedia homepage from 2004. The layout is fluid, adapting to the wide screen. At the top, there's a navigation bar with links like "Main Page", "Recent changes", "Edit this page", and "Page history". Below the navigation is a large section titled "World Wide Web" with a sub-section about its history and how it operates. On the left side, there's a sidebar with links for "Main Page", "Recent changes", "Random page", "Current events", "Edit this page", "Discuss this page", "Page history", "What links here", "Related changes", "Special pages", and "Contact us". The main content area contains text and some small images.



The screenshot shows the same Wikipedia homepage from 2004, but viewed in a much narrower 250px wide screen. The layout is still fluid but appears compressed and less readable. The sidebar on the left is very narrow, and the main content area is also reduced in width, making the text smaller and harder to read. The right sidebar and footer are also affected by the limited width.

World Wide Web
From Wikipedia, the free encyclopedia

The World Wide Web (the "Web" or "WWW" for short) is a hypertext system that operates over the Internet. Hypertext is browsed using a program called a web browser which retrieves pieces of information (called "documents" or "web pages") from web servers (or "web sites") and displays them on your screen. You can then follow hyperlinks on each page to other documents or even send information back to the server to interact with it. The act of following hyperlinks is often called "surfing" the web.

[Edit this page](#)
[Discuss this page](#)
[Page history](#)
[What links here](#)
[Related changes](#)

[Special pages](#)
[Contact us](#)
[Donations](#)

Not logged in
[Log in](#) | [Help](#)
[Go](#) | [Search](#)

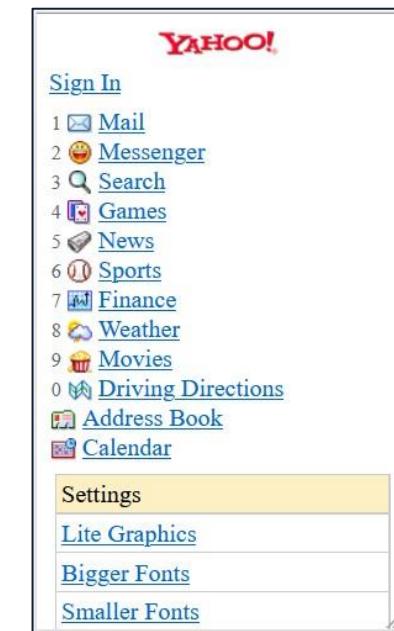
The Wikipedia website from [2004](#), viewed in a 2560px (left) and in a 250px (right) screens

SEPARATE WEBSITES

Poi sono arrivati i dispositivi mobili con uno schermo largo minimo 240px, pertanto i layout fluidi non lavoravano bene e si dovevano sviluppare siti web separati.



yahoo.com, 2006, on a 800px screen



wap.oa.yahoo.com, 2006, on a 240px screen

SEPARATE WEBSITES

Tipicamente sfruttavano lo sniffing user-agent (UA) per reindirizzare il dispositivo alla versione mobile del sito che era hostato in un sottodomino.

Ha i suoi contro, infatti:

- UA sniffing è poco sicuro (potrebbe fallire nel reidirizzamento)
- Ogni sitoweb separato dev'essere manutenuto
- Oggi la separazione di sitiweb è sfocata

Sarebbe bello avere un unico sitoweb che renderizzi in modo differente in base al dispositivo vedente.

ADAPTIVE LAYOUTS

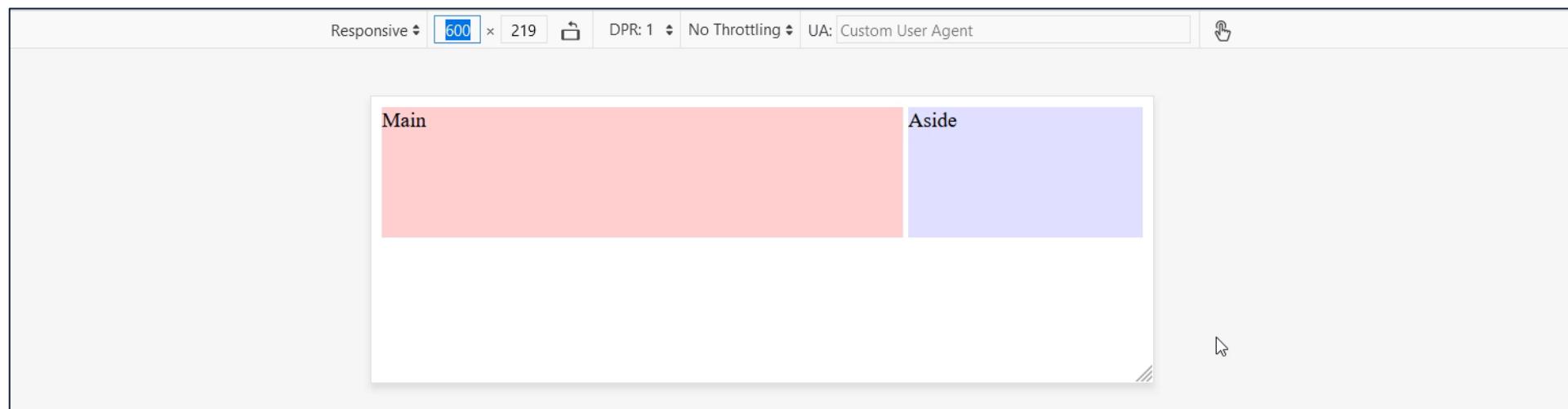
Inizialmente i web designer erano comodi con i layout a larghezza fissa. Hanno sviluppato sitiweb che cambiavano tramite un sistema di design a larghezza fissa usando le media queries. Quest'approccio è chiamato design adattivo.

ADAPTIVE LAYOUT: EXAMPLE

```
@media (max-width: 800px) {  
    main {width: 400px;}  
    aside {width: 180px;}  
}  
  
@media (min-width: 800px) {  
    main {width: 500px;}  
    aside {width: 280px;}  
}
```

```
main {  
    display: inline-block;  
    height: 200px;  
}  
  
aside {  
    display: inline-block;  
    height: 200px;  
}
```

```
<body>  
<main>Main</main>  
<aside>Aside</aside>  
</body>
```



ADAPTIVE LAYOUTS

I layout adattivi (e anche i responsivi) sono un mix di media query e di layout a larghezza fissa.

I design responsivi sono caratterizzati da: container fluidi, media fluidi e media query.

Layout e immagini di un sito responsivo dovrebbero essere ben visibili su ogni dispositivo.

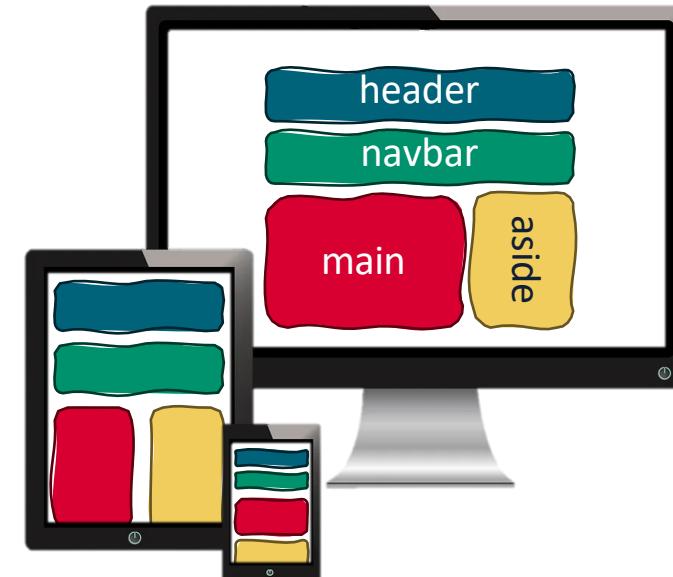
RESPONSIVE LAYOUTS

- Adaptive layouts are a mashup of media queries + fixed-width layouts
- **Responsive layouts are a mashup of media queries + liquid layouts**
- The term was coined by [Ethan Marcotte in an article](#) in 2010

Responsive design is characterized by:

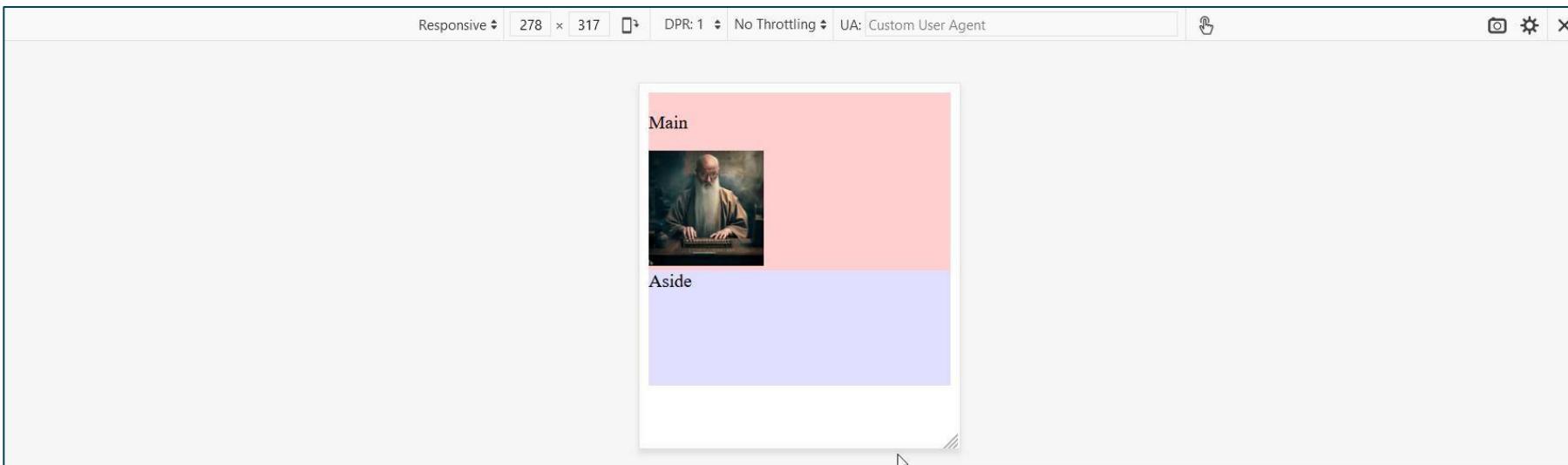
- Fluid containers
- Fluid media
- Media Queries

Layout and images of a responsive site
should look good on **any** device



RESPONSIVE LAYOUT: EXAMPLE

```
body { display: flex; justify-content: center; flex-wrap: wrap; }
@media (max-width: 600px){ main {width: 100%} aside {width: 100%}}
@media (min-width: 600px){ main {width: 60%} aside {width: 40%}}
@media (min-width: 768px){ main {width: 70%} aside {width: 20%}}
main {display: inline-block;}
aside {display: inline-block; min-height: 100px;}
img {width: 20%; min-width: 100px;}
```



RESPONSIVE LAYOUT: VIEWPORT META TAG

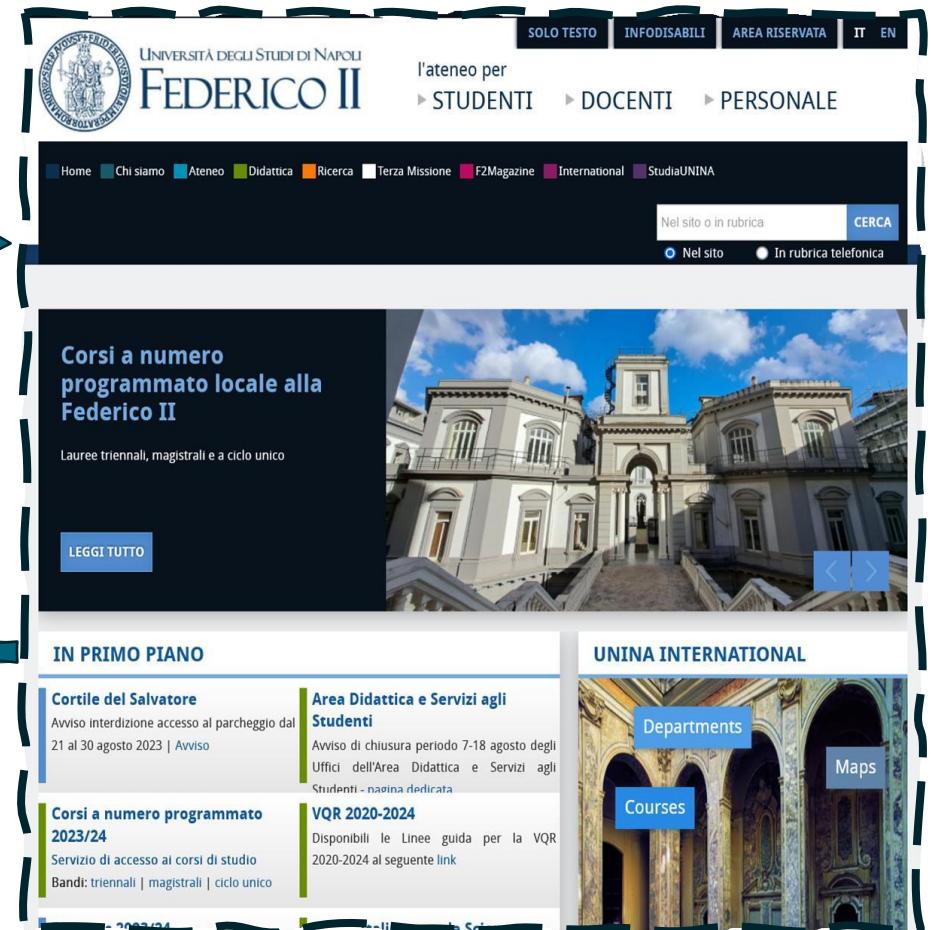
I primi browser per mobile hanno dovuto a che fare con sitiweb che erano designati per schermi più ampi.

Credevano che i sitiweb erano designati per uno schermo largo 980px e visualizzavano le pagine in una viewport virtuale di 980px, al che hanno dovuto scalare il rendering delle pagine per rientrare nell'effettiva larghezza dello schermo

RESPONSIVE LAYOUT: VIEWPORT META TAG



1: Render in virtual viewport



The screenshot shows the responsive layout of the University of Naples Federico II website. The top navigation bar includes links for SOLO TESTO, INFODISABILI, AREA RISERVATA, IT, and EN. Below the navigation is a banner for 'Corsi a numero programmato locale alla Federico II'. Further down, there are sections for 'IN PRIMO PIANO' featuring news items about parking restrictions and course access, and 'UNINA INTERNATIONAL' with links to 'Departments' and 'Courses'.

2: Scale down to device

BlackBerry Torch 9800 (2010)
480x360 display

980px-wide Virtual Viewport

RESPONSIVE LAYOUT: VIEWPORT META TAG

Il meccanismo delle viewport virtuali avevano concesso ai sitiweb non ottimizzati per i mobile di essere visibili meglio su schermi più stretti. Comunque il meccanismo non funziona per i siti che sono ottimizzati per i dispositivi mobili.

Le media query che entrano in gioco a 640px non saranno mai usate a 980px!

Il meta tag HTML viewport permette di controllare questo meccanismo e le pagine web ottimizzate per i mobile dovrebbero includerlo nella loro <head>: <metaname="viewport" content="width=device-width,initial-scale=1">

Questo tag specifica 2 regole:

- width=device-width dice al browser di far finta che la larghezza di design del sitoweb sia quello del dispositivo usato
- initial-scale=1 dice al browser di non fare alcuno scaling.

RESPONSIVE LAYOUT: VIEWPORT META TAG

```
<meta name="viewport"  
      content="width=device-width, initial-scale=1">
```

The above viewport HTML meta tag specifies two rules:

- **width=device-width** tells the browser to assume that the width the website was designed for is the width of the device
- **initial-scale=1** tells the browser to do no scaling at all.

RECAP: A BRIEF HISTORY OF WEB LAYOUTS

FIXED-WIDTH LAYOUTS

- In the late 1990s, most monitors were 640px wide and 480px tall
- In the early days of web design, it was a safe bet to design pages with a 640px width.



The Microsoft website, from 1999 (640px wide)



LIQUID (FLUID) LAYOUTS

- While most web designers used fixed-width layouts, some made their layouts **more flexible**
- There was no flexbox, grid or media queries. They did that by using percentages as column widths. The layouts are called **liquid layouts**.
- The Wikipedia website did this



The Wikipedia website from 2004, viewed in a 640px, 800px, and 1024px screens

Luigi Libero Lucio Starace, Ph.D. - University of Naples Federico II - Web Technologies Course - Lecture 04 - CSS Layouts

48

SEPARATE WEBSITES

- Then, mobile devices arrived, with screens as small as 240px
 - Fixed-width and liquid layouts do not work well on those
- One option is to have a **separate website** for mobile devices



yahoo.com, 2006, on a 800px screen



wap.yahoo.com, 2006, on a 240px screen

50

ADAPTIVE LAYOUTS

- With the introduction of CSS media queries (~2009), more flexible layouts were possible
- Initially, web designers were still most comfortable with fixed-widths layouts
- They designed websites that switched between a handful of fixed-widths designs using media queries.
- This approach is called **adaptive design**



Luigi Libero Lucio Starace, Ph.D. - University of Naples Federico II - Web Technologies Course - Lecture 04 - CSS Layouts

52

RESPONSIVE LAYOUTS

- Adaptive layouts are a mashup of media queries + fixed-width layouts
- Responsive layouts are a mashup of media queries + liquid layouts**
- The term was coined by [Ethan Marcotte](#) in an article in 2010

Responsive design is characterized by:

- Fluid containers
- Fluid media
- Media Queries

Layout and images of a responsive site should look good on **any** device



Luigi Libero Lucio Starace, Ph.D. - University of Naples Federico II - Web Technologies Course - Lecture 04 - CSS Layouts

55

REFERENCES

- **Learn CSS**

web.dev

<https://web.dev/learn/css/>

Sections: 2, 8 to 11

- **Learn Responsive Design**

web.dev

<https://web.dev/learn/design/>

Sections: 1 to 3, 15

- **Game-based approaches to learning CSS Layouts**

<https://flexboxfroggy.com/> (Flexbox)

<https://cssgridgarden.com/> (Grid layouts)



UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
WEB TECHNOLOGIES – LECTURE 05

JAVASCRIPT: PART I

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



PREVIOUSLY, ON WEB TECHNOLOGIES

So far, we've learnt to create modern, beautiful web pages

- With **HTML** we define their structure
- With **CSS** we define their appearance on possibly different media

Still, our web pages are inherently **static**

- There is no way their content can change while we are browsing
(unless we edit the html file and re-load the page)

JAVASCRIPT

È un linguaggio di alto livello, che non prevede uno step di compilazione, sono interpretati come testo standard dall'Engine JS.

JAVASCRIPT

Ormai è usato non solo per le pagine web, è anche il linguaggio di programmazione più popolare, implementa le specifiche ECMAScript.

INCLUDING JAVASCRIPT IN WEB PAGES

Il codice JS può essere incluso nei documenti HTML in 2 modi:

- JS interno: il codice è inserito all'interno di un elemento `<script>` nella `<head>` o nel `<body>`
- JS esterno: usando `<script src="url/of/script.js">` (file esterno con l'estensione `.js`)

```
<script>
    console.log("Hello World!");
</script>
```

- **External JavaScript**

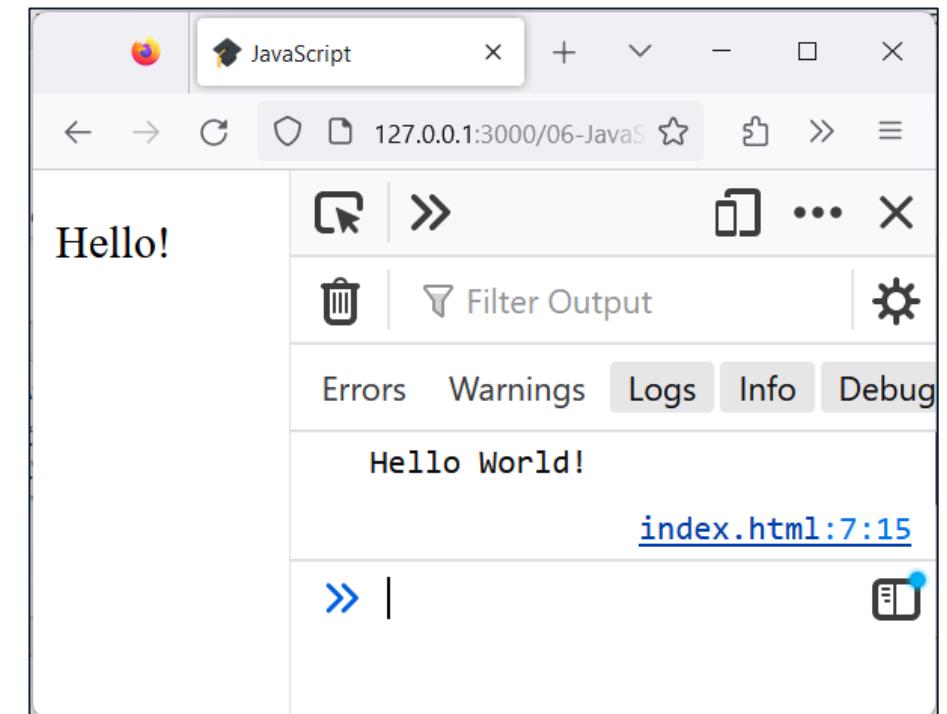
- Using `<script src="url/of/script.js">` (external file must have «`.js`» extension)

```
<script src="script.js"></script>
```

```
console.log("Hello World!");
```

JAVASCRIPT: HELLO WORLD!

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>JavaScript</title>
    <script>
      console.log("Hello World!");
    </script>
  </head>
  <body>
    <h1>Hello!</h1>
  </body>
</html>
```



MODERN JAVASCRIPT

JS è evoluto con cambi graduali e non molto impattanti, solo che dal 2009, con l'aggiunta di ECMAScript5 (ES5) ha disabilitato la retro-compatibilità automatica e si può implementare con “use strict” all'interno degli script. (Nel corso sarà fondamentale la versione moderna di “strict”).

```
"use strict";  
  
/* JavaScript code here*/
```

JAVASCRIPT: THE LANGUAGE

Supporta più paradigmi: imperativo, funzionale e orientato agli oggetti.

Ha alcune feature interessanti per la programmazione asincrona.

Un programma JS è una sequenza di statement:

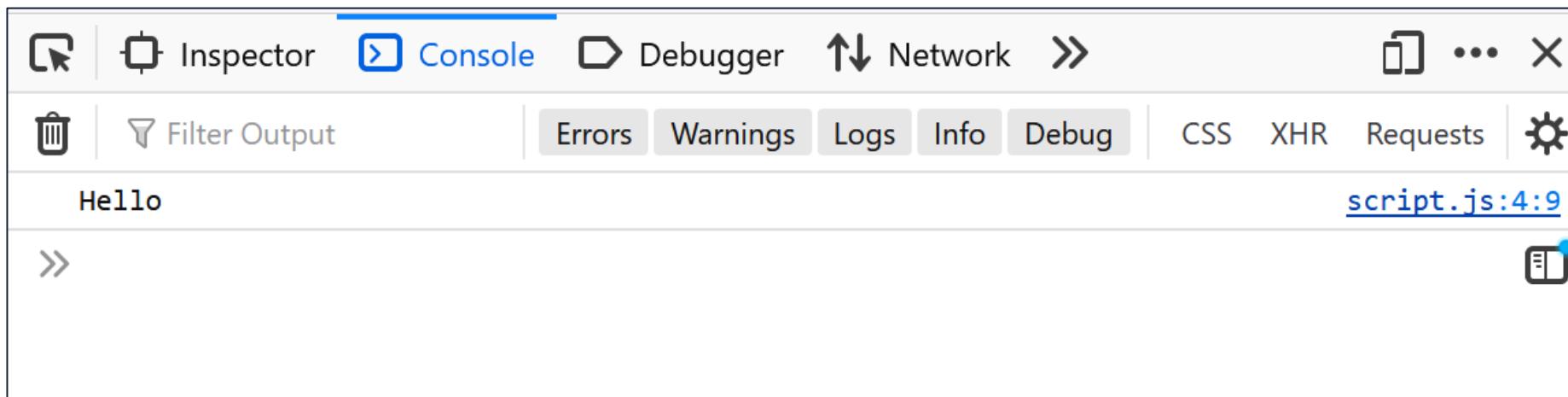
- Composti da valori, operatori, espressioni, keyword e commenti
- La sintassi è simile a Java e C (ma più permissiva)
- Gli statement sono delimitati dalle nuove linee e/o dalle ;

STATEMENTS

```
//with semicolons  
msg = "Hello";  
num = 1;  
console.log(msg);
```

```
//without semicolons  
msg = "Hello"  
num = 1  
console.log(msg)
```

```
/* statements on the same line  
(and multi-line comment) */  
  
msg="Hello"; num=1; console.log(msg);
```



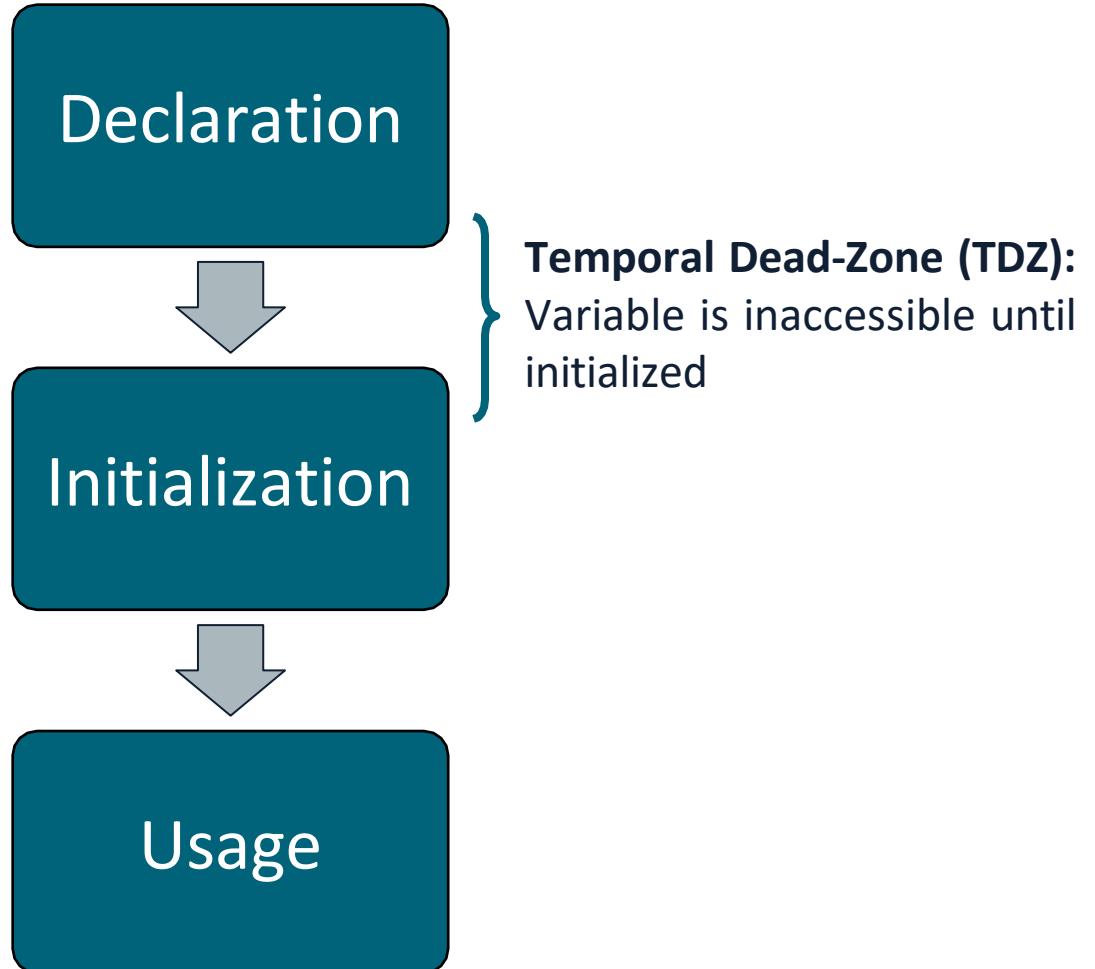
- Delimiting statements with a semicolon is a preferred **good practice**

VARIABLES: LIFECYCLE

Dichiarare una variabile in JS consiste di 3 passaggi distinti:

1. Dichiarazione: il nome della variabile è legato allo scope attuale
2. Inizializzazione: la variabile è inizializzata
3. Utilizzo: la variabile può essere referenziata

Esiste una Zona di morte temporanea (Temporal Dead-Zone - TDZ), la variabile è inaccessibile finché non è inizializzata ed è il tempo che passa dal punto 1 al punto 2.



VARIABLES: DECLARATION

In JS moderno, le variabili possono essere dichiarate e inizializzate usando:

- La keyword `let` per le variabili “tradizionali”
- Con `const` per le costanti, che non possono essere inizializzate di nuovo.

Tutte le variabili sono inizializzate con `undefined`.

Ci sono 3 livelli di scope:

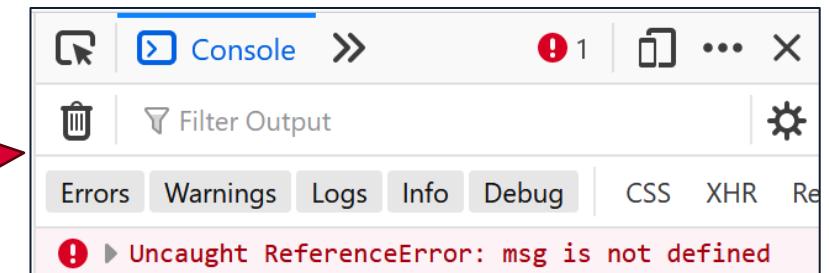
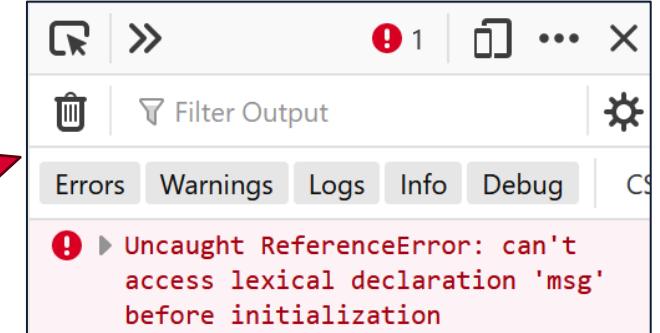
1. Globale (come un'intera finestra web)
2. Di funzione
3. Di modulo

In aggiunta, le variabili dichiarate e `const` possono avere uno scope a livello di blocco (delimitato dalle {} ad esempio).

```
let x;
const y = 42;
console.log(x); //output: undefined
console.log(y); //output: 42
```

VARIABLES AND SCOPE: LET KEYWORD

```
let bool; //variable declaration  
bool = true;  
  
if(bool){  
    //console.log(msg); //msg not accessible here  
    let msg = "Hello"; //declaration + assignment  
    console.log(msg); //output: Hello  
}  
  
//console.log(msg); //msg not defined here  
console.log(bool); //output: true
```



Lo scope delle variabili usando let/const è il blocco più vicino/interno. Le variabili sono accessibili solo dopo la riga in cui sono dichiarate e definite.

JAVASCRIPT: HOISTING

L'hoisting (sollevamento) è il comportamento implicito di spostare dichiarazioni di variabili (e funzioni) all'inizio del loro scope.

Quando si usa `let/const`, solo la dichiarazione è hoisted all'inizio dello scope, non l'inizializzazione, quindi `console.log(x)` dà errore perché non è dichiarata `x`, se lo fosse stata avrebbe stampato `undefined`.

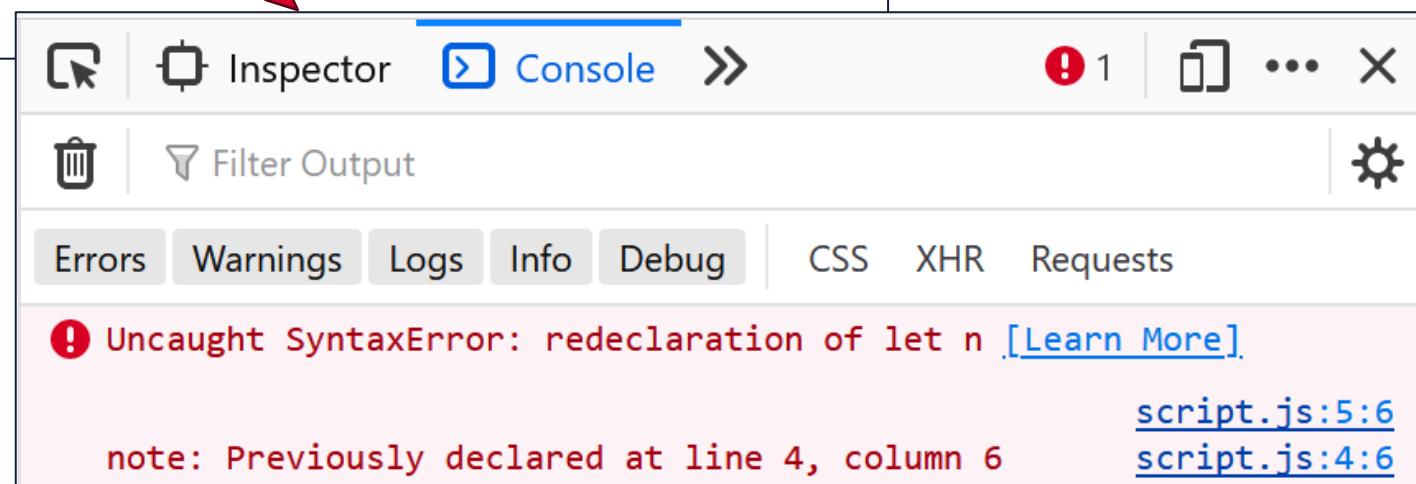
L'inizializzazione avviene sulla riga che inizialmente contiene la dichiarazione della variabile (quindi devono avvenire sullo stesso blocco).

```
// x variable not defined
{
  // Declaration for x is hoisted here
  // TDZ for the x variable
  // TDZ for the x variable
  console.log(x); //raises error
  // TDZ for the x variable
  let x = 1; // TDZ ends, x initialized
  // x variable initialized to 1
  // x variable initialized to 1
}
// x variable not defined
```

! ReferenceError: can't access lexical declaration 'x' before initialization

VARIABLES AND SCOPE: LET KEYWORD

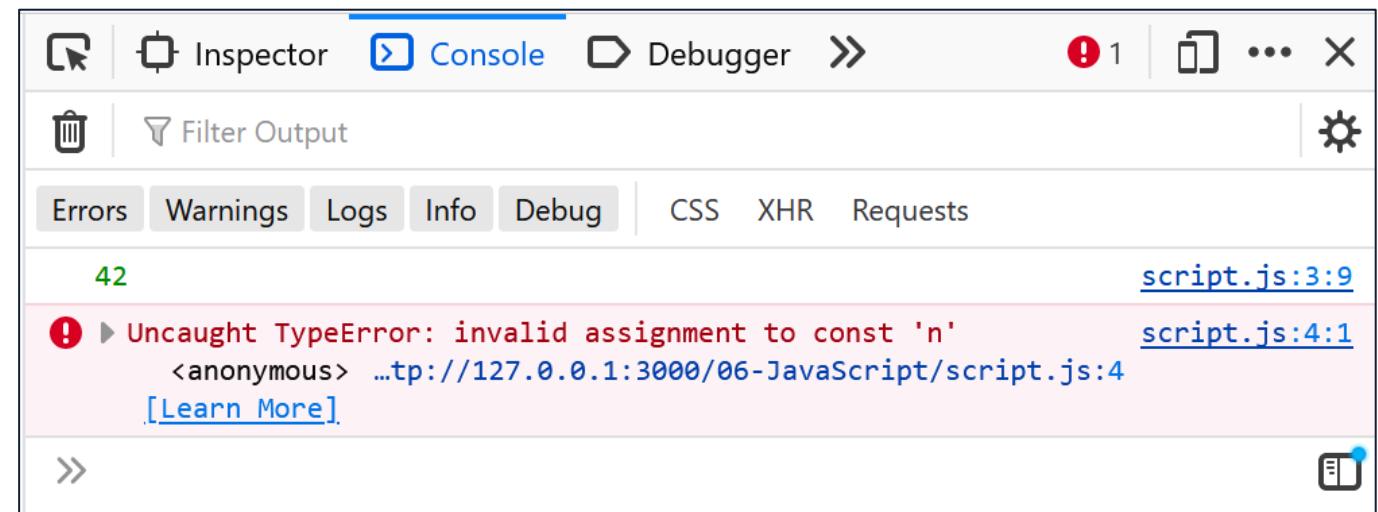
```
let n; //variable declaration  
n=1;  
if(n>0){  
    let n=2; //shadows n variable from external scope  
    //let n; //error: cannot re-declare in the same block  
    console.log(n); //output: 2  
}  
console.log(n); //output: 1
```



VARIABLES AND SCOPE: CONST KEYWORD

Const può essere usato per dichiarare costanti nello scope del blocco locale. Il comportamento di hoisting è simile a let.

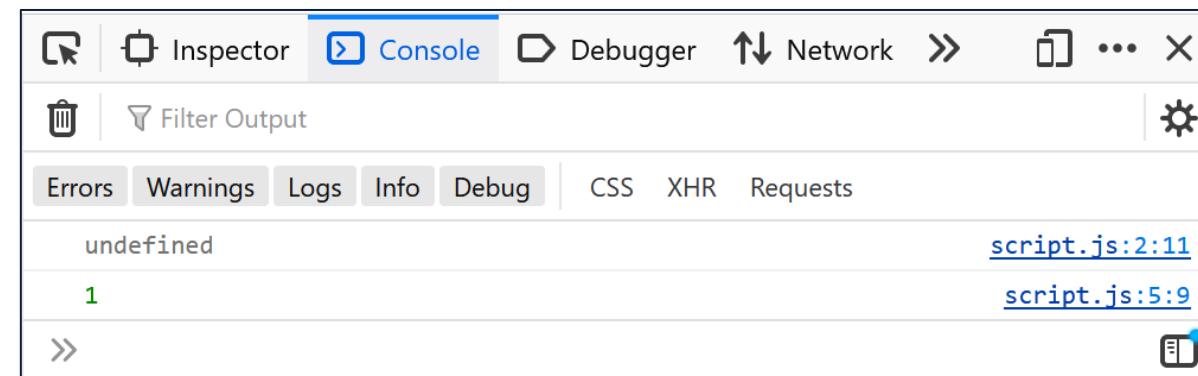
```
const n=42;  
console.log(n); //output: 42  
n=43; //raises a TypeError
```



VARIABLES: BEFORE ECMASCIRIPT 6

La dichiarazioni di variabili prima di ES5 funzionava con var, oppure veniva fatto implicitamente, questo poteva portare a proprietà con errori. Nell'esempio della slide console.log(n) stampa undefined perché n è dichiarata implicitamente, poi console.log(n) stampa 1 perché n, dopo la dichiarazione, assume uno scope globale e non più a livello di blocco.

```
if(true){  
    console.log(n);  
    var n=1;  
}  
console.log(n);
```



VARIABLES AND SCOPE: VAR HOISTING

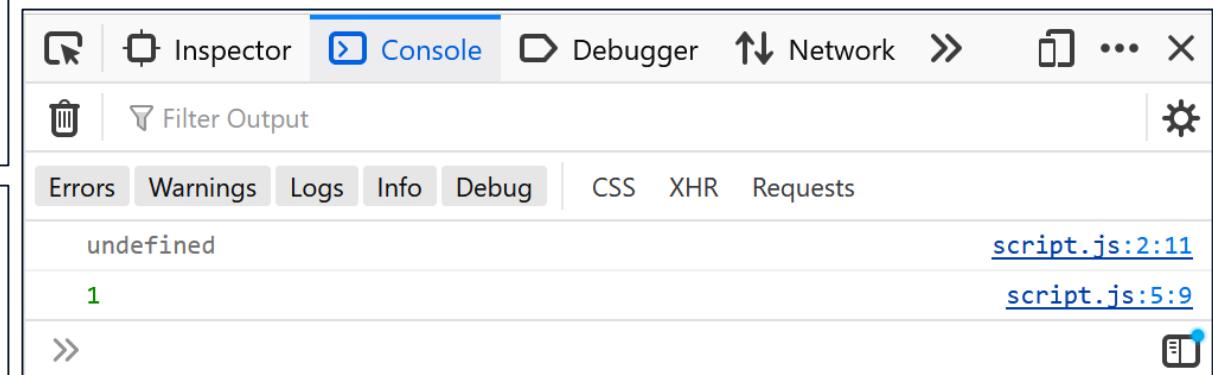
Le variabili dichiarate usando var sono hoisted alla funzione più vicina o allo scope globale (non a livello di blocco) e sono anche inizializzate come undefined.

In slide 14 c'è l'esempio che mostra come let n=2 dà errore di referenziazione in quanto si sta dichiarando di nuovo n nello stesso blocco.

```
if(true){  
    console.log(n); //output: undefined  
    var n=1;  
}  
  
console.log(n); //output: 1
```



```
var n;  
if(true){  
    console.log(n); //output: undefined  
    n=1;  
}  
  
console.log(n); //output: 1
```



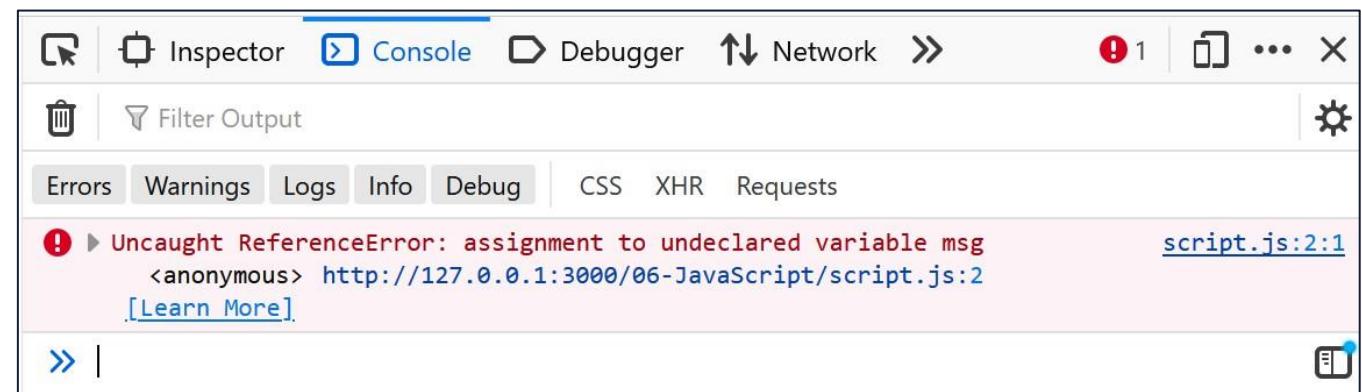
VARIABLES AND SCOPE: IMPLICIT DECL.

Le variabili possono essere dichiarate implicitamente, ad esempio per semplicità li useremo in un'assegnazione senza alcuna keyword.

Farlo risulta nella creazione di una variabile globale.

È una pessima pratica, che porta a molti errori, e non si dovrebbe mai fare. È anche proibito nella JS “strict mode” e risulta in un ReferenceError.

```
"use strict"  
  
msg = "pls don't do this"
```



PRIMITIVE DATA TYPES

```
let x; // variable declared and initialized to undefined
console.log(typeof x); // undefined

x=42;
console.log(typeof x); // number
x=3.14;
console.log(typeof x); // number
x="hello";
console.log(typeof x); // string
x='hello';
console.log(typeof x); // string
x=false;
console.log(typeof x); // boolean
x=10e12;
console.log(typeof x); // number

x=42/0;           // Infinity
console.log(typeof x); // number
x=42 * "Hello"; // NaN
console.log(typeof x); // number
```

BASIC OPERATORS

You should already be familiar with most JavaScript operators

Operator	Description
=	Assignment
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (since ECMAScript 2016)
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

COMPARISON OPERATORS

Comparisons operators are the same as Java, with the addition of `==`

Operator	Description
<code>==</code>	equal to
<code>==</code>	equal value and equal type
<code>!=</code>	not equal
<code>!==</code>	not equal value or not equal type
<code>></code>	greater than
<code><</code>	less than
<code>>=</code>	greater than or equal to
<code><=</code>	less than or equal to
<code>?</code>	ternary operator

LOOSE EQUALITY OPERATOR

L'operatore di ugualanza non stretta (==) controlla dove i suoi operatori sono uguali. L'operatore cerca di convertire e confrontare gli operandi di tipo diverso.

```
console.log(42 == 42)      //true
console.log("JS" == "JS")  //true
console.log("1" == 1)       //true
console.log(0 == false)    //true
console.log(true == 1)     //true
console.log(true == "1")   //true
console.log(true == 42)    //false
```

STRICT EQUALITY OPERATOR

L'uguaglianza ristretta (==) controlla dove i suoi operandi sono uguali senza conversioni di tipo, se gli operatori sono di tipo diverso, restituisce direttamente false.

```
console.log(42 === 42)      //true
console.log("JS" === "JS") //true
console.log("1" === 1)      //false
console.log(0 === false)    //false
console.log(true === 1)     //false
console.log(true === "1")   //false
console.log(true === 42)    //false
```

CONTROL FLOW

If,if-else,for,while,do,switch hanno la stessa sintassi e semantica in Java. Continue e break statement funzionano proprio come in Java.

```
if(condition){  
    //code  
}  
  
if(condition){  
    //code  
} else {  
    //code  
}  
  
do {  
    //code  
} while(condition);
```

```
for(let i=0;i<10;i++){  
    //code  
}  
  
while(expression){ /* code */ }  
  
switch(expression){  
    case value:  
        //code  
        break;  
    default:  
        //code  
}
```

FUNCTIONS



FUNCTIONS

Functions can be declared using the following syntax:

```
function greet(name){  
    console.log(`Hello ${name}`); //backticks for template literals  
}  
  
greet("Web Technologies"); //prints "Hello Web Technologies"
```

Lo scope di una funzione dichiarate è lo scope corrente in cui avviene la dichiarazione.

Quando nessun argomento è passato alla chiamata di una funzione i parametri vengono inizializzati ad `undefined`. Le funzioni possono avere valori di default come parametri.

Note: backticks («`») can be inserted using ALT+096 on the numpad (on Windows), if you have an italian keyboard

FUNCTIONS: DEFAULT ARGUMENTS

```
function greet(name, message="Hello"){ //message has a default value
    console.log(` ${message} ${name}`);
}

greet("Web Technologies");           // Hello Web Technologies
greet("Web Technologies", "Ciao");   // Ciao Web Technologies
greet();                            // Hello undefined
```

FUNCTIONS: HOISTING

L'hoisting si applica anche alle dichiarazioni delle funzioni.

```
greet("Web Technologies"); //prints "Hello Web Technologies" (!)

function greet(name, message="Hello"){
  console.log(` ${message} ${name}`);
}
```

FUNCTIONS: INNER SCOPE AND VISIBILITY

Una funzione crea il proprio scope: le variabili (e funzioni) dichiarate all'interno non sono visibili in scope esterni.

Una funzione può accedere alle variabili di scope esterni (chiusure): provveduti in quanto non sono mascherati nel proprio scope.

```
let message = "Hello";
console.log(greet("Web Technologies")); //output: Hello Web Technologies!

function greet(name){          // hoisted to the beginning of the global scope
    return generateMessage();
    function generateMessage(){ // hoisted to the beginning of greet's scope
        return `${message} ${name}!`;
    }
}
```

FUNCTIONS: INNER SCOPE AND VISIBILITY

Nella slide precedente la funzione generateMessage è locale allo scope della funzione greet. Non può essere riferita al di fuori di quello scope.

```
let message = "Hello";
console.log(greet("Web Technologies")); //output: Hello Web Technologies!

function greet(name){
    return generateMessage();
    function generateMessage(){ // hoisted to the beginning of greet's scope
        return `${message} ${name}!`;
    }
}

generateMessage(); // Raises ReferenceError: generateMessage is not defined
```

FUNCTION EXPRESSIONS

Le funzioni possono anche essere create usando le espressioni function e assegnarle a una variabile (come per ML, infatti val f = fn... è uguale a fun f = ...), si applicano le stesse regole dell'hoisting delle variabili in questi casi.

```
// standard function expression
let greet = function(name) {
  console.log(`Hello ${name}!`);
}

greet("Web Technologies"); // output: Hello Web Technologies!
```

```
// alternative, using arrow functions
let greet = (name) => {
  console.log(`Hello ${name}!`);
}
```

FUNCTION EXPRESSIONS

Hoisting examples:

Nel caso di greet la funzione viene definita per bene, con salute c'è un ReferenceError e viene definita in modo anonimo.

```
greet(); //ReferenceError: undefined
{
  greet(); //output: Hello

  function greet(){
    console.log("Hello");
  }
}
```

```
salute(); //ReferenceError: undefined
{
  salute(); //ReferenceError: uninitialized

  let salute = function(){
    console.log("Howdy");
  }
}
```

NESTED FUNCTIONS

Una funzione è nested quando è creata all'interno di un'altra funzione. Le funzioni annidate possono essere ritornate e usate al di fuori della funzione originale. Non importa dove sono usate, le funzioni annidate possono essere accedute in contesti esterni della funzione che le ha create.

Nell'esempio `getGreeter` richiamato in `helloGreeter` mi sta accettando solo il saluto da stampare, poi usando `helloGreeter` come funzione a sè (dopotutto `helloGreeter` è il valore restituito da `getGreeter`, essendo di "tipo" una funzione può accettare un ulteriore parametro), quindi stamperà Hello, Web o Howdy, JS, in quanto in `helloGreeter` c'è la `console.log`, non in `getGreeter`.

```
function getGreeter(message) {  
  let sep = ",";  
  return function(name) {  
    console.log(` ${message}${sep} ${name}!`);  
  }  
}  
  
let helloGreeter = getGreeter("Hello");  
helloGreeter("Web"); //Hello, Web!  
  
let howdyGreeter = getGreeter("Howdy");  
howdyGreeter("JS"); //Howdy, JS!
```

OBJECTS



OBJECTS

Gli oggetti sono contenitori di dati del tipo key:valore.

```
let a = new Object(); // "object constructor" syntax  
let b = {};          // "object literal" syntax, used more often
```

- Si possono aggiungere proprietà a un oggetto quando viene creato.

```
let pet = {  
  name: "Hannibal",    // by key "name" store value "Hannibal"  
  age: 7              // by key "age"   store value 7  
}
```

Una proprietà ha una key (il nome o l'identificatore) prima dei : e un valore alla sua destra. Le dichiarazioni di proprietà sono separate da una virgola.

OBJECTS: ACCESSING PROPERTIES

Si accede alle proprietà tramite la notazione a punto (dot).

```
let pet = {  
    name: "Hannibal",  
    age: 7  
}  
  
console.log(pet.name);      // output: Hannibal  
console.log(pet.age);      // output: 7  
console.log(pet.nickname); // output: undefined
```

OBJECTS: ADDING/DELETING PROPERTIES

Le proprietà possono essere anche aggiunte usando la notazione dot.

```
let pet = {  
    name: "Hannibal",  
    age: 7  
}  
  
pet.nickname = "the Affable"; // new property  
delete pet.age; // delete property  
  
console.log(pet.name); // "Hannibal"  
console.log(pet.age); // undefined  
console.log(pet.nickname); // "the Affable"  
  
console.log(pet); // print object in console  
console.table(pet); // alternative way to print objects in console
```

▶ Object { name: "Hannibal", nickname: "the Affable" }
script.js:21:9

console.table()
script.js:22:9

(index)	Values
name	Hannibal
nickname	the Affable

»



OBJECTS: SQUARE BRACKETS NOTATION

Le chiavi delle proprietà possono anche contenere spazi ed essere accesse usando la notazione con le parentesi quadre [], con questa notazione si possono anche usare espressioni per concatenare stringhe od operatori per modificare variabili.

```
let dog = {  
    name: "Sif",  
    "is a good boy": true  
}  
  
console.log(dog["name"]);  
console.log(dog["is a good boy"]);  
//expressions can be used as keys  
dog["is the "+"best boy"] = true;  
  
console.table(dog);
```

console.table()
script.js:33:9

(index)	Values
name	Sif
is a good boy	true
is the best boy	true

» |



OBJECTS: REFERENCES

Le variabili assegnate a un oggetto contengono un riferimento all'oggetto, non l'oggetto stesso. Con secondPet è come se facessi `*secondPet = &firstPet`.

```
let firstPet = {  
    name: "Richard",  
    species: "Lizard"  
}  
  
let secondPet = firstPet;  
secondPet.name = "Chuck";  
secondPet.species = "Duck";  
  
console.log(firstPet); // Object { name: "Chuck", species: "Duck" }  
console.log(secondPet); // Object { name: "Chuck", species: "Duck" }  
console.log(firstPet == secondPet); // true, same value  
console.log(firstPet === secondPet); // true (both are references, same value)
```

OBJECTS: CLONING

Per clonare un oggetto si deve iterare su ogni proprietà e copiarla una a una.

```
function clone(object){  
  let copy = {};  
  // new object  
  for(let key in object){  
    copy[key] = object[key];  
  }  
  return copy;  
}
```

```
let pet = {  
  name: "Richard", species: "Lizard"  
}  
  
let copy = clone(pet);  
  
copy.name = "Chuck";  
console.log(pet.name); // Richard  
console.log(copy.name); // Chuck
```

OBJECTS: SHALLOW AND DEEP CLONING

I valori della proprietà di un oggetto possono essere altri oggetti, la funzione clone precedentemente implementata fa una copia leggera (shallow), gli oggetti interni non sono clonati per intero, sono copiati solo i riferimenti.

```
function clone(object){  
  let copy = {};  
  for(let key in object){  
    copy[key] = object[key];  
  }  
  return copy;  
}
```

```
let pet = {  
  name: "Garfield", species: "Cat",  
  owner: {  
    name: "John", age: 17  
  }  
}  
  
let copy = clone(pet);  
copy.owner.name = "Liz";  
console.log(pet.owner.name); // Liz  
console.log(copy.owner.name); // Liz
```

OBJECTS: SHALLOW AND DEEP CLONING

Per ottenere un clone più profondo (deep) devo implementare i metodi clone da capo come scopo didattico, ma non serve implementarli sempre da 0, ci sono delle alternative built-in: `Object.assign()` per una copia shallow, `structuredClone()` per una copia deep.

```
// deep clone
function clone(object){
  let copy = {};
  for(let key in object){
    if(typeof object[key] === "object"){
      copy[key] = clone(object[key]);
    }
    else {
      copy[key] = object[key];
    }
  }
  return copy;
}
```

OBJECTS: METHODS

Le proprietà di oggetti possono essere anche delle funzioni, chiamate metodi. Ogni modo di definire metodi è equivalente.

```
let p = {  
  name: "John",  
  age: 17,  
  greet: function(){  
    console.log("Hi!");  
  }  
}  
  
p.greet(); // Hi!
```

```
let p = {  
  name: "John",  
  greet: greet  
}  
  
function greet(){  
  console.log("Hi!");  
}
```

```
let p = {  
  name: "John"  
}  
  
p.greet = function(){  
  console.log("Hi!");  
}
```

OBJECTS: METHOD SHORTHAND

Esiste anche una sintassi breve per dichiarare i metodi in un oggetto letterale.

```
// classic version                                // shorthand version
let p = {                                         let p = {
    name: "John",                                 name: "John",
    greet: function(){                           greet(){
        console.log("Hi!");                      console.log("Hi!");
    }                                           }
}                                             }
```

p.greet(); // Hi!

THE "THIS" KEYWORD

È comune per i metodi degli oggetti far riferimento ad altre proprietà dello stesso oggetto. I metodi possono essere acceduti dall'oggetto che li contiene tramite la keyword `this`.

```
let john = {
    name: "John",
    greet(){
        console.log(`Hi, I'm ${this.name}!`);
    }
}

john.greet(); // Hi, I'm John!
```

THE "THIS" KEYWORD

Il valore di this è valutato a run-time e dipende dal contesto.

```
let john = {name: "John"};
let will = {name: "Will"};

let greet = function(){
  console.log(`Hi, I'm ${this.name}`);
}

// same function is assigned as a method to two objects
john.greet = greet;
will.greet = greet;

john.greet(); // Hi, I'm John!
will.greet(); // Hi, I'm Will!
```

OBJECT METHODS: ARROW FUNCTIONS

Le funzioni definite con una freccia non hanno il this e se il this è riferziato, allora è chiesto dal contesto esterno.

```
let john = {nick: "John"};
let will = {nick: "Will"};

let greet = () => {
  console.log(`Hi, I'm ${this.nick}`);
}
john.greet = greet;
will.greet = greet;

john.greet(); // Hi, I'm undefined!
will.greet(); // Hi, I'm undefined!
```

OBJECTS: CONSTRUCTORS

Finora gli oggetti sono stati definiti usando la sintassi in modo letterale. Per scrivere molti oggetti simili si possono usare delle funzioni costruttore con la keyword new.

I costruttori sono funzioni regolari e si applicano 2 convenzioni:

1. Il loro nome deve iniziare con una lettera in maiuscolo
2. Devono essere invocati usando la keyword new.

OBJECTS: CONSTRUCTORS

```
function Pet(name, species){  
    this.name      = name;  
    this.species   = species;  
    this.age       = undefined;  
}  
  
let rick  = new Pet("Richard", "Lizard");  
let chuck = new Pet("Chuck", "Duck");
```

Quando una funzione è eseguita con new:

1. Viene creato un nuovo oggetto assegnato come this all'interno del costruttore
2. Viene eseguito il corpo della funzione. Tipicamente modifica this.
3. Il valore di this viene restituito.

OBJECTS: OPTIONAL CHAINING

```
let pet = {  
    name: "Garfield"  
}  
console.log(pet.species); // undefined
```

Se si accede a una proprietà undefined si ottiene un valore undefined. Alcune volte si può provare ad accedere a una proprietà undefined, che risulta in un errore.

```
console.log(pet.owner.name); //throws ReferenceError
```

OBJECTS: OPTIONAL CHAINING

In molti casi pratici, si può preferire ottenere un valore `undefined` invece di un errore. Si possono controllare direttamente le proprietà definite in precedenza accedendo alle sue proprietà interne, ma è molto ripetitivo e poco elegante.

L'operatore di concatenazione opzionale `?` entra in gioco in queste situazioni, ferma direttamente la valutazione della parte sinistra se è indefinita e restituisce `undefined`.

```
let ownerName = pet.owner ? pet.owner.name : undefined;
```

```
let ownerName = pet.owner?.name;
```

OPTIONAL CHAINING: VARIANTS

Il chaining opzionale può essere usato per chiamare una funzione che può non esistere, o quando si accede ad alcune proprietà della notazione a parentesi quadre.

```
let john = {  
    name: "John",  
    greet(){ return "Hi!"; },  
    address: { "street name": "Web Dev Blvd" }  
}  
  
let mike = { name: "Mike", age: 17 }  
  
console.log( john.greet?.() ); // Hi!  
console.log( mike.greet?.() ); // undefined  
console.log( john.address?.['street name']); // Web Dev Blvd  
console.log( mike.address?.['street name']); // undefined
```

OBJECTS: CONFIGURING PROPERTIES

Le proprietà degli oggetti, oltre ad avere un valore, hanno altri 3 attributi speciali (chiamati flags):

1. Writeable: se è vero, può essere modificato, altrimenti è solo di lettura
2. Enumerable: se è vero, la proprietà è listata nei loop.
3. Configurable: se è vero può essere rimosso e la sua flag può essere modificata.

Quando creiamo una proprietà nel modo “tradizionale” tutte le flag sono true.

OBJECTS: CONFIGURING PROPERTIES

```
let pet = { species: "cat" }

Object.defineProperty(pet, "name", {
  value: "Garfield",
  writable: false,
  enumerable: false,
  configurable: false
})

for(let key in pet){
  console.log(`Key: ${key}`); //only prints Key: species
}
pet.name = "Odie"; //TypeError: "name" is read-only
delete pet.species; //works
delete pet.name; //TypeError: property "name" is non-configurable
```

- Properties can be defined also using [Object.defineProperty\(\)](#)
- When props are created in this way, all flags default to **false**

OBJECTS: PROPERTY GETTERS AND SETTERS

Ci sono 2 tipi di proprietà:

- Data proprietà: immagazzinano un valore
- Accessor proprietà: funzioni che sono eseguite quando una proprietà è acceduta

Le proprietà accessor sono rappresentate da un getter (invocato quando la proprietà viene letta) e da un setter (invocato quando viene assegnato). Le keyword get e set possono essere usate per denotare getter e setter negli oggetti letterali.

OBJECTS: PROPERTY GETTERS AND SETTERS

```
let p = {
    first: "John",
    last: "Smith",
    get fullName(){
        return `${this.first} ${this.last}`;
    },
    set fullName(name) {
        [this.first, this.last] = name.split(" "); // fancy destructuring assignment
    }
}
// notice that we access fullName as a property and not as a function (no "()")!
// from the outside, there is no difference between data and accessor properties
console.log(p.fullName); //John Smith
p.fullName = "Jane White";
console.log(p.fullName); //Jane White
p.fullName(); //TypeError: p.fullName is not a function
```

REFERENCES

- **The Modern JavaScript Tutorial**

Freely available at <https://javascript.info/> or on [GitHub](#)

Part 1: An Introduction, JavaScript Fundamentals, Code Quality (3.1 to 3.4), Objects: the basics (4.1 to 4.6), Data types (5.1 to 5.3), Advanced working with functions (6.3).

- **Eloquent JavaScript (3rd edition)**

By Marijn Haverbeke

Freely available at <https://eloquentjavascript.net/>

Chapters: Introduction, 1 to 6.

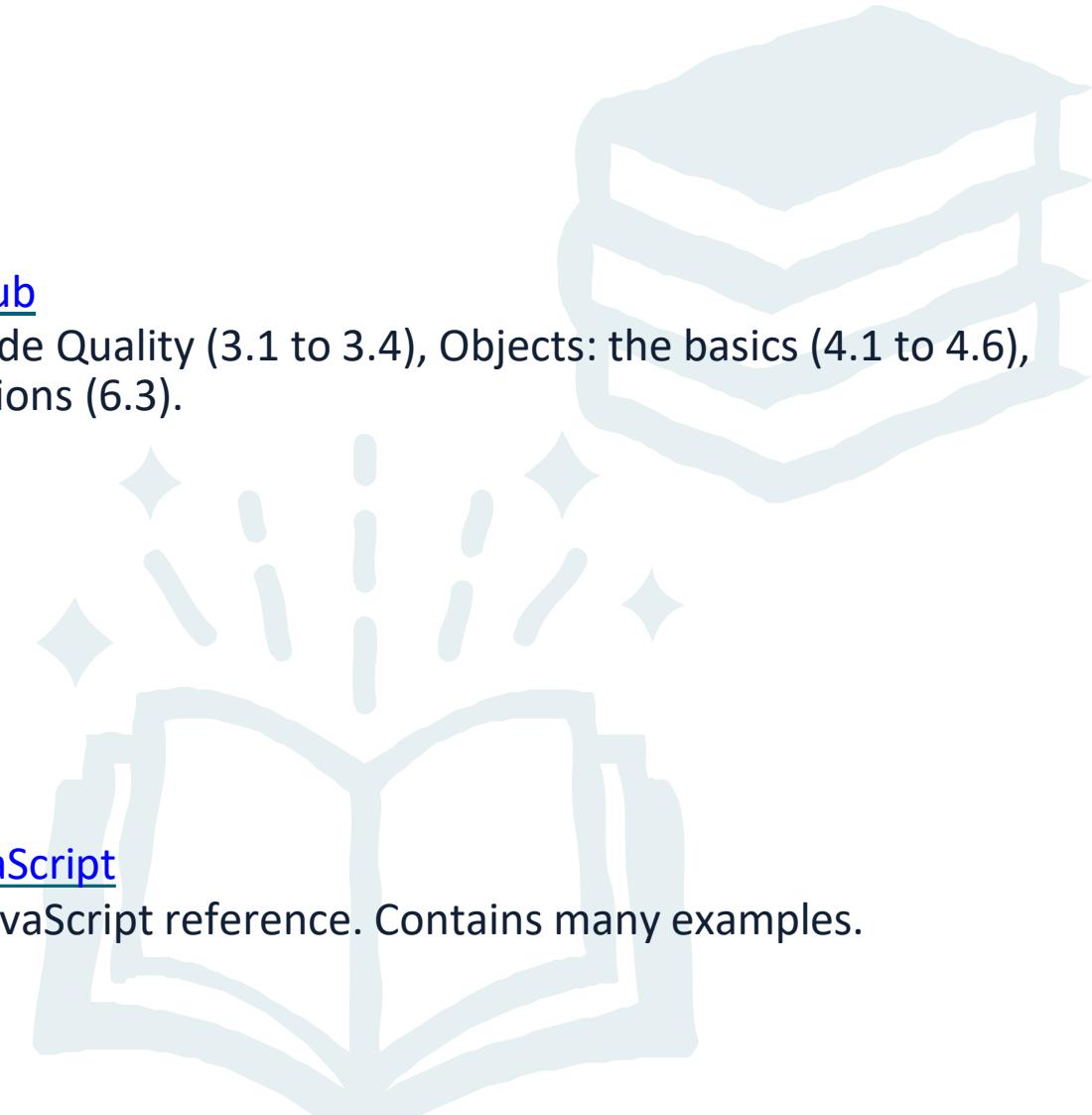
- **Learn JavaScript**

MDN web docs course

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript>



You can check out this page if you need a quick JavaScript reference. Contains many examples.



UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
WEB TECHNOLOGIES – LECTURE 06

JAVASCRIPT: PART II

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



PREVIOUSLY, ON WEB TECHNOLOGIES

So far we've learned the key concepts of the JavaScript language:

- **Variables, functions, scope (and hoisting)**
- **Objects, constructors**

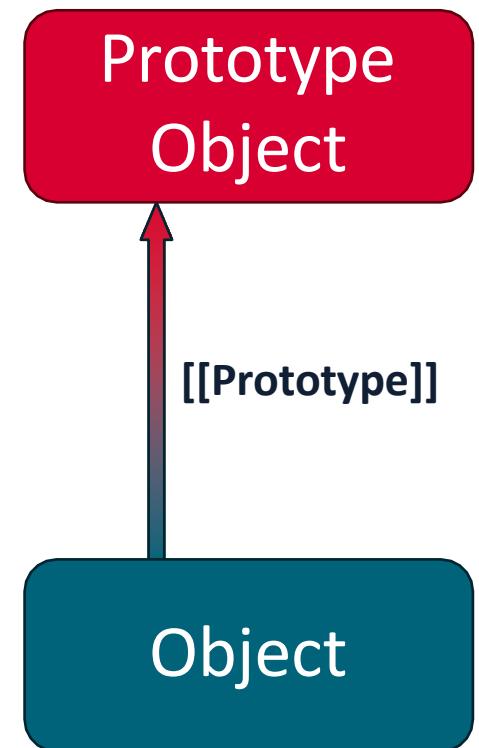
Today we'll continue dwelling into the JavaScript language and learn some other core concepts:

- **Prototypes and Inheritance**
- **Data Structures (Arrays, Maps, Sets)**
- **Classes**
- **Modules**

PROTOTYPES AND INHERITANCE

OBJECTS: PROTOTYPES AND INHERITANCE

- L'ereditarietà è un aspetto essenziale della programmazione orientata agli oggetti
- Funzionalità di JavaScript Ereditarietà prototipale
- Ogni oggetto ha una speciale proprietà nascosta chiamato `[[Prototype]]`
- `[[Prototype]]` è nullo, o fa riferimento ad un altro oggetto
- Quando accediamo a una proprietà di un oggetto e questa manca, JavaScript cerca la proprietà attraversando la catena di prototipi



OBJECTS: PROTOTYPES

```
let pet = {  
    legs: 4, greet(){ console.log("..."); }  
}  
let cat = {  
    __proto__: pet //setting the prototype  
}  
cat.greet(); //..."  
console.log(cat.legs); //4  
console.log(cat.__proto__); //Object { legs: 4, greet: greet() }
```

- `__proto__` è un getter/setter per l'effettiva proprietà `[[Prototype]]`
- Nel moderno JavaScript, è possibile utilizzare
`Object.getPrototypeOf()` e `Object.setPrototypeOf()`

OBJECTS: WORKING WITH PROTOTYPES

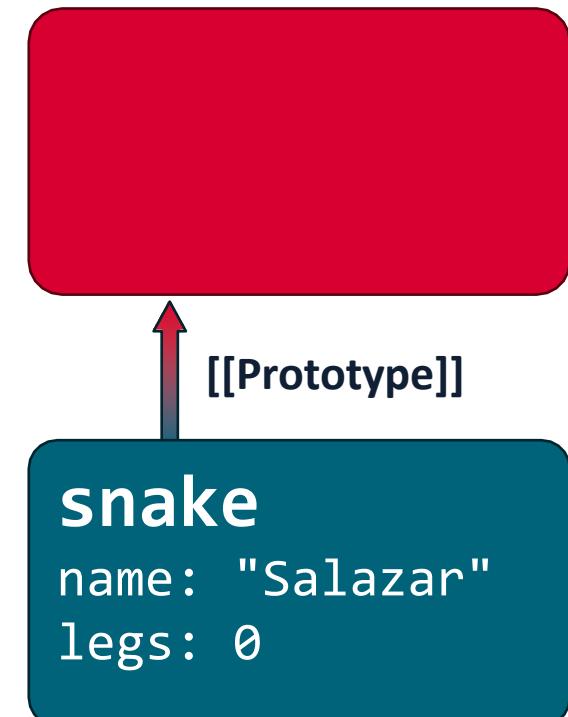
- I prototipi vengono utilizzati solo durante la lettura delle proprietà
- Le operazioni di scrittura/eliminazione funzionano direttamente sull'oggetto

```
let pet    = { legs: 4 }

let cat    = { name: "Garfield" }
let snake = { name: "Salazar" }

Object.setPrototypeOf(cat, pet);
Object.setPrototypeOf(snake, pet);

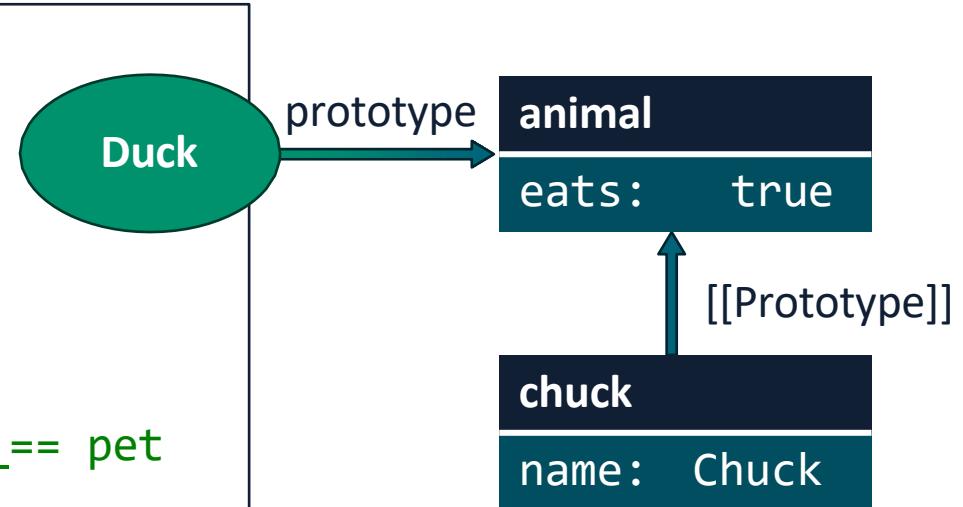
snake.legs = 0
console.log(`Cat legs: ${cat.legs}`);      // Cat legs: 4
console.log(`Snake legs: ${snake.legs}`); // Snake legs: 0
console.log(`Pet legs: ${pet.legs}`);      // Pet legs: 4
```



CONSTRUCTORS AND PROTOTYPES

- Possiamo creare oggetti utilizzando le funzioni del Costruttore, come «new Pet()»
- Quando Pet.prototype è un oggetto, l'operatore new lo utilizza per impostare il [[Prototype]] per l'oggetto appena creato

```
let pet = {  
  eats: true  
};  
function Duck(name) {  
  this.name = name;  
}  
Duck.prototype = pet;  
let chuck = new Duck("Chuck"); //chuck.__proto__ == pet  
console.log(chuck.eats); //true
```



CONSTRUCTOR AND PROPERTIES

Cosa succede quando il costruttore non ha un prototipo proprietà?

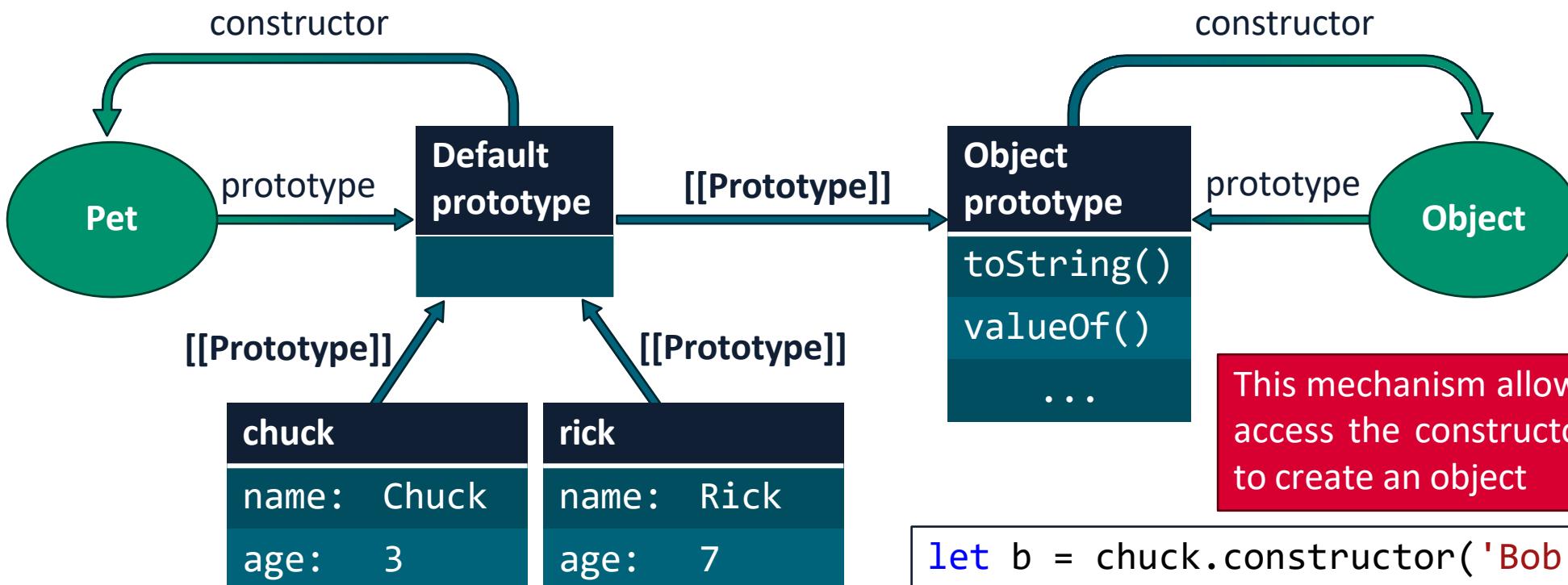
```
function Pet(name, age) {  
    this.name = name; this.age = age;  
}  
let chuck = new Pet('Chuck', 3); let rick = new Pet('Rick', 7);
```

- In tal caso, viene utilizzato un prototipo predefinito
- Il prototipo predefinito è un oggetto con una proprietà del costruttore, che punta alla funzione del costruttore

```
Pet.prototype = { constructor: Pet }
```

CONSTRUCTORS AND PROTOTYPES

```
function Pet(name, age) {  
    this.name = name; this.age = age;  
}  
let chuck = new Pet('Chuck', 3); let rick = new Pet('Rick', 7);
```



CONSTRUCTORS AND PROTOTYPES

- Due modi alternativi per aggiungere metodi agli oggetti creati:

```
function describe(){ console.log(`I'm ${this.name} and my age is ${this.age}`); }

function Pet(name, age) {
  this.name = name; this.age = age;
  this.describe = describe; // (1) adds a describe method to each created object
}

let chuck = new Pet('Chuck', 3); let rick = new Pet('Rick', 7);

Pet.prototype.describe = describe; // (2) adds a describe method to the prototype
                                  // of each created object

chuck.describe(); rick.describe();
```

DATA STRUCTURES

ARRAYS

- Gli array consentono di memorizzare sequenze ordinate di valori
- Può essere dichiarato utilizzando la sintassi letterale dell'array con parentesi quadre [] o il costruttore Array

```
//array literal syntax
let a = ["HTML", "CSS", "JS"];
console.log(a); //Array(3) [ "HTML", "CSS", "JS" ]
```

```
//constructor syntax
let b = new Array("HTML","CSS","JS");
console.log(b); //Array(3) [ "HTML", "CSS", "JS" ]
```

ARRAYS: INDEXING

- Le matrici sono indicizzate, a partire da 0.
- È possibile accedere a valori specifici in modalità di lettura/scrittura utilizzando la notazione delle parentesi quadre
- La proprietà length contiene l'indice massimo, più uno

```
let a = ["HTML", "CSS", "JS"];
a[2] = "JavaScript";
a[3] = "React";

console.log(a[0]);      // HTML
console.log(a[1]);      // CSS
console.log(a[2]);      // JavaScript
console.log(a[3]);      // React
console.log(a.length); // 4
```

ARRAYS: LENGTH

- Probabilmente avrete notato che abbiamo definito array.length in un modo strano...
- Questo perché la lunghezza non è in realtà il conteggio dei valori nell'array!

```
let a = [1, 2, 3, 4, 5];                                // an interesting thing about the
                                                       // length is that it is writeable
a[999] = 998;
console.log(a)
console.log(a.length); // 1000 (!)
console.log(a[5]);   // undefined
console.log(a[999]); // 998
console.log(a[2000]);// undefined
                                                               a.length = 3;
                                                               console.log(a); // [1,2,3]
                                                               a.length = 5;
                                                               console.log(a); // [1,2,3,<2 empty>]
                                                               a.length = 0; // clears the array
```

ARRAYS: MIXED TYPES

- Un array può contenere tipi di dati eterogenei

```
let a = [
  "JavaScript",
  {name: "John", job: "Dev"},
  12,
  function(name){
    console.log(`Hello ${name}`);
  }
]

console.log(a[1].name); // John
a[3]("Web Technologies"); // Hello Web Technologies
```

ARRAYS: METHODS

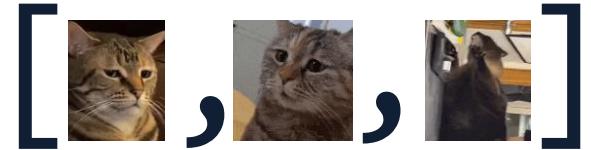
Gli array forniscono metodi dedicati per aggiungere/rimuovere elementi

- `push()`: aggiunge un elemento alla fine
- `shift()`: prende un elemento dall'inizio
- `pop()`: Prendi un elemento dalla fine
- `unshift()`: aggiunge un elemento all'inizio

```
let a = [1];      // [1]
a.push(2);      // [1, 2]
a.unshift(0);   // [0, 1, 2]
x = a.pop();    // [0, 1]
y = a.unshift(); // [1]
console.log(x); // 2
console.log(y); // 0
```

ARRAY METHODS: VISUALIZED

[, ] .push()



[, ] .unshift()



[, , ] .pop()



[, , ] .shift()



For a visualization of all array methods, you can check out: <https://js-arrays-visualized.com/>

ARRAYS: ITERATION

```
let a = ["a", "b", "c"];
a[4] = "e";

for(let i = 0; i < a.length; i++)
  console.log(a[i]); //a,b,c,undefined,e

for(let item of a)
  console.log(item); //a,b,c,undefined,e

a.forEach( (value, index, array) => {
  console.log(`a[${index}]=${value}`);
});

for(let key in a){
  console.log(a[key]); //a,b,c,e
}
```

- L'array può essere iterato utilizzando i cicli for sugli indici, utilizzando l'alternativa for.. della sintassi o il metodo forEach.
- Poiché gli array sono oggetti, è possibile utilizzarli anche for item in array, ma non è una buona idea
- È 10-100 volte più lento
- Potrebbero esserci altre proprietà enumerabili al di fuori dei valori dell'array...

MULTIDIMENSIONAL ARRAYS

- Array items can also be other arrays
- We can use this to define multidimensional arrays (e.g.: matrices)

```
let matrix = [
  [1,2,3],
  [4,5,6],
  [7,8,9],
]

console.log(matrix[0][0]); //1
console.log(matrix[1][1]); //5
console.log(matrix[2][2]); //9
```

DESTRUCTURING ASSIGNMENTS

A special syntax that allows to **unpack arrays** into variables

```
let [x, y] = ["a", "b", "c"]; // remaining array elements are discarded
console.log(x); //a
console.log(y); //b

let [a, b, ...rest] = [1,2,3,4,5]; // remaining array elements are stored in rest
console.log(a);    //1
console.log(b);    //2
console.log(rest); //[3, 4, 5]

//similar syntax can also be used in function parameters
function greet(msg, ...names){
  console.log(` ${msg} to ${names}`);
}
greet("Hello", "Ann", "Bob", "Carl"); //Hello to Ann,Bob,Carl
```

ITERABLES

- Il ciclo `for...of` può essere utilizzato con oggetti iterabili
- Gli array sono oggetti iterabili (così come le stringhe)

```
let string = "Web Technologies!";
for(let char of string){
  console.log(char); //W, e, b, , T, e, ...
}
```

ITERABLES

```
//range represents a set of integers:  
//starting at min, and arriving up to max, with steps of the specified size  
let range = {  
    min: 1,  
    max: 10,  
    step: 2  
}  
  
for(let value of range){ //TypeError: range is not iterable  
    console.log(value);  
}
```

- Per rendere iterabile un oggetto, è necessario implementare un apposito metodo `Symbol.iterator`

ITERABLES: SYMBOL.ITERATOR

- Quando il ciclo for...of, chiama una volta il metodo Symbol.iterator sull'oggetto (e genera un TypeError se il metodo non esiste)
- Avanti, il per.. Il ciclo of funziona solo con l'oggetto (Iterator) restituito dalla chiamata Symbol.iterator
- Quando il del for...of ciclo deve accedere al valore successivo, chiama il next() sull'iteratore
- Il risultato dell'invocazione di next() è un oggetto del form
- dove done=true significa che il ciclo è terminato, altrimenti value è il valore successivo.

IMPLEMENTING SYMBOL.ITERATOR

```
let range = { min: 1, max: 10, step: 3 }
range[Symbol.iterator] = function(){
    return {
        current: this.min, max: this.max, step: this.step, next(){
            let n = {done: false, value: this.current};
            if(n.value > this.max)
                n.done = true; //termina l'iterazione
            this.current += this.step; //se l'iterazione non è
                                //terminata aggiunge il numero
                                //di step al valore attuale
                                //(nella prima iterazione è 1,
                                //poi 4, poi 7 e infine 10)
            return n;
        }
    }
} for(let value of range)
console.log(value); //1,4,7,10
```

WORKING WITH ITERATORS EXPLICITLY

- È anche possibile lavorare direttamente con gli iteratori, come mostrato di seguito
- Il ciclo while sottostante replica il for...of loop della diapositiva precedente

```
let iterator = range[Symbol.iterator]();
while(true){
  let result = iterator.next();
  if(result.done)
    break;
  console.log(result.value);
}
```

MAPS

- Le mappe sono raccolte di coppie chiave-valore.
- Quindi, sono solo come oggetti? Non proprio, Maps consente chiavi di qualsiasi tipo!
- I metodi e le proprietà sono:

new Map()

Creates the map

map.set(key, value)

Stores the value by the key

map.get(key)

Returns the value by the key, or undefined if not present

map.has(key)

Returns true if the key exists, false otherwise

map.delete(key)

Removes the key-value pair by the key

map.clear()

Removes everything from the map

map.size

Is the current element count

MAPS VS OBJECTS

```
let map = new Map();
let o = {};

map.set(1, "Num");
map.set(true, "Bool");

console.log(map); //Map { 1 → "Num", "1" → "Str", true → "Bool", "true" → "Str" }
console.log(map.size); // 4

o[1]      = "Num";  o["1"]      = "String";
o[true]    = "Bool"; o["true"]   = "String";
console.log(o); //Object { 1: "String", true: "String" }
```

MAPS: ITERATIONS

```
let map = new Map();

map.set("Hello", "JS"); map.set(1, true); map.set(false, 42);

for(let key of map.keys()){ //map.keys() returns an iterable over keys
    console.log(key); //Hello, 1, false
}
for(let value of map.values()){ //map.values() returns an iterable over values
    console.log(value); //JS, true, 42
}
for(let [key, value] of map.entries()){ //iterable over entries
    console.log(` ${key}: ${value}`);
}
for(let [key, value] of map){ //equivalent to the one before
    console.log(` ${key}: ${value}`);
}
```

SETS

- I set sono una struttura di dati per memorizzare raccolte di valori senza ripetizioni
- I metodi principali di un set sono:

new Set([iterable])

set.add(value)

set.delete(value)

set.has(value)

set.clear()

set.size

Creates the Set. If an iterable is provided, copies its values

Stores the value, returns the set itself

Deletes value from Set. Returns true if value existed, false otherwise

Returns true if value exists in the set, false otherwise

Removes everything from the set

Is the current element count

SETS: EXAMPLES

```
let set = new Set();

let eric = { name: "Eric" };
let stan = { name: "Stan" };
let kyle = { name: "Kyle" };

set.add(eric); set.add(stan);
set.add(kyle); set.add(eric);
set.add(kyle);

// set keeps only unique values
console.log( set.size ); // 3

for (let user of set) {
  console.log(user.name); //Eric, Stan, Kyle
}
```

CLASSES



CLASSES IN JAVASCRIPT

Conosciamo il concetto di classi nel software orientato agli oggetti

Le classi sono fondamentalmente modelli per la creazione di oggetti

I costruttori e new possono aiutare in JavaScript

JavaScript presenta anche un costrutto di classe più avanzato, che offre alcuni vantaggi

```
class Pet {  
    constructor(name){ this.name = name }  
    method1(){/*...*/}  
    method2(){/*...*/}  
    /* more methods */  
}  
  
let pet = new Pet("Chuck");
```

CLASSES IN JAVASCRIPT

```
class Pet {  
    constructor(name){ this.name = name }  
    method1(){/*...*/}  
    method2(){/*...*/}  
    /* more methods */  
}  
  
let pet = new Pet("Chuck");
```

- `new Pet()` crea un nuovo oggetto e invoca il metodo costruttore con gli argomenti dati, che imposta la proprietà `name` sull'oggetto. Il nuovo oggetto viene quindi restituito.

CLASSES IN JAVASCRIPT

- Che cos'è esattamente una classe? Non si tratta di una nuova entità a livello di linguaggio
- `type(Pet);` funzione
- In JavaScript, le classi sono tipi speciali di funzioni
- Cosa fa davvero il costrutto di classe `Pet {...}` allora?
- Crea una funzione costruttore denominata `Pet`. Il codice della funzione è preso dal metodo `constructor()` (o è vuoto se non esiste tale metodo).
- Memorizza altri metodi di classe in `Pet.prototype`

CLASSES IN JAVASCRIPT

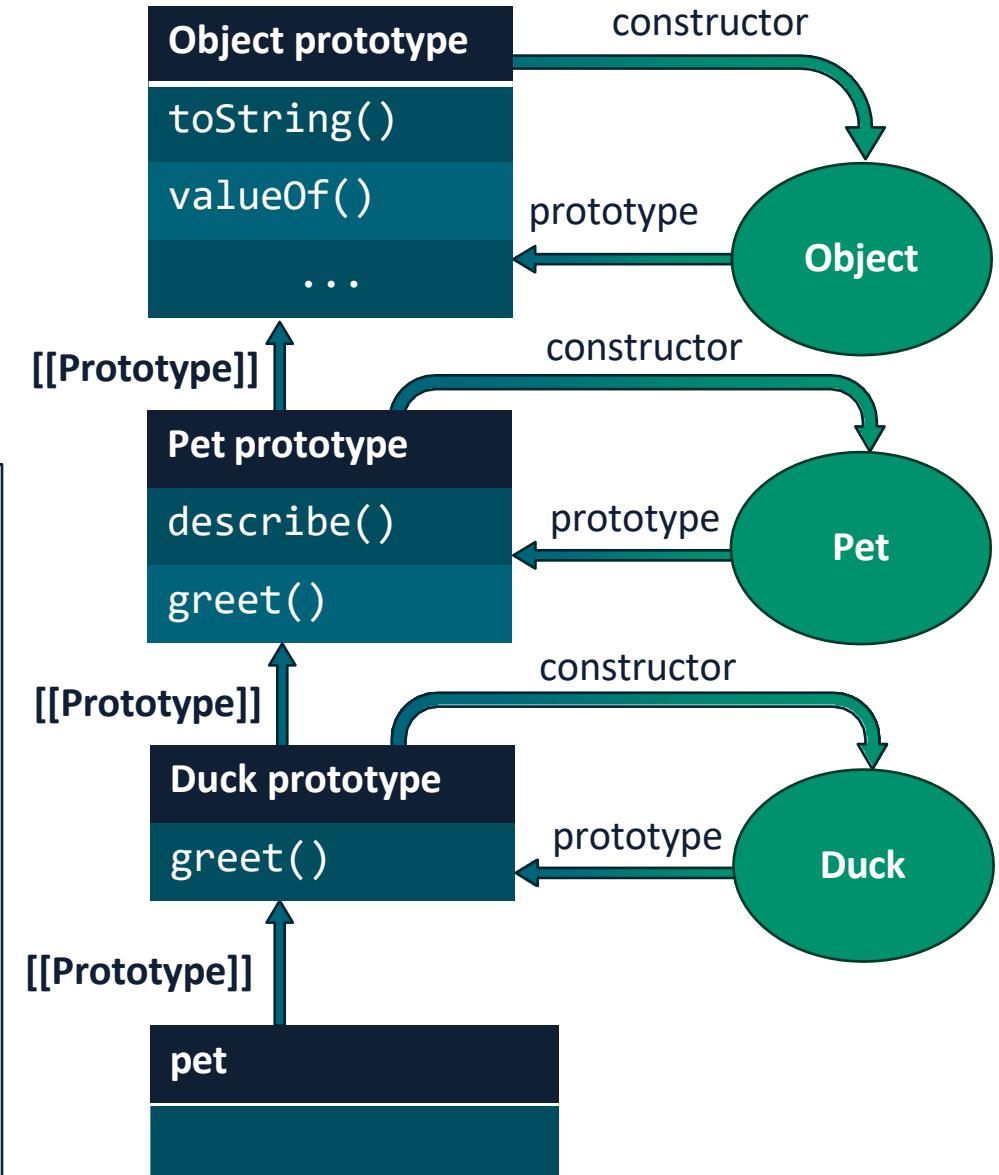
- Analogamente agli oggetti, le classi supportano le proprietà e i getter/setter

```
class Duck {  
    species = "Duck"; //property  
    constructor(name){this.name = name}  
    greet(){  
        console.log("Quack!")  
    }  
    get fullDescription(){return `${this.name} the ${this.species}`};  
  
    [Symbol.iterator](){ /*...*/}  
}  
  
let duck = new Duck("Chuck");  
duck.greet(); //Quack!  
console.log(duck.fullDescription); //Chuck the Duck
```

CLASS INHERITANCE

- Le classi JavaScript possono ereditare da altre classi utilizzando la parola chiave `extends`

```
class Pet {  
    describe(){ console.log("I'm a pet"); }  
    greet(){ console.log("..."); }  
}  
  
class Duck extends Pet {  
    greet(){ console.log("Quack!"); }  
}  
  
let chuck = new Duck();  
chuck.describe(); //I'm a pet  
chuck.greet();   //Quack!
```



CLASS INHERITANCE

- Internally, extends is implemented using the **[[Prototype]]** property

```
» chuck
← ▼ Object { }
  ▶ <prototype>: Object { ... }
    ▶ constructor: class Duck {} ↵
    ▶ greet: function greet() ↵
  ▶ <prototype>: Object { ... }
    ▶ constructor: class Pet {} ↵
    ▶ describe: function describe() ↵
    ▶ greet: function greet() ↵
    ▶ <prototype>: Object { ... }
```

```
»
```



ERROR HANDLING

ERRORS

- Durante l'esecuzione di uno script, possono verificarsi errori
- Possono accadere perché i programmati commettono errori, perché gli utenti hanno fornito input inaspettati e così via...
- Quando si verificano errori, gli script in genere si interrompono immediatamente, stampando un messaggio di errore nella console

```
let myvar = 0;

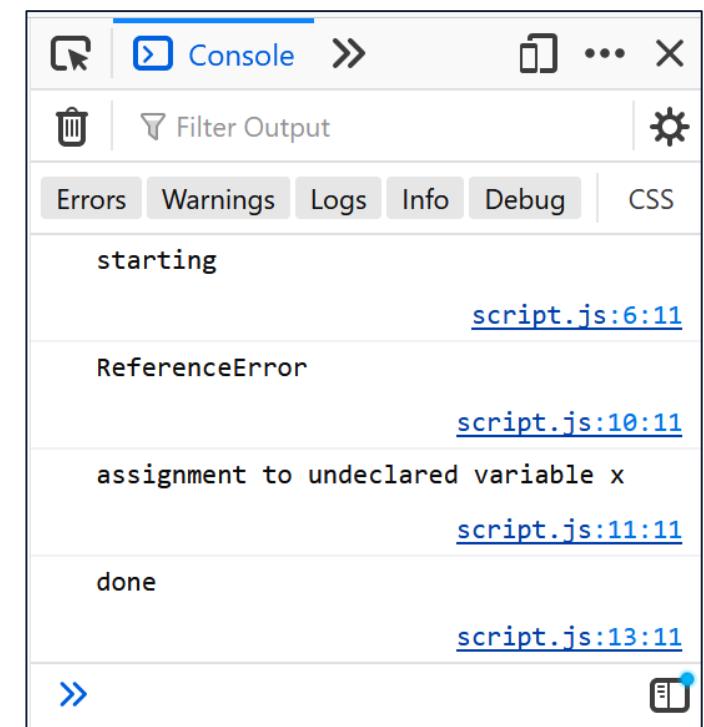
for(let num of [1,2,3,4,5]){
    mvvar += num; // ReferenceError: mvvar is not defined
}

console.log(myvar);
```

HANDLING ERRORS: TRY/CATCH/FINALLY

- Un costrutto di sintassi try/catch/finally fornisce modi per gestire gli errori.
- Concettualmente, è lo stesso costrutto che conosci da Java. Con qualche piccola differenza (le vedremo!)

```
try {  
    console.log("starting");  
    x = 1; //forgot the let keyword  
    console.log("assignment done"); // not executed  
} catch (error) {  
    console.log(error.name);  
    console.log(error.message);  
} finally {  
    console.log("done");  
}
```



HANDLING ERRORS: TRY/CATCH/FINALLY

Peculiarità dei costrutti try/catch in JavaScript:

Deve esserci al massimo un blocco catch (try/finally è ammissibile)

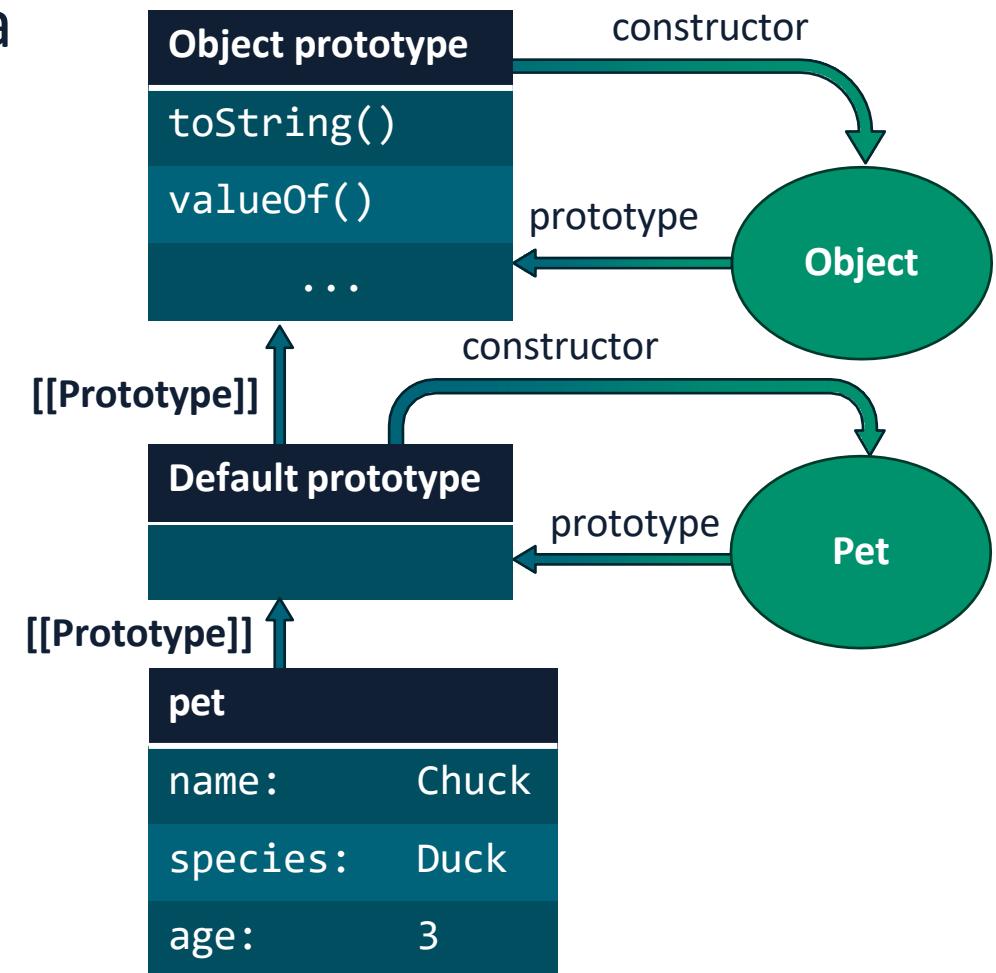
All'interno del blocco catch vengono gestiti diversi tipi di errori

```
try {
  x = 1; //forgot the let keyword
} catch (error) {
  if(error instanceof ReferenceError){
    console.log("Got a ReferenceError");
  } else {
    console.log(`Got something else: ${error.name}`);
  }
}
```

THE INSTANCEOF OPERATOR

- L'operatore instanceof verifica se la proprietà prototype di un determinato costruttore viene visualizzata in qualsiasi punto della catena di prototipi di un oggetto

```
function Pet(name, species, age) {  
    this.name = name;  
    this.species = species;  
    this.age = age;  
}  
const pet = new Pet('Chuck', 'Duck', 3);  
  
console.log(pet instanceof Pet); //true  
console.log(pet instanceof Object); //true
```



THROWING ERRORS

- Gli errori in JavaScript si propagano allo stesso modo di Java
- verso l'alto fino a quando non viene raggiunta la parte superiore dell'albero delle chiamate (in quel caso la valutazione dello script viene interrotta bruscamente e l'errore viene registrato nella console)
- Gli errori possono essere generati utilizzando la parola chiave `throw`

```
try {
    throw new Error("Oops!");
} catch (err) {
    console.log(err.name); // Error
    console.log(err.message); // Oops!
}

class MyError extends Error {
    constructor(message = "Whoops") {
        super(message);
        this.name = "MyError";
    }
}

try {
    throw new MyError();
} catch (err) {
    console.log(err.name); // MyError
    console.log(err.message); // Whoops
}
```

MODULES



MODULARITY

- JavaScript moderno consente la definizione di moduli a livello di linguaggio
- I moduli sono modi per dividere programmi complessi in più file (moduli), con ogni modulo contenente classi o funzioni per uno scopo specifico
- Migliora la manutenibilità, promuove la separazione dei problemi
- Può aiutare a disambiguare e prevenire conflitti di denominazione (migliora la riutilizzabilità)

MODULES

- Un modulo è solo un file JavaScript
- I moduli possono caricarsi a vicenda e utilizzare direttive speciali, esportare e funzionalità di importazione in interscambio
- export permette di esportare etichette, variabili e funzioni che devono essere accessibili al di fuori del modulo corrente
- import consente l'importazione di funzionalità specifiche (ad esempio: variabili, funzioni) da altri moduli (a condizione che queste funzionalità vengano esportate!)

MODULES: EXAMPLE

- Suppose you want to re-use some code you developed for some other projects: a **pet.js** file and a **greet.js** file

```
//pet.js
function Pet(name, age){
  this.name=name; this.age=age;
}
let msg = "Hello";
function greet(pet){
  console.log(` ${msg}, I'm ${pet.name}`);
}
```

```
//greet.js
let msg = "Howdy";

function greet(name, message=msg) {
  console.log(` ${message}, ${name}`);
}
```

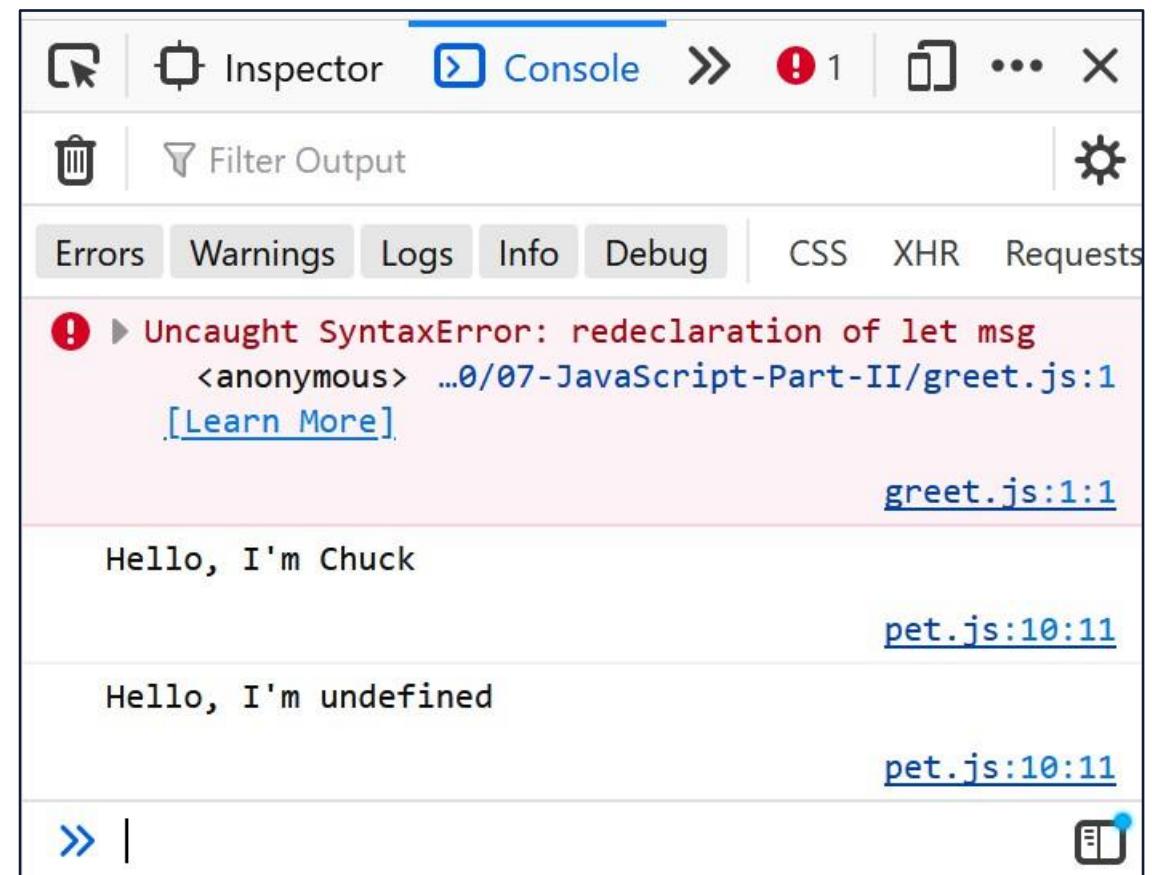
```
//script.js (The new code we need to implement)
let chuck = new Pet("Chuck", 7); // Pet is defined in pet.js
greet(chuck); // Desiderata: Hello, I'm Chuck
greet("Web Technologies"); // Desiderata: Howdy, Web Technologies
```

MODULES: EXAMPLE

- One might try to include all the scripts in the web page

```
<script src="./pet.js"></script>
<script src="./greet.js"></script>
<script src="./script.js"></script>
```

```
//script.js (The new code)
let chuck = new Pet("Chuck", 7);
// Pet is defined in pet.js
greet(chuck);
// Desiderata: Hello, I'm Chuck
greet("Web Technologies");
// Desiderata: Howdy, Web Technology
```



MODULES: EXAMPLE

- Per farla funzionare, dovremmo cambiare il codice che vogliamo riutilizzare, ad esempio utilizzando identificatori diversi per variabili e funzioni
- Questo non è l'ideale e vanifica lo scopo principale di riutilizzare il codice esistente così com'è
- I moduli vengono in soccorso!

```
//pet-module.js
export function Pet(name, age){
  this.name=name; this.age=age;
}
let msg = "Hello"; //no need to export

export function greet(pet){
  console.log(` ${msg}, I'm ${pet.name}`);
}
```

```
//greet-module.js
let msg = "Howdy"; //no need to export

export function greet(name, message=msg) {
  console.log(` ${message}, ${name}`);
}
```

MODULES: EXAMPLE

- Nel documento HTML, dobbiamo solo includere lo script principale, come modulo:`<script type="module" src="./script-module.js"></script>`

```
//script-module.js, other modules are imported given their URLs
import {Pet, greet as greetPet} from './pet-module.js';
import {greet} from './greet-module.js';

let chuck = new Pet("Chuck", 7);

greetPet(chuck); //Hello, I'm Chuck

greet("Web Technologies"); //Howdy, Web Technologies
```

REFERENCES

- **The Modern JavaScript Tutorial**

Freely available at <https://javascript.info/> or on [GitHub](#)

Part 1: Prototypes and inheritance, Data types (5.4 to 5.7), Classes (9.1 to 9.4, 9.6), Generators and advanced iteration (12.1), Modules (13.1, 13.2)

- **Eloquent JavaScript (3rd edition)**

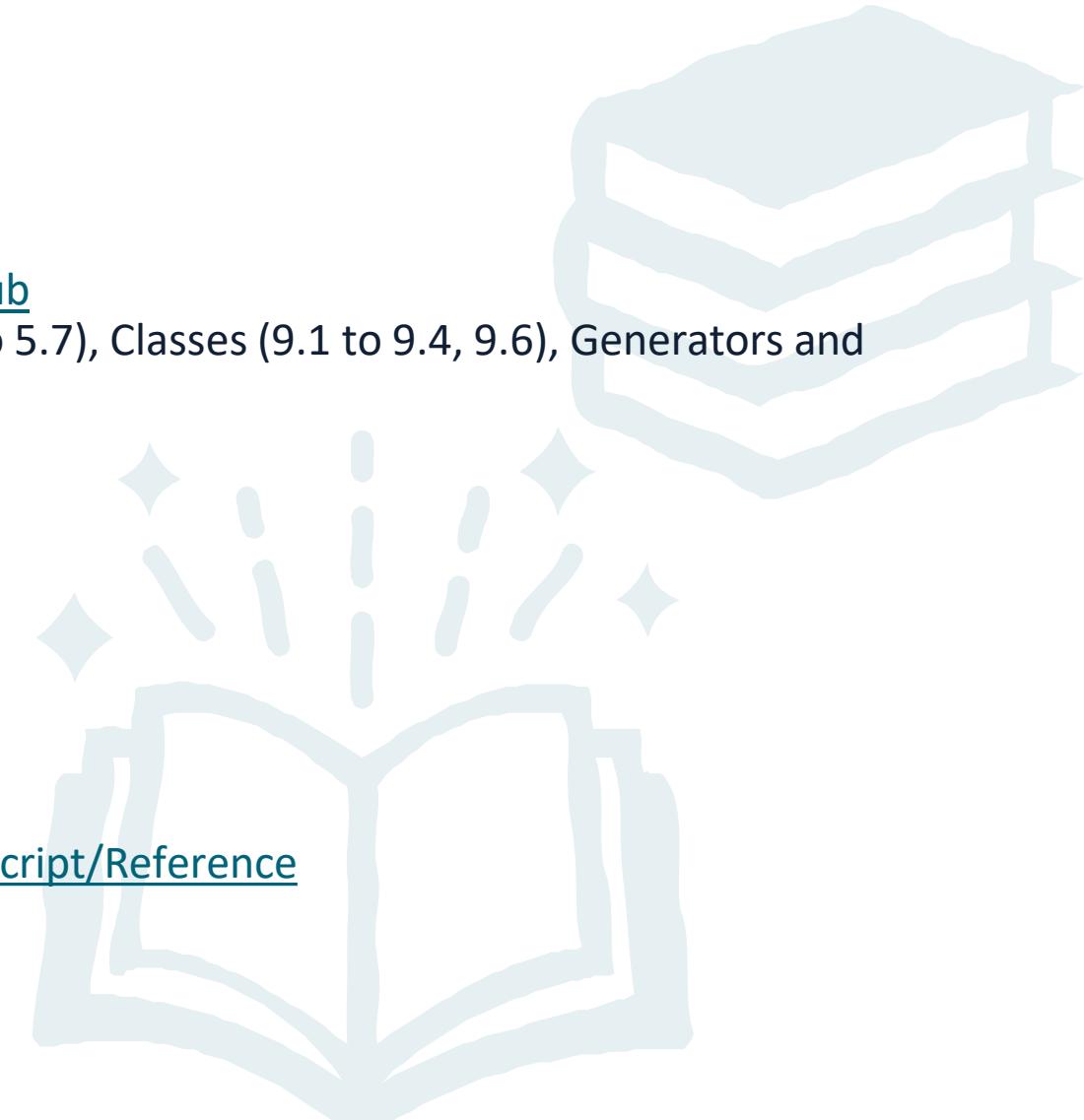
By Marijn Haverbeke

Freely available at <https://eloquentjavascript.net/>
Chapters 6, 10

- **JavaScript Reference**

MDN web docs

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>



UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
WEB TECHNOLOGIES – LECTURE 07

JAVASCRIPT IN A BROWSER ENVIRONMENT

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



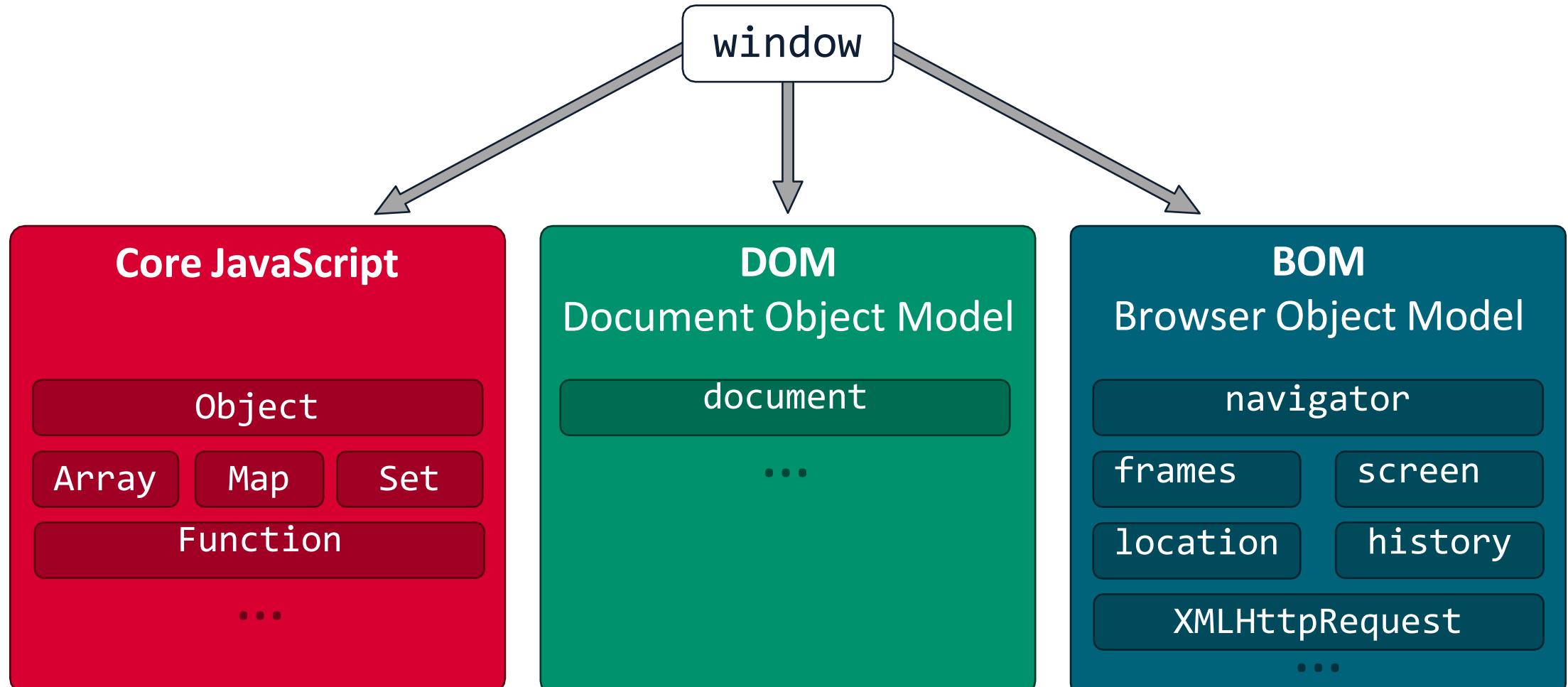
PREVIOUSLY, ON WEB TECHNOLOGIES

So far we've learned the core concepts of the JavaScript language.

JavaScript can run on a number of different **host environments**:

- **Web Browsers** (for which JavaScript was originally created)
- **Web Servers** (e.g.: via the Node.js runtime)
- Host environments provide their own **objects** and **functions**, in addition to those included in the **language core**
- Today, we'll learn about the **web browser host environment**

THE BROWSER ENVIRONMENT: OVERVIEW



THE BROWSER ENVIRONMENT: WINDOW

- L'ambiente del browser presenta un oggetto «root» chiamato window
- L'oggetto window:
- È un oggetto globale per il codice JavaScript
- Rappresenta la «finestra del browser» e fornisce il metodo per controllarla
- Le funzioni globali e le variabili sono proprietà dell'oggetto window

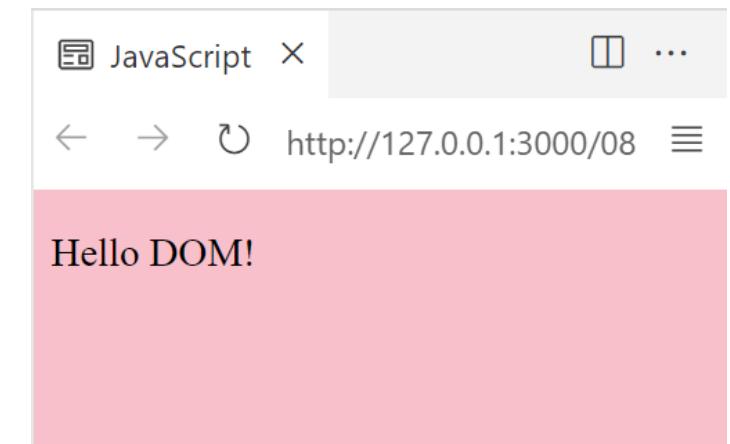
```
let msg = "Hello!";
function greet(name){
  console.log(` ${msg}, ${name}! `);
}

greet("Web Technologies"); //Hello, Web Technologies!
window.greet("window object"); //Hello, window object!
```

THE DOCUMENT OBJECT MODEL (DOM)

- Il DOM rappresenta il contenuto del documento corrente
- L'oggetto documento è il punto di ingresso principale alla pagina web
- Fornisce modi per accedere e manipolare i contenuti

```
<body>
  <p>Hello!</p>
  <script>
    document.body.style.background = "pink";
    document.querySelector("p").innerText="Hello DOM!"
  </script>
</body>
```



THE BROWSER OBJECT MODEL (BOM)

- Il BOM include oggetti aggiuntivi forniti dal browser per l'utilizzo del browser stesso, non del contenuto del documento

```
//The location object contains information on the URL of the current document
console.log(location.href); //http://localhost:3000/08-JavaScript...

//screen contains information about the display used by the user
console.log(screen.availWidth); //2048

//navigator contains additional details about the web browser and platform
console.log(navigator.userAgent); //Mozilla/5.0 (Windows NT 10.0...

//history represents the navigation history
history.back(); //same as clicking on the "back" button in the browser GUI
```

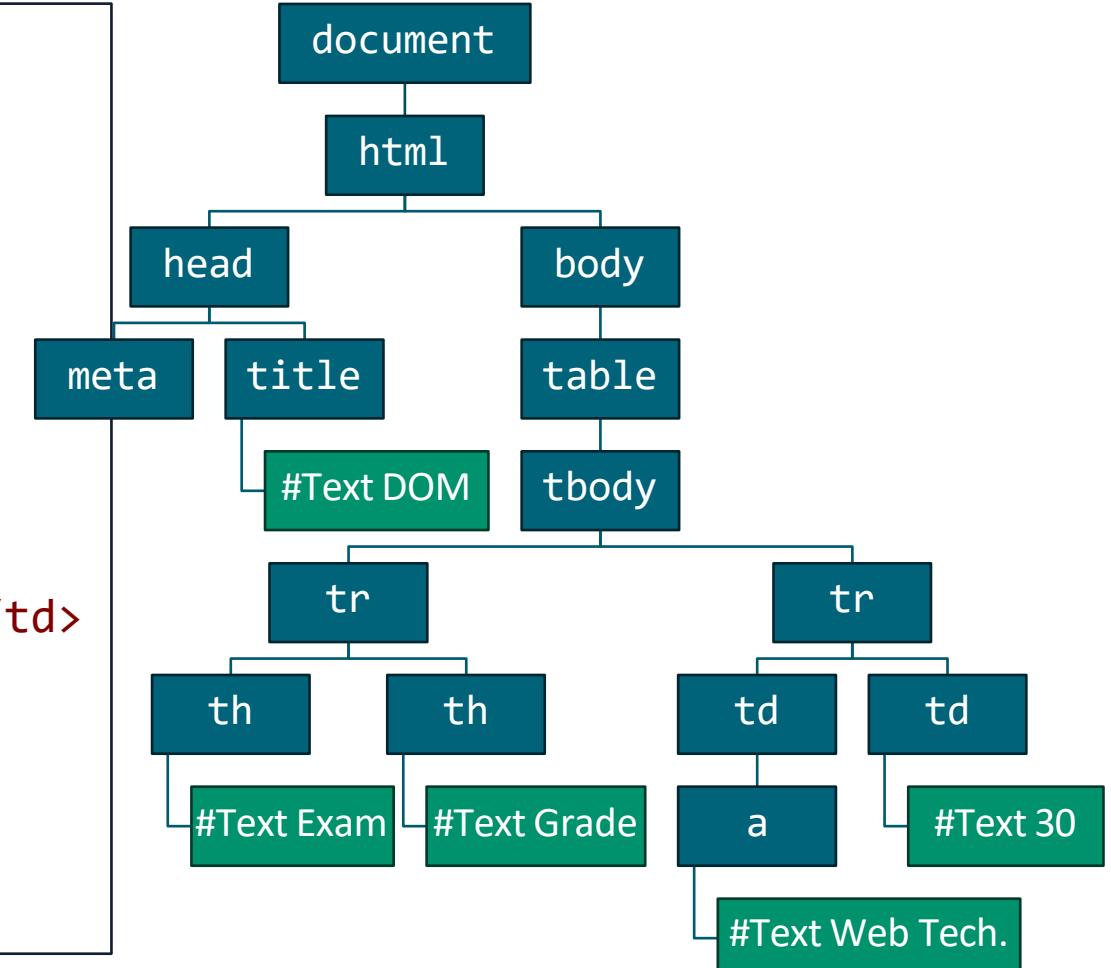
WORKING WITH THE DOM

THE DOM

- Quando abbiamo appreso dei selettori CSS, abbiamo accennato al fatto che i documenti HTML potevano essere visti come alberi
- Abbiamo parlato di discendenti, figli, fratelli...
- Il DOM formalizza questa intuizione
- Ogni tag HTML è un oggetto nel DOM
- I tag nidificati sono figli di quelli che li contengono
- Anche il contenuto del testo di un tag è un oggetto e un elemento secondario del tag
- Gli attributi di un tag sono accessibili come proprietà dell'oggetto corrispondente

THE DOM: EXAMPLE

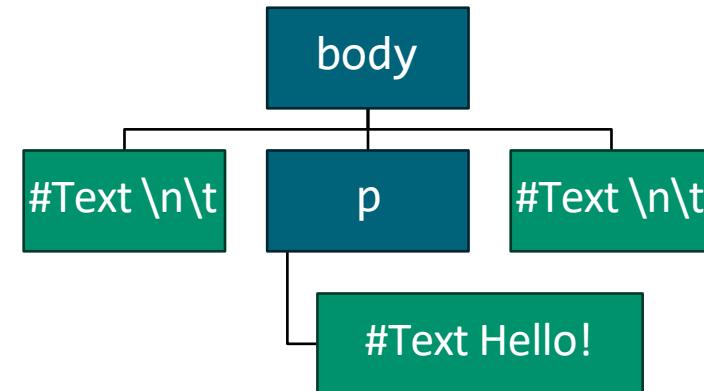
```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>DOM</title>
</head>
<body>
  <table>
    <tbody>
      <tr><th>Exam</th><th>Grade</th></tr>
      <tr><td><a href="/wt">Web Tech.</a></td>
          <td>30</td></tr>
    </tbody>
  </table>
</body>
</html>
```



THE DOM: A NOTE ABOUT THE EXAMPLE

- L'esempio nella diapositiva precedente è un po' una semplificazione
- Il DOM vero e proprio include anche spazi e nuove righe tra i tag come nodi di testo. Per brevità, generalmente ometteremo tali nodi

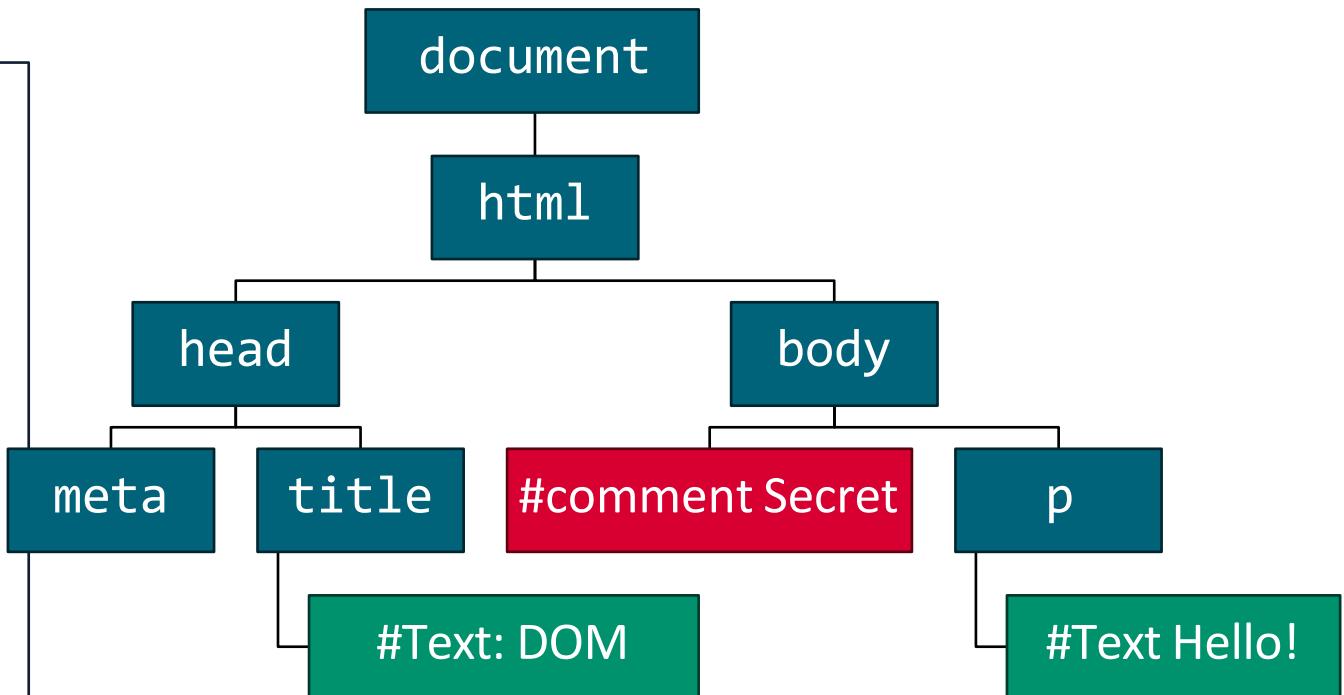
```
<!DOCTYPE html>
<html>
  <head><meta charset="utf-8">
    <title>JavaScript</title>
  </head>
  <body>
    <p>Hello!</p>
  </body>
</html>
```



THE DOM: COMMENTS

- I commenti HTML non vengono visualizzati dai browser Web
- Ciononostante, vengono ancora analizzati e aggiunti al DOM come nodi di commento speciali

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>JavaScript</title>
  </head>
  <body>
    <!-- Secret -->
    <p>Hello!</p>
  </body>
</html>
```



THE DOM: NODE TYPES

- La specifica DOM definisce 12 diversi tipi di nodi
- In genere, lavoreremo con quattro tipi di nodi:
- nodo del documento: è la radice e il punto di ingresso del DOM
- Nodi elemento: rappresentano elementi HTML
- Nodi di testo: rappresentano il contenuto del testo
- Nodi di commento: rappresentano i commenti

THE DOM: FINDING ELEMENTS

- L'oggetto documento fornisce diversi metodi per selezionare elementi da un documento HTML

```
<ul>
  <li id="a">A</li>
  <li class="foo bar">B</li>
  <li name="c">C</li>
</ul>
<script>
  console.log(document.getElementById("a")); //select the first list item
  console.log(document.getElementsByClassName("foo")); //HTMLCollection (1)
  console.log(document.getElementsByName("c")); //NodeList (1)
  console.log(document.getElementsByTagName("li")); //HTMLCollection (3)
</script>
```

THE DOM: FINDING ELEMENTS (2)

`document.querySelector(css)` e `querySelectorAll(css)` sono i metodi più versatili e prendono come input un selettore css

`querySelector` restituisce solo il primo elemento (se presente) corrispondente al selettore, mentre `querySelectorAll` restituisce un elenco di corrispondenze

Tutti i metodi restituiscono null quando non riescono a individuare gli elementi

```
<ul>
  <li id="a">A</li> <li class="foo bar">B</li> <li name="c">C</li>
</ul>
<script>
  console.log(document.querySelector("ul > li")); //first li element
  console.log(document.querySelectorAll("ul > li")); //NodeList (3)
  console.log(document.querySelector("ul > li.myclass")); //null
</script>
```

THE DOM: FINDING ELEMENTS (3)

Tutti i metodi della famiglia `getElementsBy*` restituiscono una `HTMLCollection`, nota anche come «live collection»

Tali raccolte vengono aggiornate automaticamente e riflettono sempre lo stato corrente del documento

```
<a href="/">First link</a>
<script>
  let links = document.getElementsByTagName("a");
  console.log(links); //HTMLCollection (1)
</script>
<a href="/">Second Link</a> <a href="/">Third Link</a>
<script>
  console.log(links); //HTMLCollection (3)
</script>
```

THE DOM: FINDING ELEMENTS (4)

Quando vengono richiamati sul documento, i metodi per cercare l'elemento cercano corrispondenze nell'intero documento HTML.

Se gli elementi di cui abbiamo bisogno sono discendenti di un altro elemento, potrebbe essere più efficiente invocare i metodi di ricerca sull'elemento antenato stesso.

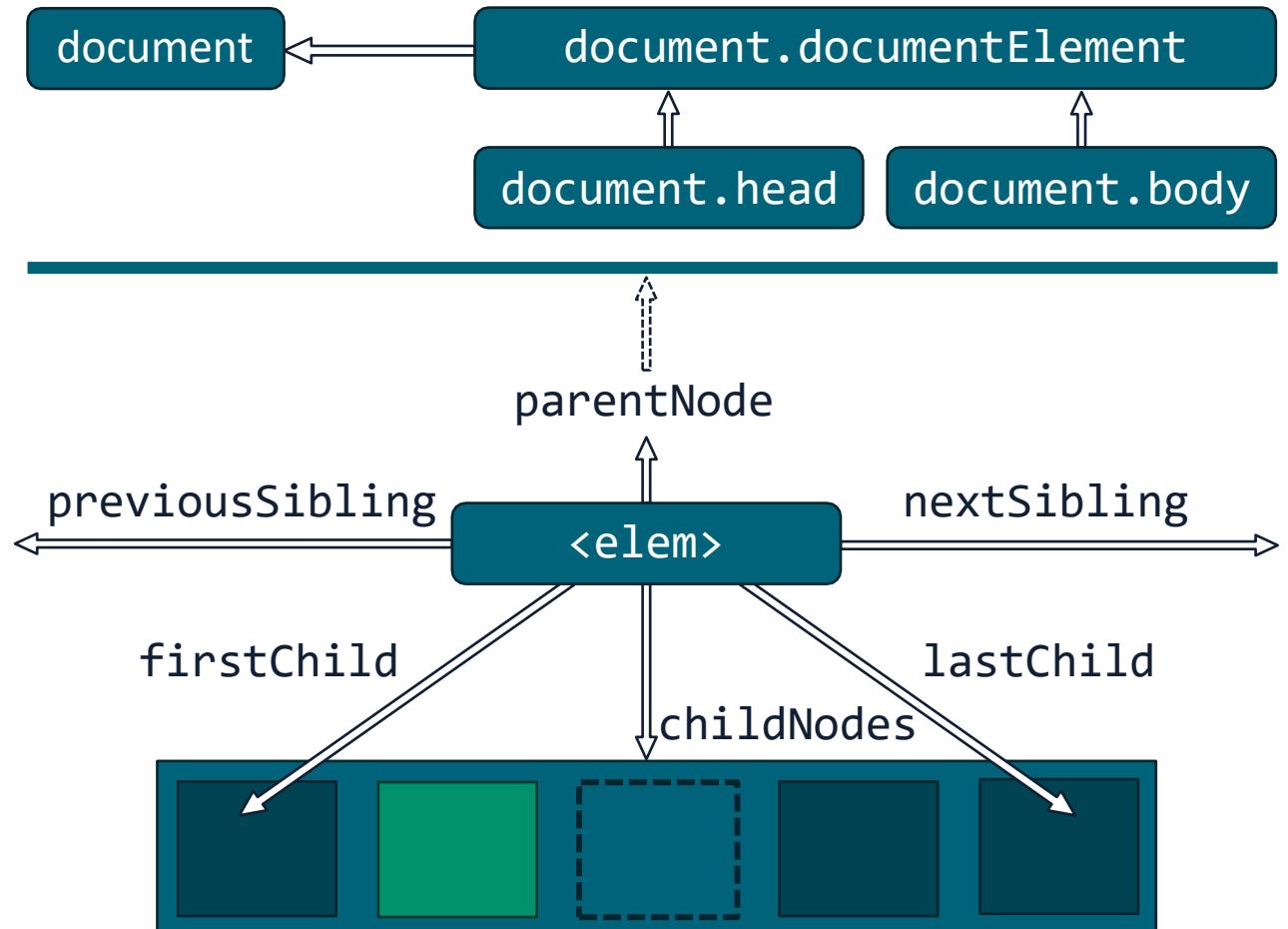
```
<a href="/">First link</a>
<script>
  let sidebar = document.getElementById("sidebar");
  let sidebarLinks = sidebar.querySelectorAll("a"); //only searches within sidebar
</script>
```

THE DOM: FINDING ELEMENTS RECAP

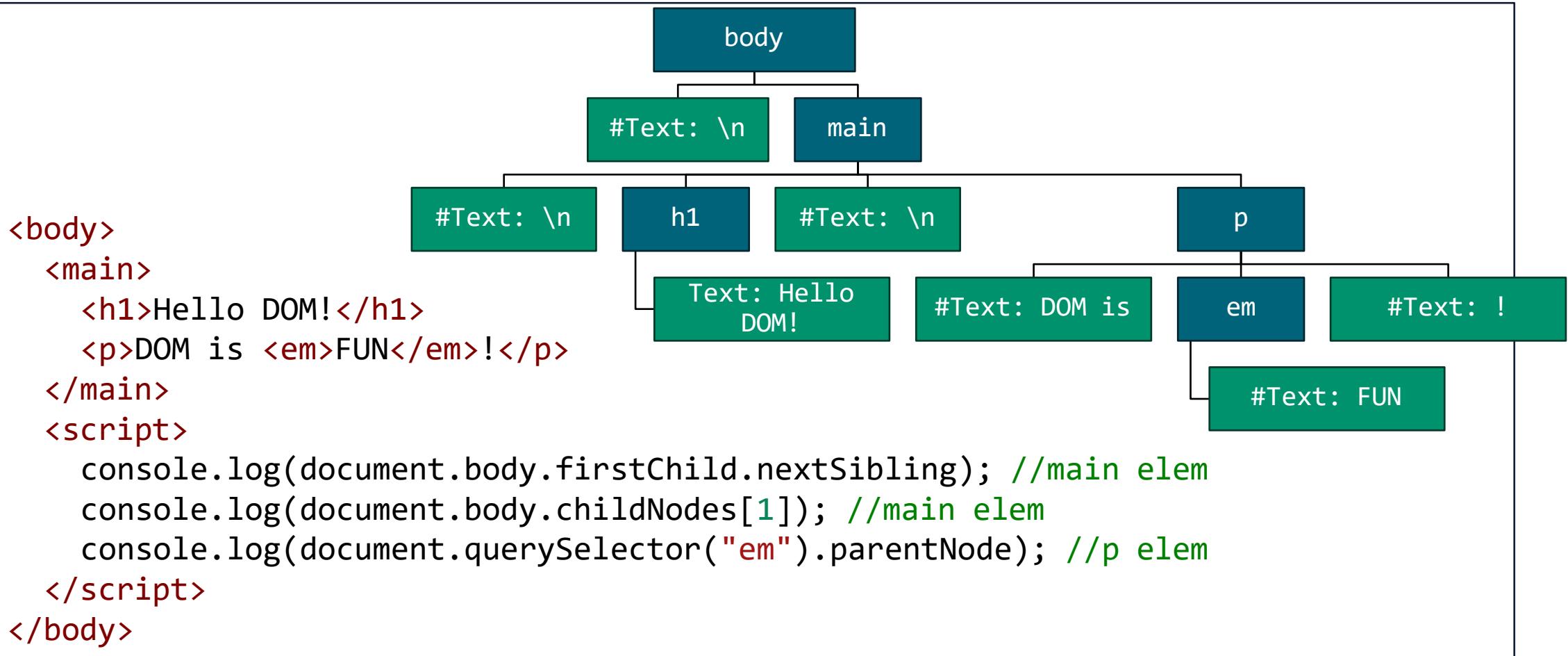
Method	Searches by...	Can be called on elements?	Returns Live Collection?
<code>querySelector</code>	CSS selector	✓	-
<code>querySelectorAll</code>	CSS selector	✓	-
<code>getElementById</code>	Id attribute	-	-
<code>getElementsByName</code>	Name attribute	-	✓
<code>getElementsByTagName</code>	Tag name or '*'	✓	✓
<code>getElementsByClassName</code>	Class attribute	✓	✓

NAVIGATING THE DOM

- Gli elementi HTML più in alto (html, head, body) sono disponibili anche come proprietà del documento
- Gli oggetti nodo DOM generali contengono riferimenti all'elemento padre, agli elementi di pari livello e agli elementi figlio
- Questi riferimenti sono di sola lettura!

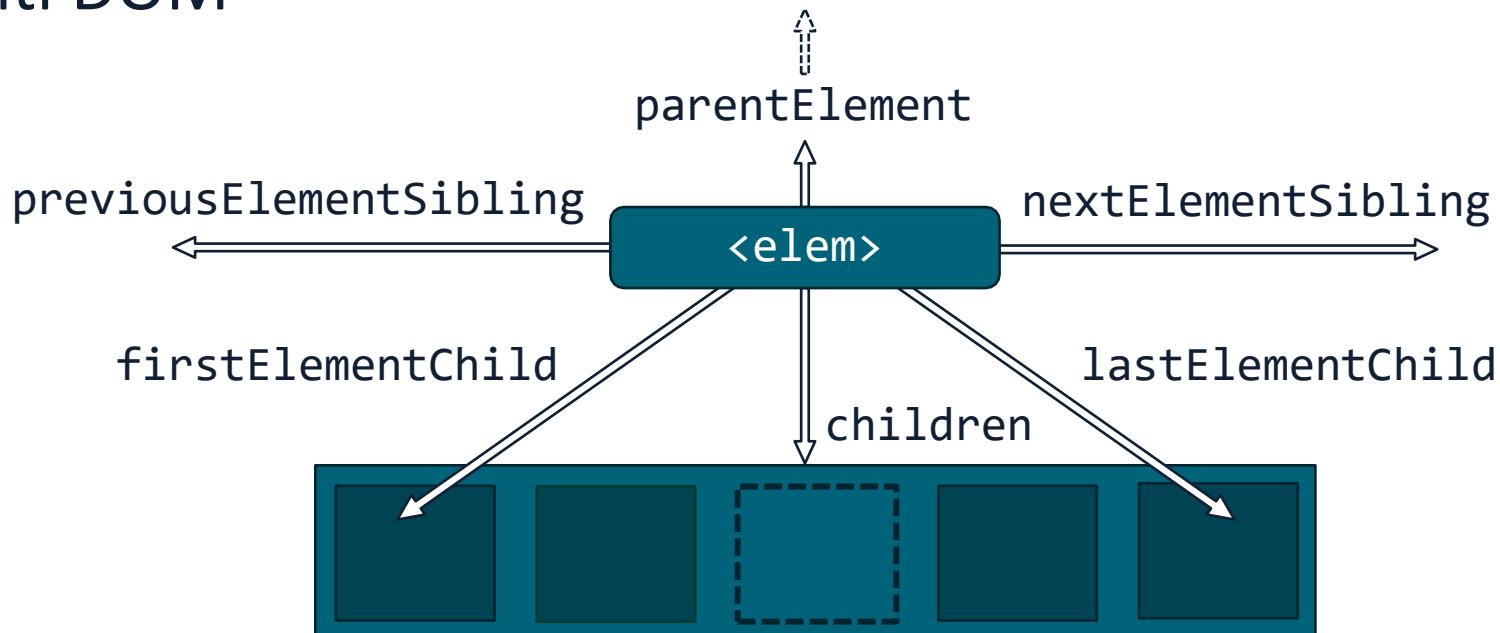


NAVIGATING THE DOM: EXAMPLE



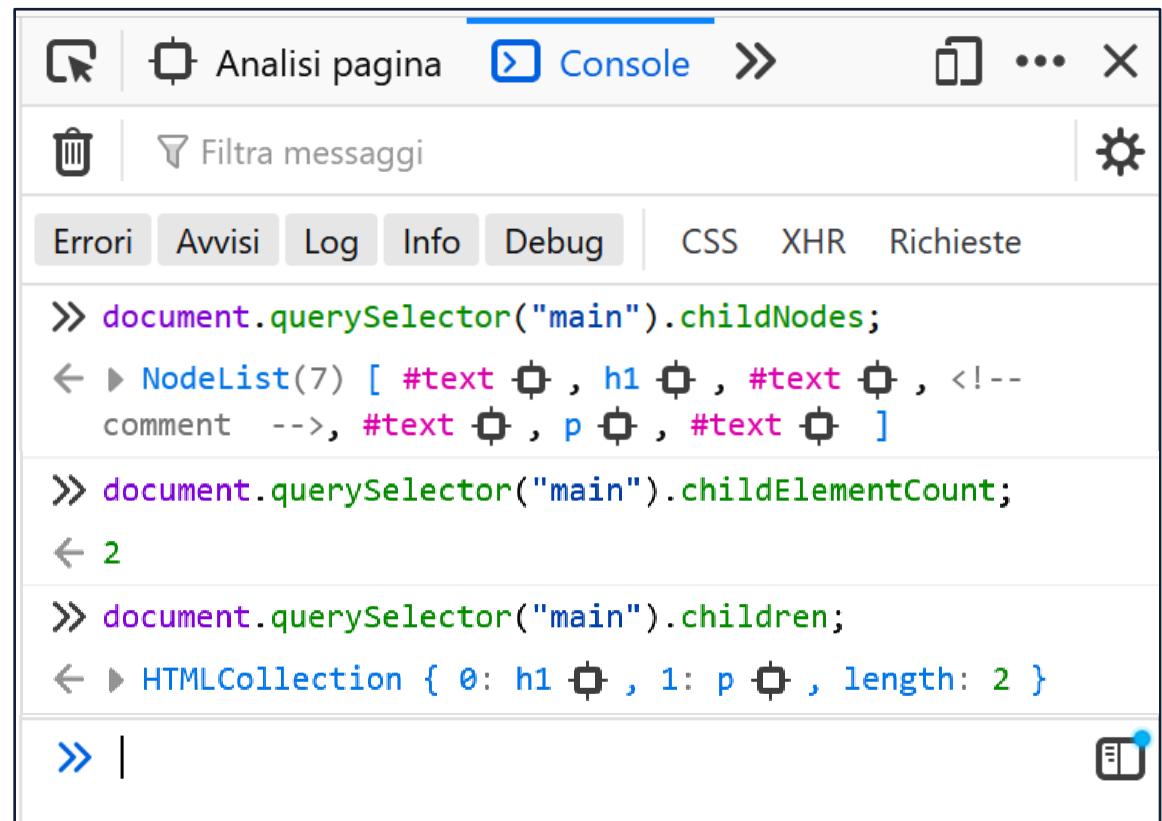
ELEMENT-ONLY DOM NAVIGATION

- Potremmo voler ignorare i nodi di testo o di commento durante la navigazione nel DOM e concentrarci solo sugli elementi DOM.
- Le proprietà di navigazione seguenti considerano solo gli elementi DOM



NAVIGATING THE DOM

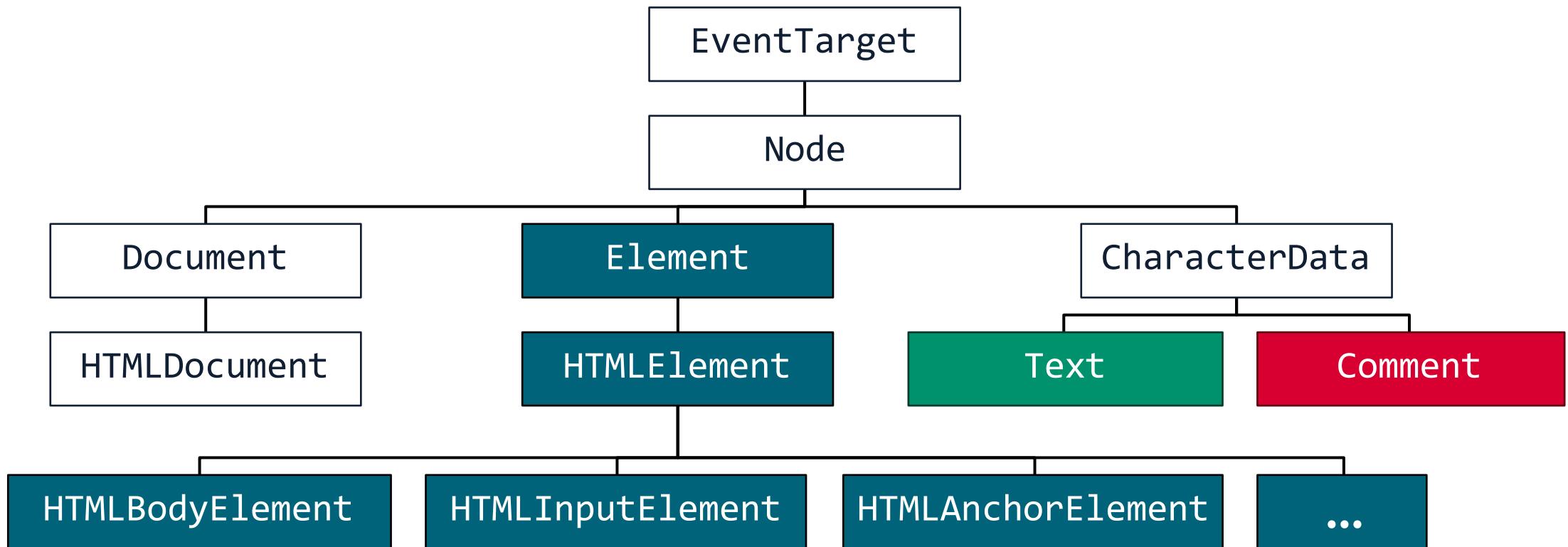
```
<main>
  <h1>Hello DOM!</h1>
  <!-- comment -->
  <p>DOM is <em>Fun</em>!</p>
</main>
```



The screenshot shows the browser's developer tools open to the 'Console' tab. The console interface includes a toolbar with icons for back, forward, search, and refresh, followed by 'Analisi pagina' and 'Console'. Below the toolbar is a filter bar with a trash icon and a dropdown for 'Filtrare messaggi'. The main area has tabs for Errori, Avvisi, Log, Info, Debug (which is selected), CSS, XHR, and Richieste. The console output shows the following code and its results:

```
» document.querySelector("main").childNodes;
← ▶ NodeList(7) [ #text , h1 , #text , <!--
comment -->, #text , p , #text ]
» document.querySelector("main").childElementCount;
← 2
» document.querySelector("main").children;
← ▶ HTMLCollection { 0: h1 , 1: p , length: 2 }
» |
```

DOM NODES CLASS HIERARCHY



NODE PROPERTIES

I nodi DOM hanno alcune proprietà utili a cui possiamo accedere: nodeName/tagName può essere utilizzato per accedere alle informazioni sul tipo di nodo. Queste proprietà sono di sola lettura.

```
<body>
  <h1 id="h">Hello <em>DOM!</em></h1>
  <!-- comment -->
  <script>
    let body = document.body;
    console.log(body.firstChild.nodeName);          // #text
    console.log(body.firstChild.tagName);           // undefined
    console.log(body.childNodes[3].nodeName);       // #comment
    console.log(body.firstElementChild.nodeName);   // H1
    console.log(body.firstElementChild.tagName);    // H1
  </script>
</body>
```

NODE PROPERTIES

Anche gli attributi standard vengono analizzati e resi disponibili come proprietà degli elementi corrispondenti

```
<body>
  <input id="x" class="inp ok" name="field" custom="y">
  <script>
    let i = document.body.firstChild;
    console.log(i.id); //x
    console.log(i.classList); //DOMTokenList ["inp", "ok"]
    console.log(i.name); //field
    console.log(i.custom); //undefined, because custom is not a standard attrib.
    console.log(i.getAttribute("custom")); //y
  </script>
</body>
```

NODE PROPERTIES

These properties are also **writable**!

```
<body>
  <input id="x" class="inp ok" name="field" custom="y">
  <script>
    let i = document.body.firstChild;

    i.id = "w";
    i.classList.remove("ok"); i.classList.add("err");
    i.setAttribute("custom", "k");
    i.removeAttribute("name");
  </script>
</body>

<!-- the input after the script executes --&gt;
&lt;input id="w" class="inp err" custom="k"&gt;</pre>
```

MODIFYING THE DOM STRUCTURE

- Non ci limitiamo a modificare gli elementi DOM esistenti
- Ad esempio: modificando i loro attributi
- Possiamo creare dinamicamente nuovi elementi e rimuovere quelli esistenti e modificare la struttura del DOM
- L'aggiornamento dinamico del DOM è la chiave per creare pagine web reattive e «live»

NODE PROPERTIES: innerHTML / outerHTML

- **innerHTML** può essere utilizzato per accedere al codice HTML contenuto in un **HTMLElement**
- **outerHTML** può essere utilizzato per accedere all'intero codice HTML di un **HTMLElement**

```
<body>
  <h1 id="h">Hello <em>DOM!</em></h1>
  <!-- comment -->
  <script>
    let body = document.body;
    console.log(body.firstChild.innerHTML); //Hello <em>DOM!</em>
    console.log(body.firstChild.innerHTML); //undefined (not an HTMLElement)
    console.log(body.firstChild.outerHTML); //<h1 id="h">Hello <em>DOM!</em></h1>
  </script>
</body>
```

NODE PROPERTIES: innerHTML / outerHTML

Una cosa divertente dell'innerHTML e dell'outerHTML è che sono scrivibili
Possiamo assegnare loro nuovi valori per modificare il contenuto della pagina!

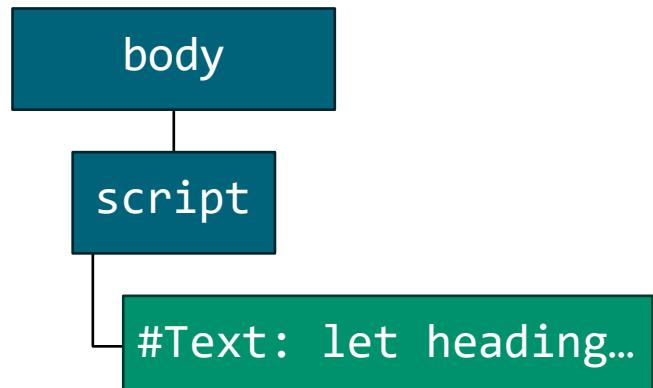
```
<body>
  <h1 id="h">Hello <em>DOM!</em></h1>
  <!-- comment -->
  <script>
    let elem = document.body.firstChild;

    elem.innerHTML = "DOM: <em>Wow!</em>";
    //We changed the content of the <h1> element

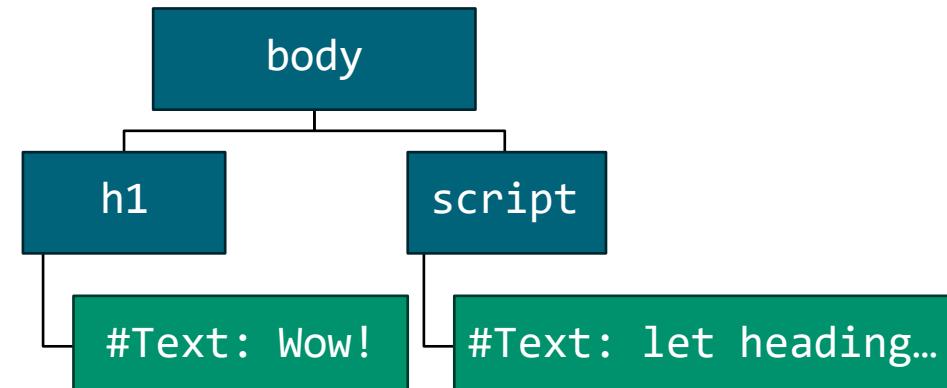
    elem.outerHTML = `<p>${elem.innerHTML}</p>`;
    // Now the <h1> element is a <p>, with the same content as before!
  </script>
</body>
```

CREATING NEW DOM ELEMENTS

```
<body>
  <script>
    let heading = document.createElement("h1"); //create a new <h1> elem
    // the new <h1> is not yet in the DOM
    let title = document.createTextNode("Wow!"); //create a new #text node
    // the new text node is not yet in the DOM
    heading.append(title); //add the #text node as a child of the <h1>
    document.body.prepend(heading); //<h1> (and its child) at the beginning of <body>
  </script>
</body>
```

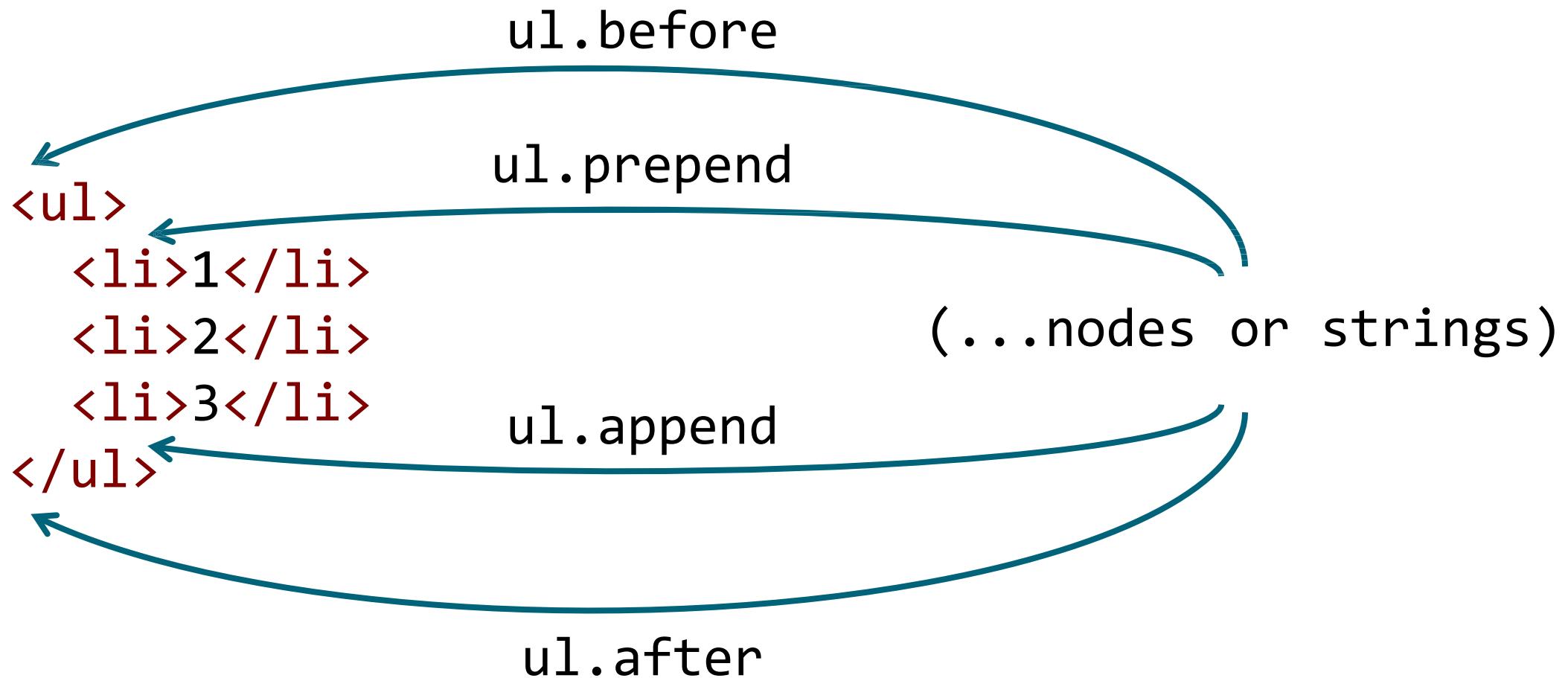


DOM Tree **before** executing the script



DOM Tree **after** executing the script

DOM MANIPULATION METHODS



DOM MANIPULATION: EXAMPLE

```
<body>
  <ul>
    <li class="kw">CSS</li>
  </ul>
</body>
```

```
let ul = document.querySelector("ul");
let html = ul.querySelector("li").cloneNode();
let js = document.createElement("li");
html.innerHTML = "HTML";
js.innerHTML = "JavaScript";
ul.before("Keywords:");
ul.prepend(html);
ul.append(js);
ul.after("End of keyword list");
```



```
<body>
  Keywords:
  <ul>
    <li class="kw">HTML</li>
    <li class="kw">CSS</li>
    <li>JavaScript</li>
  </ul>
  End of keyword list
</body>
```

DOM MANIPULATION: EXAMPLE (2)

```
<body>
  <ul>
    <li class="kw">CSS</li>
  </ul>
</body>
```

```
let ul = document.querySelector("ul");
let ol = document.createElement("ol");

ol.append(ul.firstElementChild);
ul.replaceWith(ol);
```



```
<body>
  <ol>
    <li class="kw">CSS</li>
  </ol>
</body>
```

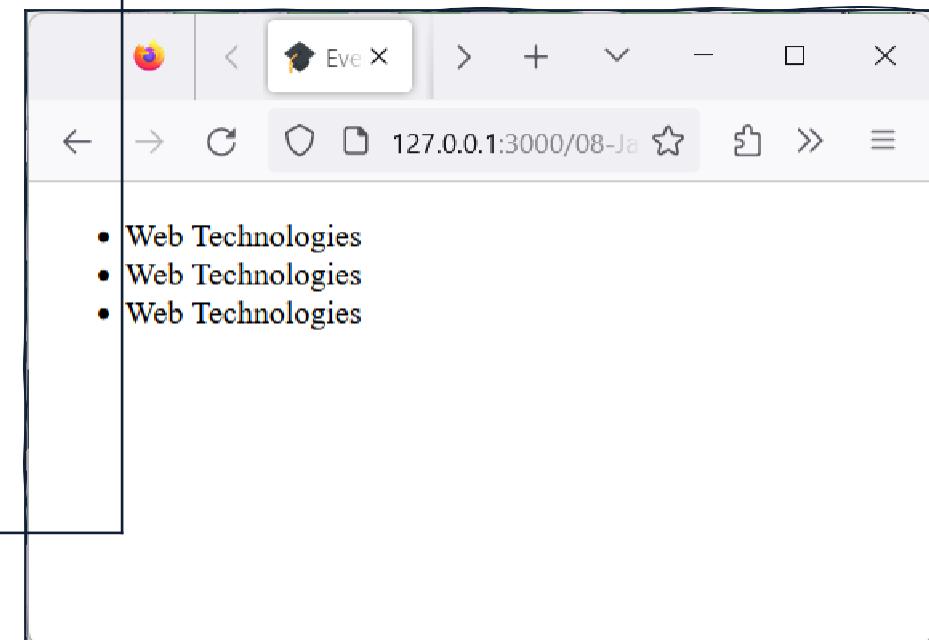
DOM MANIPULATION: RECAP

Method	Description
<code>document.createElement(tag)</code>	creates an element with the given tag
<code>document.createTextNode(value)</code>	creates a text node (rarely used)
<code>elem.cloneNode(deep)</code>	clones the element, if deep==true with all descendants
<code>node.append(...nodes or strings)</code>	insert into node, at the end
<code>node.prepend(...nodes or strings)</code>	insert into node, at the beginning
<code>node.before(...nodes or strings)</code>	insert right before node
<code>node.after(...nodes or strings)</code>	insert right after node
<code>node.replaceWith(...nodes or strings)</code>	replace node
<code>node.remove()</code>	remove the node

INTERACTING WITH USERS

```
<ul></ul>
<script>
  alert("This page will ask for your input");
  if(confirm("Do you want to continue?") === true) {
    let name = prompt("State your name:");
    let num = parseInt(prompt("Insert a number:"));
    for(let i = 0; i < num; i++) {
      let li = document.createElement("li");
      li.innerHTML = name;
      document.querySelector("ul").append(li);
    }
  } else {
    console.log("user cancelled");
  }
</script>
```

alert, **prompt**, and **confirm** sono modi utili per interagire con gli utenti in un browser web.



EVENTS

BROWSER EVENTS

Gli eventi sono segnali che qualcosa è successo. Potrebbero essere correlati a:

Comportamento dell'utente:

Clic, pressioni di tasti sulla tastiera, movimenti del mouse...

Modulistica:

Invio, focus (o perdita) su un campo di input

Eventi del documento:

Documento o elementi caricati, richieste di rete completate...

USEFUL BROWSER EVENTS

Mouse	<code>click</code>	User clicks (taps, for touchscreen devices) on element
	<code>contextMenu</code>	Mouse right-click on element
	<code>mouseover/mouseout</code>	Mouse cursor goes over / leaves an element
	<code>mousedown/mouseup</code>	The main mouse button is pressed / released over an element
	<code>mousemove</code>	The mouse cursor moves
Keyboard	<code>keydownkeyup</code>	A keyboard keys is pressed / released
Forms	<code>submit</code>	User submits a form
	<code>focus</code>	User focuses on an element (e.g.: <input>)
Document	<code>DOMContentLoaded</code>	The HTML has been parsed, DOM is fully built
Window	<code>load</code>	The web pages has been completely loaded and rendered

EVENT HANDLERS

- Per reagire agli eventi, è possibile definire le funzioni del gestore
- Si tratta di funzioni che vengono eseguite quando viene attivato un evento
- Il modo più semplice per definire i gestori consiste nell'utilizzare gli `<event>` attributi HTML. Il valore dell'attributo è il codice JavaScript da eseguire

```
<input type="button" value="Click me" onclick="handleClick();>

<script>
  function handleClick(){
    console.log("Click handled");
  }
</script>
```

EVENT HANDLERS: DYNAMIC DEFINITION

- I gestori di eventi possono anche essere definiti in modo dinamico, tramite JavaScript
- Questa operazione può essere eseguita scrivendo la <event> proprietà on sugli elementi HTML o utilizzando il metodo addEventListener

```
<input type="button" value="Click me" onclick="handleClick();>

<script>
    function handleClick(){ console.log("Click handled"); }

    let input = document.querySelector("input");
    input.onclick = () => {console.log("Tap");}; // overrides HTML-attrib. handler

    input.addEventListener("click", handleClick); // adds new handler function
</script>
```

EVENT HANDLERS: VALUE OF THIS

- All'interno di un gestore eventi, questo viene valutato dall'elemento a cui è assegnato il gestore richiamato

```
<input type="button" value="Click me">
<p>Hello Events!</p>

<script>
  function handleClick(){
    console.log(this.outerHTML);
  }
  document.querySelector("input").addEventListener("click", handleClick);
  // <input type="button" value="Click me">
  document.querySelector("p").addEventListener("click", handleClick);
  // <p>Hello Events!</p>
</script>
```

EVENT HANDLERS: THE EVENT OBJECT

Per gestire correttamente gli eventi, potremmo aver bisogno di maggiori dettagli su di essi

In un evento keydown, quale tasto è stato premuto sulla tastiera?

In un evento click, in quale coordinata è avvenuto il click? L'utente ha anche premuto CTRL sulla tastiera, mentre faceva clic?

Tutti i dettagli sull'evento vengono forniti come oggetto evento, che viene passato ai gestori eventi

Diversi tipi di eventi hanno proprietà diverse. Ad esempio, gli eventi correlati alla tastiera hanno una proprietà del tasto che rappresenta il tasto premuto

THE EVENT OBJECT: EXAMPLE

```
<input type="text" placeholder="Write here" autofocus>
<script>
  function handler(event){
    console.log(`You pressed ${event.key}`);
  }
  document.querySelector("input").addEventListener("keyup", handler);
</script>
```

EVENT BUBBLING

Quando un evento viene attivato su un elemento elem:

Innanzitutto, vengono eseguiti tutti i gestori registrati su elem per quell'evento

Quindi, tutti i gestori per l'evento vengono eseguiti sul genitore di elem

I gestori vengono eseguiti fino a quando il
il nodo radice del DOM viene raggiunto

Questo meccanismo è noto come bubbling degli eventi

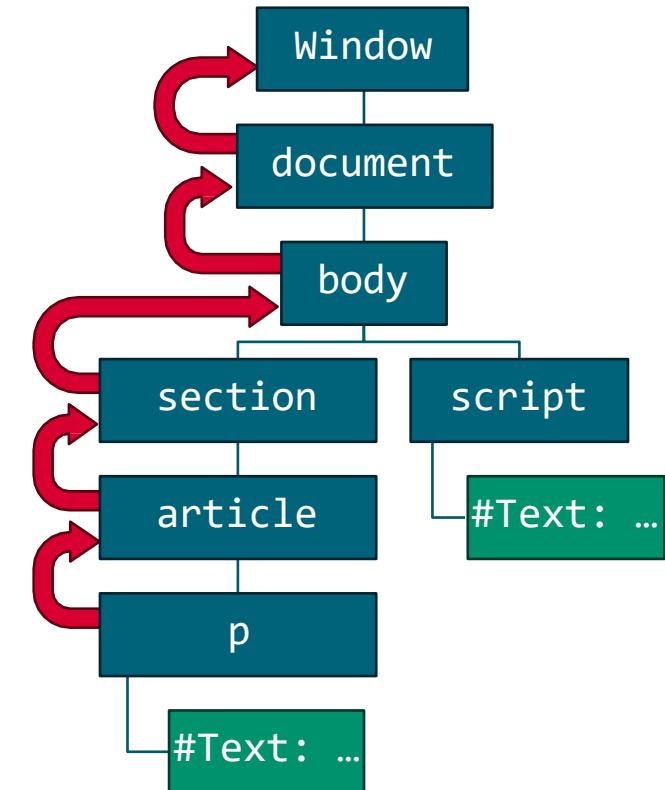
Si applica alla maggior parte degli eventi, ma non a tutti (ad esempio: gli eventi focus non si propagano in bolle)

EVENT BUBBLING: EXAMPLE

```
<body>
  <section onclick="console.log('section');">
    <article onclick="console.log('article');">
      <p onclick="console.log('p');">Bubbling</p>
    </article>
  </section>
  <script>
    document.onclick = () => console.log("document");
  </script>
</body>
```

Console output after a click on <p>:

p
article
section
document



EVENT DELEGATION

La delega degli eventi è un modello potente per la gestione degli eventi

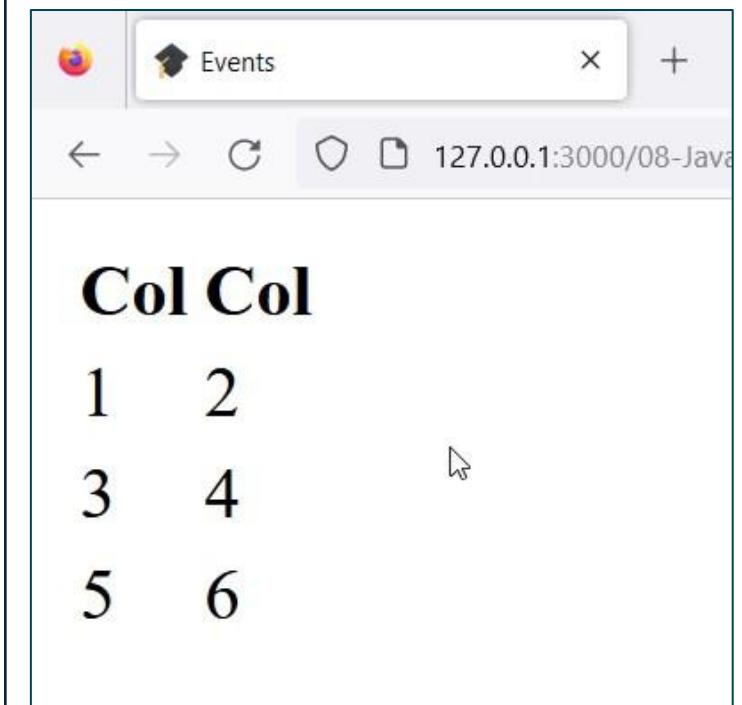
Idea: se molti elementi possono generare eventi simili, invece di assegnare un gestore a ciascuno di essi, utilizziamo un singolo gestore sul loro antenato comune

Gli elementi delegano la gestione degli eventi a un predecessore

La proprietà target dell'oggetto evento può essere utilizzata per determinare l'elemento in cui è stato generato l'evento

EVENT DELEGATION: EXAMPLE

```
<table onclick="handler(event);>
  <tr><th>Col</th><th>Col</th></tr>
  <tr><td>1</td><td>2</td></tr>
  <tr><td>3</td><td>4</td></tr>
  <tr><td>5</td><td>6</td></tr>
</table>
<script>
  function handler(event){
    if(event.target.nodeName === "TD"){
      let oldValue = parseInt(event.target.innerHTML);
      event.target.innerHTML = ++oldValue;
    }
  }
</script>
```

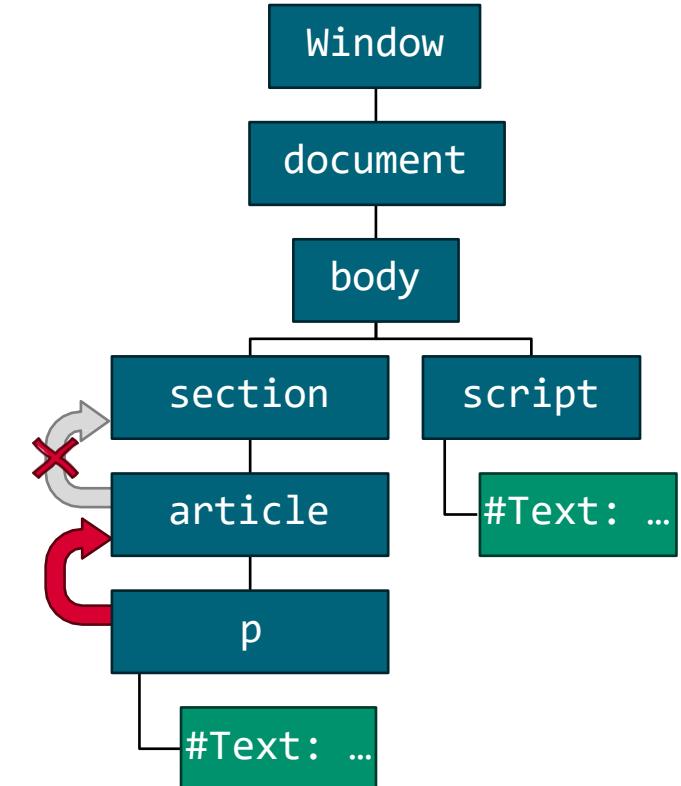


STOPPING EVENT BUBBLING

- Un evento di bubbling ha origine da un determinato elemento e si propaga
- (si gonfia) fino all'oggetto finestra globale.
- In alcuni casi, un gestore potrebbe decidere che l'evento è stato completamente elaborato e non è necessario eseguire un'ulteriore propagazione
- Il bubbling può essere interrotto invocando il metodo stopPropagation() sull'oggetto evento
- Regola generale: non interrompere la propagazione a meno che non si abbia una buona ragione. Potrebbe essere necessario catturare gli eventi su un antenato in futuro!

STOPPING EVENT BUBBLING: EXAMPLE

```
<body>
  <section onclick="console.log('section');">
    <article>
      <p onclick="console.log('p');">Bubbling</p>
    </article>
  </section>
  <script>
    let art = document.querySelector("article");
    art.addEventListener("click", handler);
    function handler(event){
      console.log(this.tagName);
      event.stopPropagation();
    }
  </script>
</body>
```



DISPATCHING EVENTS

- Da JavaScript è anche possibile generare eventi
- La proprietà dell'evento `isTrusted` è impostata su `false` per gli eventi «artificiali»

```
<input type="button" value="Click me">
<script>
  function handler(event){
    console.log(` ${event.type} event. Trusted: ${event.isTrusted}`);
  }
  let input = document.querySelector("input");
  input.onclick = handler;

  let click = new Event("click");
  input.dispatchEvent(click); //isTrusted is false for events generated via JS
</script>
```

JAVASCRIPT: EXECUTION ORDER

Man mano che i nostri script diventano più complessi, interagiscono con il DOM e generano dinamicamente gestori di eventi, dobbiamo essere consapevoli di come viene eseguito il codice JavaScript quando viene caricata una pagina web.

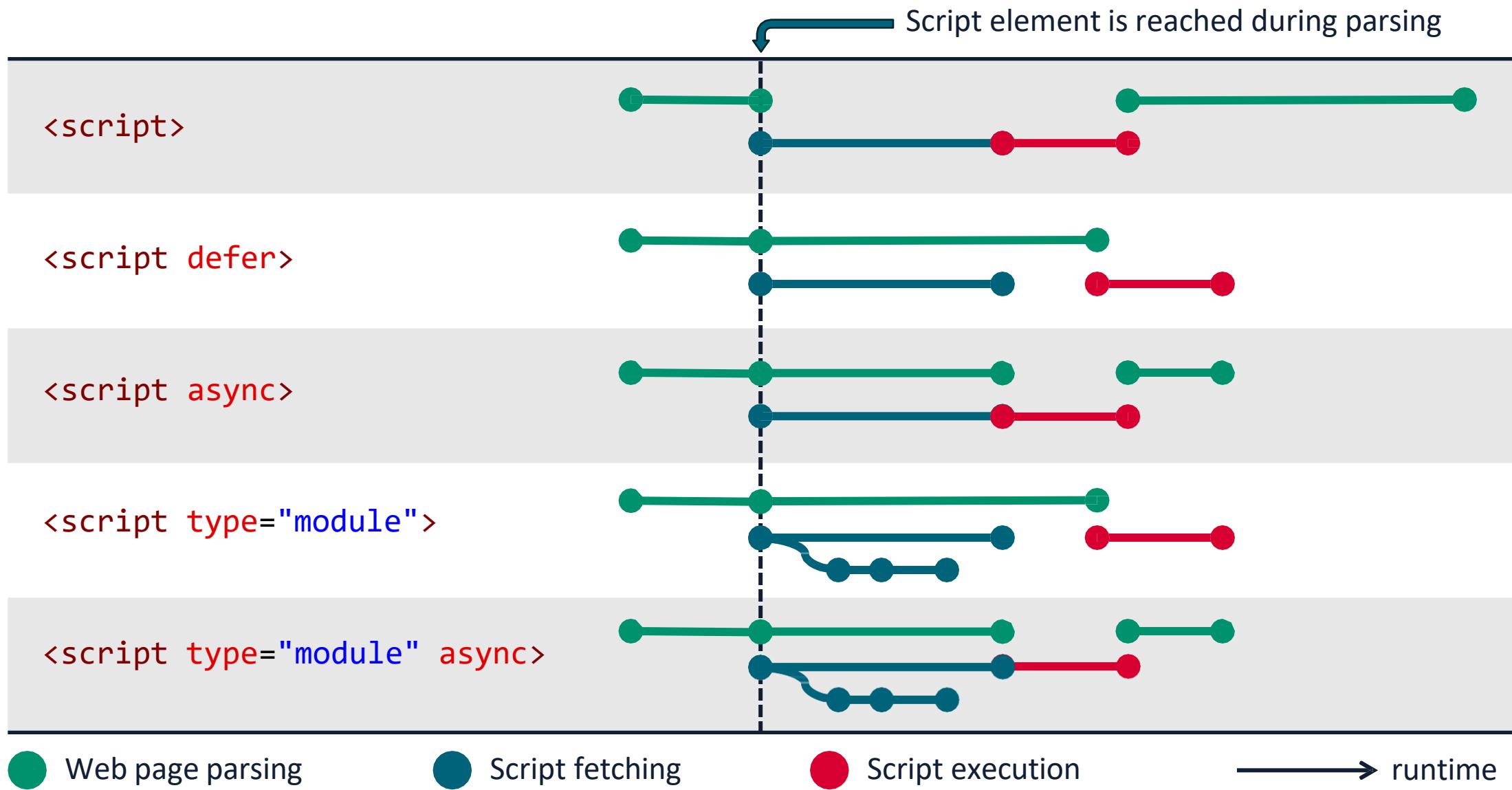
In genere, gli script vengono recuperati ed eseguiti nell'ordine in cui appaiono, mentre l'analisi della pagina Web viene messa in pausa

Gli script esterni con l'attributo defer vengono scaricati in parallelo ed eseguiti al termine dell'analisi della pagina

Gli script esterni con l'attributo async vengono scaricati in parallelo ed eseguiti non appena vengono scaricati

Gli script del modulo rinviano per impostazione predefinita e possono essere dichiarati come asincroni

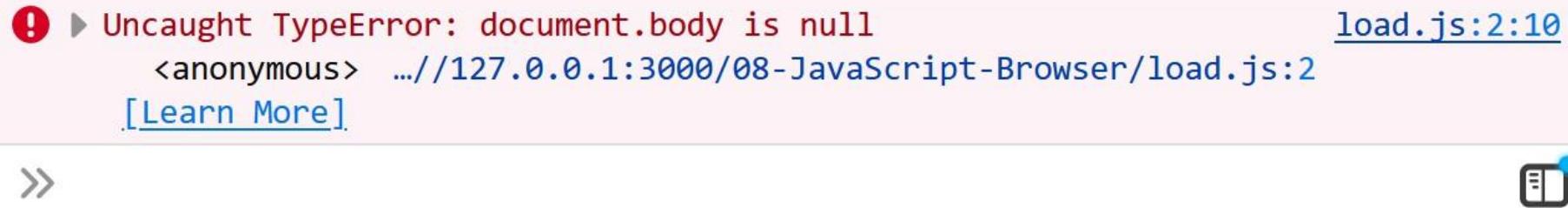
JAVASCRIPT: EXECUTION ORDER



EXECUTION ORDER MATTERS: EXAMPLES

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Events</title>
    <script src="load.js"></script>
  </head>
  <body>
    <h1>JavaScript</h1>
  </body>
</html>
```

```
//load.js file
let h1 = document.body.querySelector("h1");
console.log(h1.innerText);
```



EXECUTION ORDER MATTERS: EXAMPLES

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Events</title>
    <script defer src="load.js"/>
  </head>
  <body>
    <h1>JavaScript</h1>
  </body>
</html>
```

```
//load.js file
let h1 = document.body.querySelector("h1");
console.log(h1.innerText);
```

JavaScript

[load.js:3:9](#)

»



EXECUTION ORDER MATTERS: EXAMPLES

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Events</title>
    <script async src="load.js"/>
  </head>
  <body>
    <h1>JavaScript</h1>
  </body>
</html>
```

```
//load.js file
let h1 = document.body.querySelector("h1");
console.log(h1.innerText);
```

We can't really know for sure. It depends on how much time it takes to fetch the script!

EXECUTION ORDER MATTERS: EXAMPLES

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Events</title>
    <script src="load.js"></script>
  </head>
  <body>
    <h1>JavaScript</h1>
  </body>
</html>
```

```
//load.js file
document.addEventListener("DOMContentLoaded",
() => {
  let h1 = document.body.querySelector("h1");
  console.log(h1.innerText);
});
```

JavaScript

[load.js:3:9](#)

»



REFERENCES

- **The Modern JavaScript Tutorial**

Freely available at <https://javascript.info/> or on [GitHub](#)

Part 2: Document (1.1 to 1.7), Introduction to events, UI events, Document and resource loading

- **Eloquent JavaScript (3rd edition)**

By Marijn Haverbeke

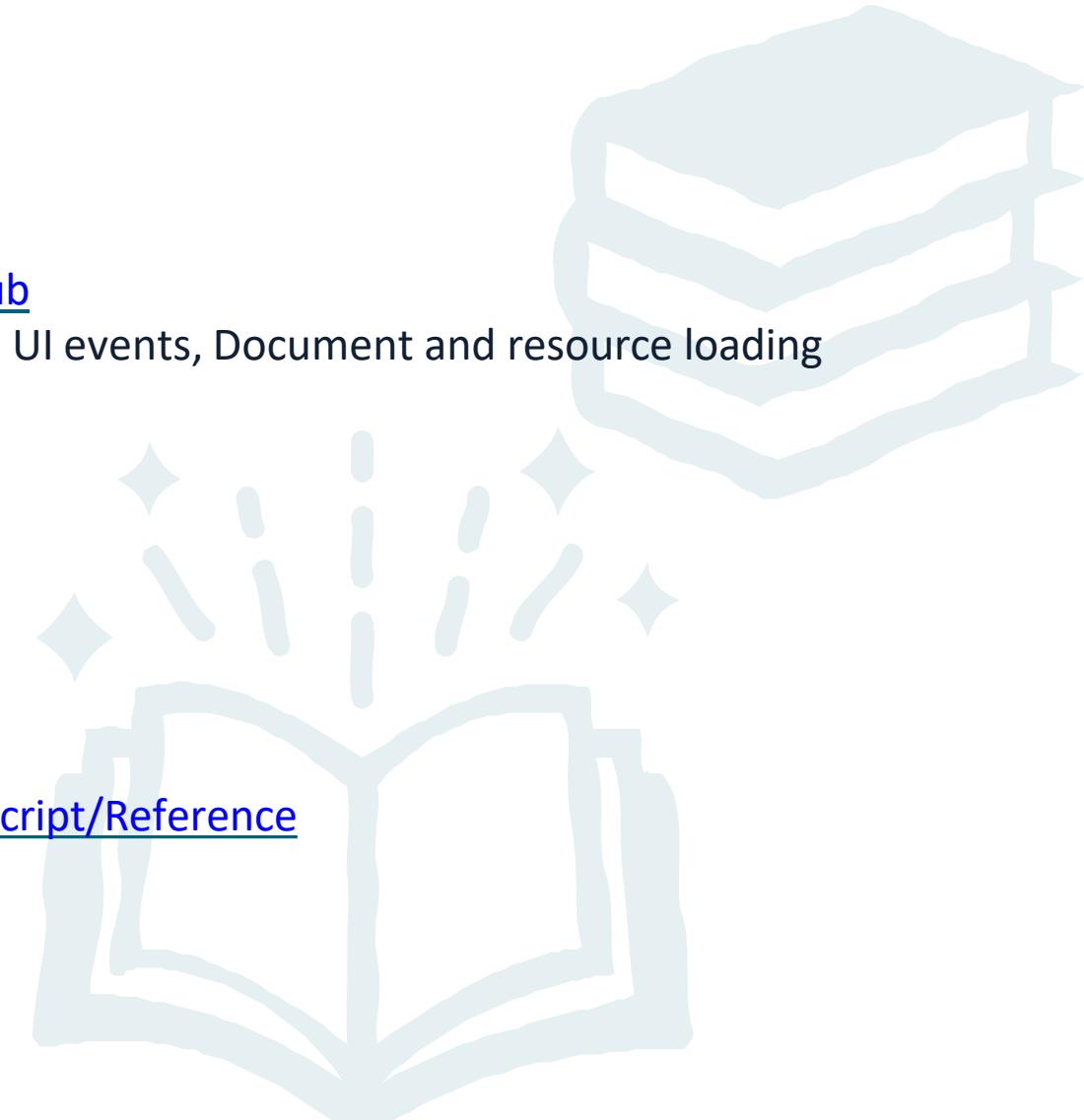
Freely available at <https://eloquentjavascript.net/>

Chapters 13 to 15

- **JavaScript Reference**

MDN web docs

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>



JAVASCRIPT: ASYNCHRONISM AND NETWORK REQUESTS

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



PREVIOUSLY, ON WEB TECHNOLOGIES

So far we've learned the core concepts of JavaScript, and the functionalities available within the **JavaScript Browser Environment**

Today, we'll complete our overview of JavaScript, and we'll see:

- **Browser data storage (Cookies, Local/Session storage, IndexedDB)**
- **Asynchronous Programming constructs**
- **Network Requests**

BROWSER DATA STORAGE

BROWSER STORAGE APIs

Modern web browsers provide a number of APIs that can be used via JavaScript to **store** and **retrieve application data**

- **Cookies**
- **Local/Session Storage**
- **IndexedDB**

COOKIES

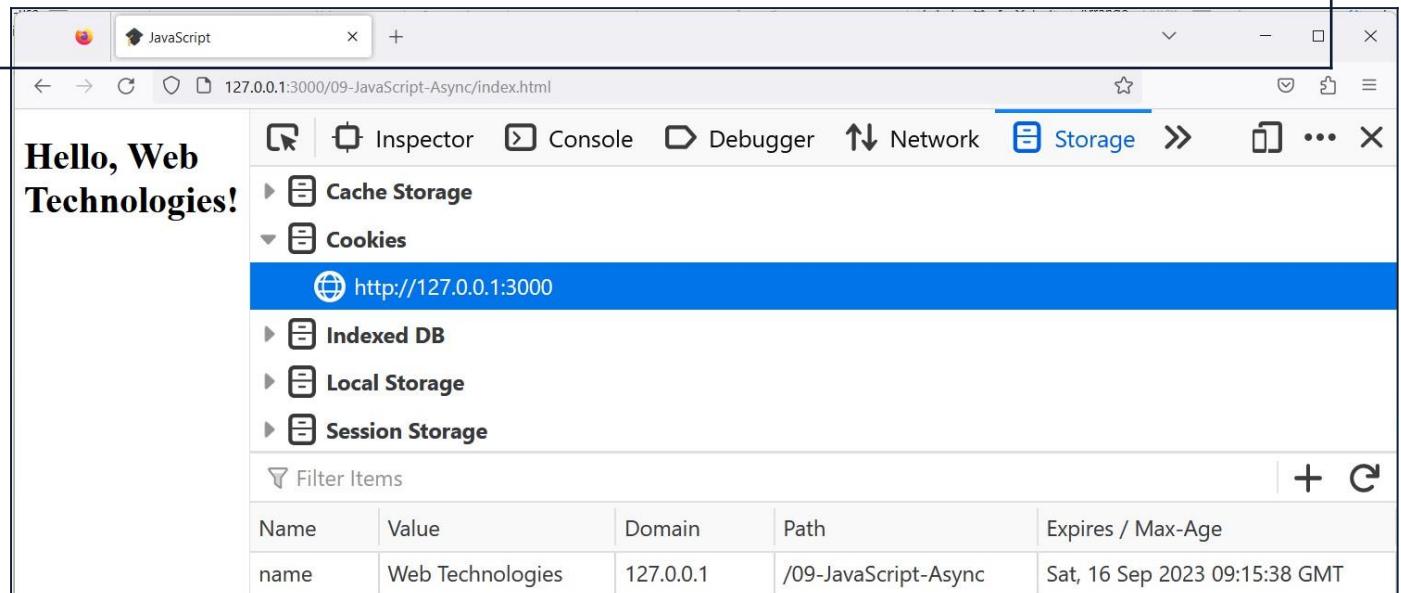
- I cookie sono piccole stringhe di dati
- Parte del protocollo HTTP, è utilizzato per «superare» la sua statelessness
- In genere vengono impostati in una risposta HTTP, con l'intestazione Set-Cookie
- I browser li memorizzano e li inviano in tutte le richieste successive allo stesso dominio, utilizzando l'intestazione Cookie delle richieste HTTP

COOKIES IN JAVASCRIPT

- I cookie per il sito web corrente sono accessibili tramite il `document.cookie`.
- Il valore di `document.cookie` è una stringa di coppie nome=valore, delimitata da «;».
- Ogni coppia è un cookie separato.

COOKIES: EXAMPLE

```
<h1>Hello!</h1>
<script>
  if(document.cookie.indexOf("name=") === -1){ //no cookie called "name" found
    let name = prompt("Please, state your name:");
    document.cookie = `name=${name}; max-age=10` //expires in 10 seconds
  }
  let name = document.cookie.replace("name=", "");
  document.querySelector("h1").innerHTML = `Hello, ${name}!`;
</script>
```



WEB STORAGE: LOCALSTORAGE

La gestione dei cookie tramite JavaScript non è molto semplice.

Soprattutto quando abbiamo a che fare con più cookie, dobbiamo gestire stringhe, dividerle, usare espressioni regolari...

L'oggetto localStorage fornisce modi semplici per memorizzare coppie chiave/valore nei browser Web:

I dati non vengono inviati con ogni richiesta (possiamo memorizzare molti più dati)

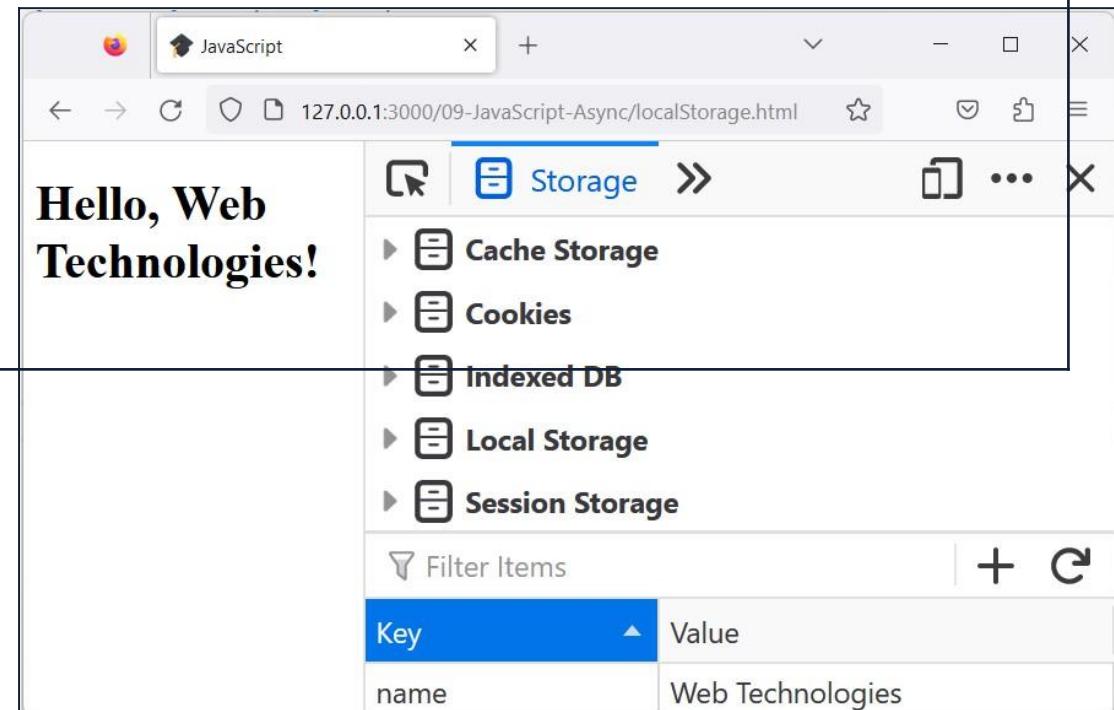
Abbastanza simile a una Map

Può memorizzare solo valori stringa

Gli oggetti in localStorage diversi per ogni dominio/protocollo/porta

LOCALSTORAGE: EXAMPLE

```
<h1>Hello!</h1>
<script>
  if(localStorage.getItem("name") === null){ //no "name" stored
    let name = prompt("Please, state your name:");
    localStorage.setItem("name", name);
  }
  let name = localStorage.name;
  document
    .querySelector("h1")
    .innerHTML = `Hello, ${name}!`;
</script>
```



WEB STORAGE: SESSIONSTORAGE

- I dati di **localStorage** sopravvivono anche a un riavvio completo del browser
- **sessionStorage** fornisce le stesse funzionalità di **localStorage**, ma viene utilizzato molto più raramente
- **sessionStorage** esiste solo all'interno della scheda del browser corrente
- I dati sopravvivono a un aggiornamento della pagina, ma non alla chiusura/riapertura della scheda.

IndexedDB

IndexedDB è un database integrato in un browser

Supporta diversi tipi di chiavi, transazioni, query di intervalli di chiavi

Può memorizzare ancora più dati rispetto a localStorage

Inutilmente potente per le tradizionali applicazioni Web client-server,
per lo più destinate alle applicazioni offline

(A)SYNCHRONISM



ASYNCRONISM

In genere, il codice JavaScript viene eseguito in modo sincrono
Ogni istruzione attende il completamento delle precedenti
Metodo di programmazione intuitivo, ma presenta alcuni inconvenienti
Il completamento di alcune istruzioni potrebbe richiedere del tempo
In attesa dei prompt dell'utente
In attesa del completamento di una richiesta di rete
in attesa che un calcolo complesso finisca
Le istruzioni importanti successive potrebbero essere bloccate in attesa di esse
L'asincronismo può aiutare a risolvere questi problemi!

SYNCHRONOUS STYLE

```
function firstOperation(value){  
    return 1 + value;  
}  
function secondOperation(value){  
    return 2 + value;  
}  
function thirdOperation(value){  
    return 3 + value;  
}  
  
function setupPage(){  
    let result = 0;  
    result = firstOperation(result);  
    result = secondOperation(result);  
    result = thirdOperation(result);  
    console.log(`Result is ${result}`);  
}
```

CALLBACKS

I callback sono funzioni passate come argomenti ad altre funzioni, con l'aspettativa che vengano richiamate al momento opportuno
Stai pensando ai gestori di eventi? Beh, sono davvero dei callback!

ASYNCHRONOUS STYLE: CALLBACKS

```
function firstOperation(value, cb){  
    cb(1 + value);  
}  
  
function secondOperation(value, cb){  
    cb(2 + value);  
}  
  
function thirdOperation(value, cb){  
    cb(3 + value);  
}
```



Callback Hell or
Pyramid of Doom!

```
function setupPage(){  
    let result = 0;  
    firstOperation(result, (r1) => {  
        secondOperation(r1, (r2) => {  
            thirdOperation(r2, (r3) => {  
                console.log(`Result is ${r3}`);  
            });  
        });  
    });  
    console.log("Done");  
}
```

CALLBACKS: LIMITATIONS

- Quando si gestiscono molti callback all'interno dei callback, il codice diventa molto disordinato molto velocemente
- Potrebbe anche diventare più complicato. Pensa alla gestione degli errori...
- C'è un motivo per cui si chiama Callback Hell o Pyramid of Doom
- Il vecchio JavaScript gestiva l'asincronismo utilizzando le callback.
- Nel JavaScript moderno, in genere utilizziamo un nuovo costrutto: Promise

PROMISES: PRODUCERS AND CONSUMERS

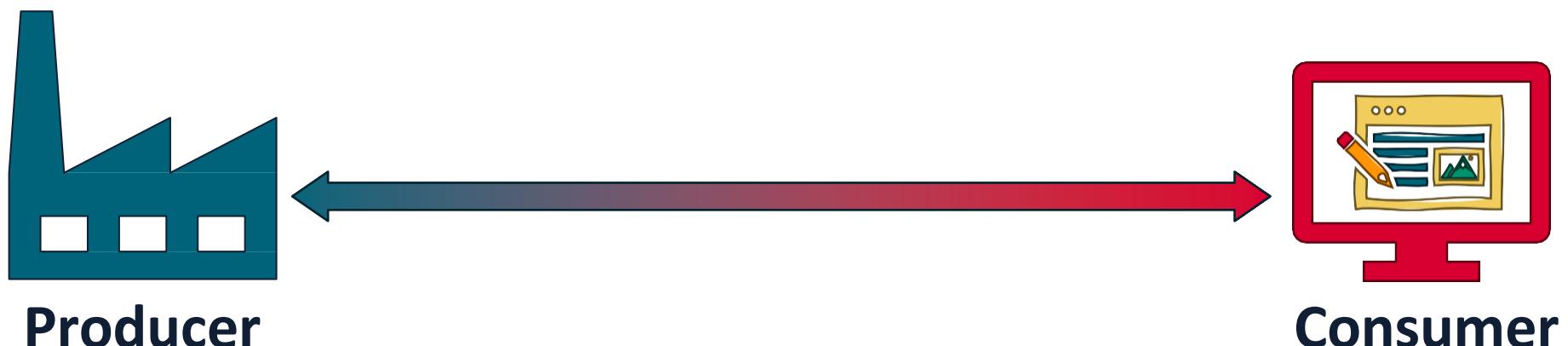
Nella programmazione (JavaScript) capita spesso che ci sia:

Un codice «produttore» che fa qualcosa e richiede un po' di tempo

Attendi gli input dell'utente, recupera i dati di rete, calcola cose complesse...

Un codice «consumer», che ha bisogno dei risultati di un produttore

Le promesse sono un modo per «collegare» produttori e consumatori



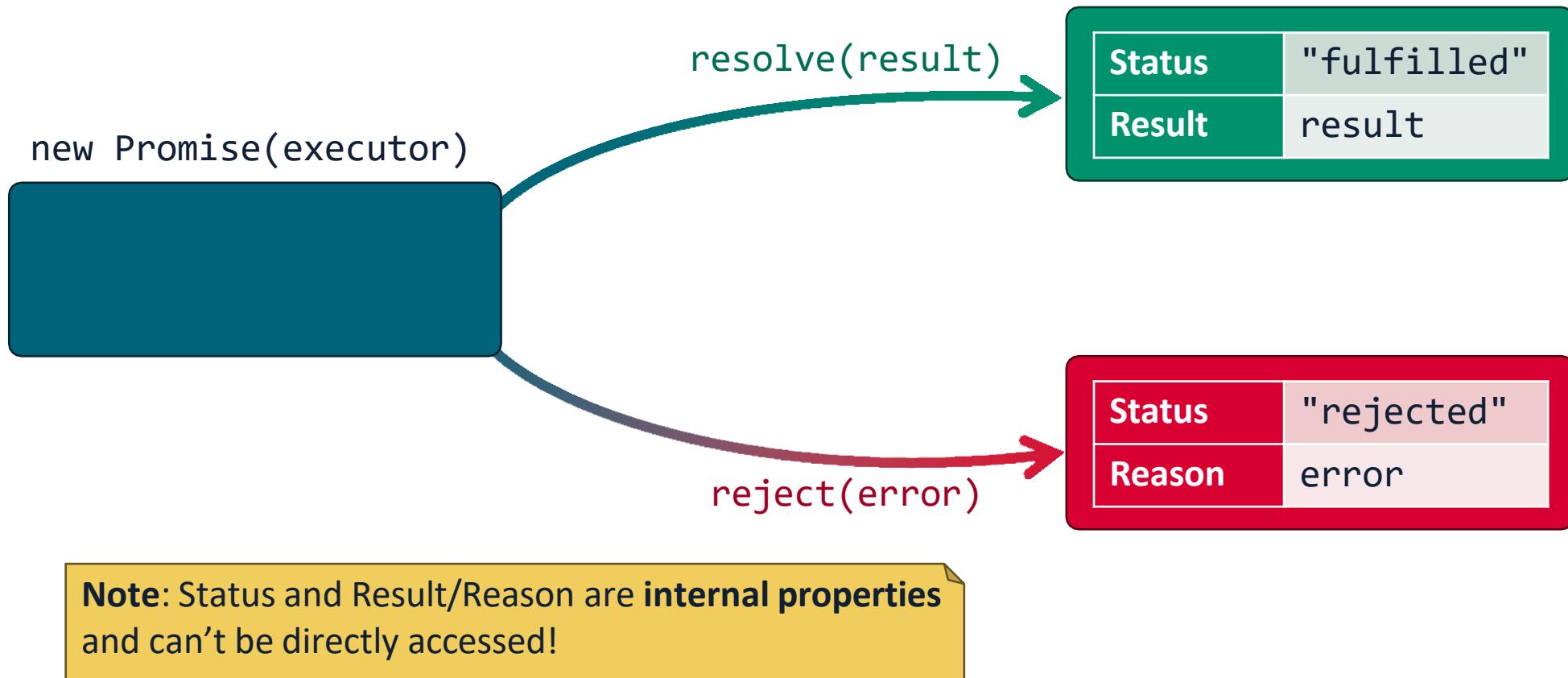
CREATING PROMISES

- Un oggetto Promise può essere creato utilizzando il costruttore appropriato

```
let promise = new Promise(function(resolve, reject){  
    //Executor: producer code here  
});
```

- La funzione data come argomento è l'esecutore
- L'esecutore viene immediatamente richiamato quando viene creata la Promise
- resolve e reject sono callback forniti da JavaScript
- Al termine del codice dell'esecutore, deve chiamare
- resolve(value): se è stato completato con successo, con result value
- reject(error): se si è verificato un errore (error è l'oggetto errore)

PROMISES: LIFECYCLE



PROMISES: EXAMPLE

```
<script>
let promise = new Promise(function(resolve, reject){
  let number = Math.random();
  setTimeout( () => { // setTimeout takes as input a callback and a time in ms
    if(number > 0.5){
      resolve(number);
    } else {
      reject(new Error("Number too small"));
    }
  }, 1000);
});

console.log(promise); // Promise { <state>: "pending" }
</script>
```

PROMISES: CONSUMER CODE

- Il codice «consumatore» riceve la promessa che alcuni dati saranno resi disponibili in futuro.
- Le azioni da eseguire quando la Promise viene soddisfatta (o rifiutata) possono essere registrate utilizzando metodi specifici offerti dall'oggetto Promise.
- Il più importante di questi metodi è `.then()`

PROMISES: THEN

- The `.then()` method takes as input **two callbacks**

```
promise.then(  
  function(result) { /* called when the promise is fulfilled */ },  
  function(error) { /* called when the promise is rejected */ }  
)
```

- È anche possibile registrare una sola funzione da eseguire quando la promessa viene soddisfatta

```
promise.then(  
  (result) => { /* called when the promise is fulfilled */ }  
)
```

PROMISES: CATCH

- Quando siamo interessati solo alla gestione degli scenari di errore, possiamo passare null come primo argomento:

```
promise.then(  
  null,  
  function(error)  { /* called when the promise is rejected */ }  
)
```

- Oppure possiamo usare il metodo .catch(), che è equivalente

```
promise.catch(  
  (error) => { /* called when the promise is rejected */ }  
)
```

PROMISES: FINALLY

- Proprio come i blocchi try/catch, le promise dispongono anche di un metodo .finally()
- Questo metodo accetta un callback senza input
- Viene invocato non appena la Promise è risolta: sia essa risolta o rifiutata
- È destinato ad essere utilizzato per eseguire operazioni di pulizia e altre operazioni che devono essere eseguite indipendentemente dallo stato della Promise.

PROMISES: EXAMPLE

```
let promise = new Promise(function(resolve, reject){
  let number = Math.random();
  setTimeout( () => { //setTimeout takes as input a callback and a time in ms
    if(number > 0.5){
      resolve(number);
    } else {
      reject(new Error(`Number too small: ${number}`));
    }
  }, 1000);
});
console.log("Here");
promise.finally( () => {console.log("Promise settled")})
promise.then(
  (result) => {console.log(`Done: ${result}`)},
  (error)  => {console.log(error.message)})
);
console.log("There");
```

// Console output
Here
There
Promise settled
Done: 0.6 (or "Number too small")

PROMISES: CHAINING

- What if we need to perform a sequence of asynchronous steps?

ASYNCHRONOUS STYLE: CALLBACKS

```
function firstOperation(value, cb){  
    cb(1 + value);  
}  
function secondOperation(value, cb){  
    cb(2 + value);  
}  
function thirdOperation(value, cb){  
    cb(3 + value);  
}
```



```
function setupPage(){  
    let result = 0;  
    firstOperation(result, (r1) => {  
        secondOperation(r1, (r2) => {  
            thirdOperation(r2, (r3) => {  
                console.log(`Result is ${r3}`);  
            });  
        });  
    });  
    console.log("Done");  
}
```

Callback Hell or Pyramid of Doom!

Luigi Libero Lucio Starace, Ph.D. - University of Naples Federico II - Web Technologies Course - Lecture 07 - JavaScript: Browser Environment

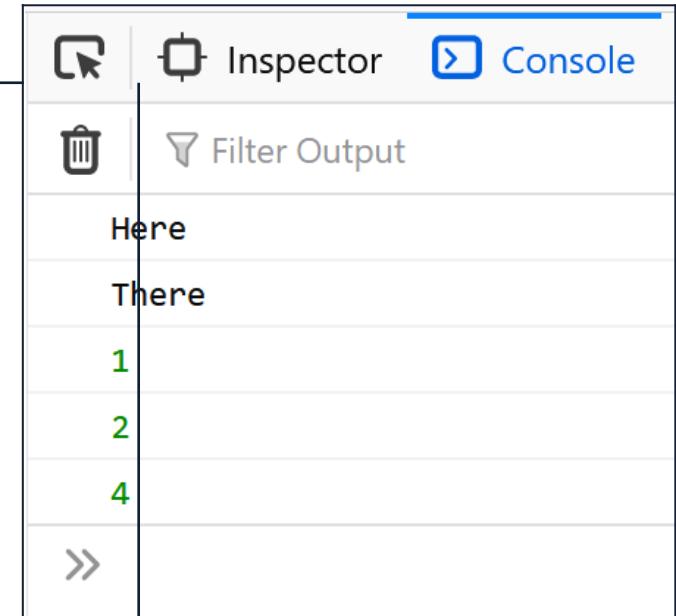
16

- Il concatenamento delle promesse è il modo per gestire tali scenari

PROMISE CHAINING

```
<script>
let promise = new Promise(function(resolve, reject){
  let number = Math.random();
  setTimeout( () => {
    resolve(1);
  }, 1000);
});

console.log("Here");
promise
  .then( (result) => {console.log(result); return result*2;})
  .then( (result) => {console.log(result); return result*2;})
  .then( (result) => {console.log(result); return result*2;})
console.log("There");
</script>
```



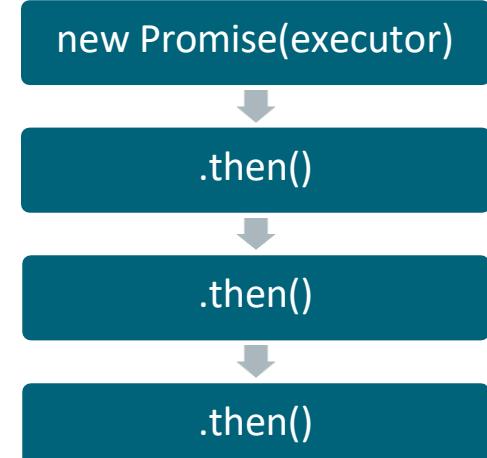
PROMISE CHAINING: HOW DOES IT WORK?

- How does that even work?
- La Promise originale viene risolta una sola volta, con un risultato di 1
- Bene, ogni invocazione `.then()` restituisce effettivamente un nuovo oggetto Promise.
- Queste nuove promesse rappresentano il completamento dell'handler stesso
- Quando un gestore restituisce un valore, questo diventa il risultato del nuovo oggetto Promise

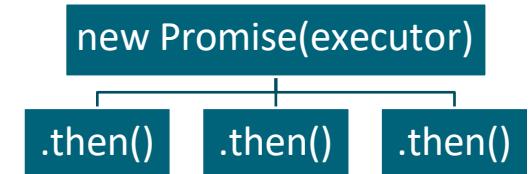
PROMISE CHAINING: NEWBIE MISTAKES

- I seguenti frammenti sono molto diversi!

```
promise //prints 1, then 2, then 4
  .then( (r) => {console.log(r); return r*2;})
  .then( (r) => {console.log(r); return r*2;})
  .then( (r) => {console.log(r); return r*2;});
```

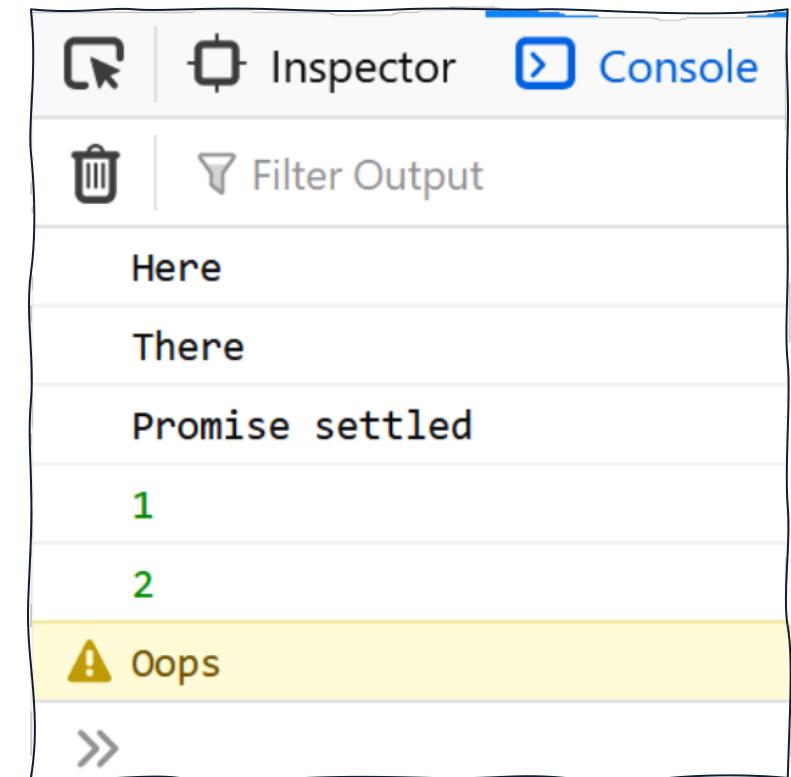


```
//prints 1, 1, 1
promise.then( (r) => {console.log(r); return r*2;} );
promise.then( (r) => {console.log(r); return r*2;} );
promise.then( (r) => {console.log(r); return r*2;} );
```



PROMISE CHAINING: EXAMPLE

```
promise.finally(() => {
  console.log("Promise settled");
}).then((r) => {
  console.log(r);
  return r * 2;
}).then((r) => {
  console.log(r);
  throw new Error("Oops"); //implicit reject()
  return r * 2;
}).then((r) => {
  console.log(r);
  return r * 2;
}).catch((error) => {
  console.warn(error.message);
});
```



PROMISE API: PROMISE.ALL()

- Il metodo statico Promise.all() prende come input un array di Promise e restituisce una nuova Promise che rappresenta il completamento di tutte le Promise di input
- Il risultato è un array, ogni input Promise contribuisce e l'elemento
- Se una Promise rifiuta, Promise.all rifiuta immediatamente

```
let prom = Promise.all([
  new Promise( resolve => setTimeout( () => resolve(3) ), 3000),
  new Promise( resolve => setTimeout( () => resolve(2) ), 2000),
  new Promise( resolve => setTimeout( () => resolve(1) ), 1000),
]).then((result) => {
  console.log(result); //Array(3) [3, 2, 1]
});
```

PROMISE API: PROMISE.ALLSETTLED()

- **Promise.all rifiuta se una qualsiasi delle promesse di input viene rifiutata (tutto o niente)**
- **Promise.allSettled() è simile: attende che tutte le promesse di input vengano risolte, ma restituisce come risultato un array di oggetti che rappresentano lo stato di ogni promessa di input.**

```
let p = Promise.allSettled([
  new Promise((res, rej) => { setTimeout(() => rej(new Error("Oops")), 1000)),
  new Promise((res, rej) => { setTimeout(() => res(1), 2000)}),
]).then(result) => {
  console.log(result);
}).catch(err) => {
  console.warn(err.message); //Not executed
});
```



```
▼ Array [ ..., ... ]
  ▼ 0: Object { status: "rejected", reason: Error }
    ► reason: Error: Oops
      status: "rejected"
    ► <prototype>: Object { ... }
  ▼ 1: Object { status: "fulfilled", value: 1 }
    status: "fulfilled"
    value: 1
    ► <prototype>: Object { ... }
  length: 2
  ► <prototype>: Array []
```

PROMISE API: PROMISE.RACE()

- Simile a Promise.all(), ma viene liquidato non appena una delle promesse di input viene risolta e ottiene il suo risultato (o errore)

```
let p = Promise.race([
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Oops")), 2000)}),
  new Promise((res, rej) => { setTimeout( () => res(1), 1000)}),
]).then((result) => {
  console.log(result); //1
});

let p2 = Promise.race([
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Oops")), 1000)}),
  new Promise((res, rej) => { setTimeout( () => res(1), 2000)}),
]).catch((err) => {
  console.warn(err.message) //Oops
});
```

PROMISE API: PROMISE.ANY()

- Simile a Promise.race(), ma attende la prima Promise risolta

```
let p = Promise.any([
  new Promise((res, rej) => { setTimeout(() => res(3), 3000)},
  new Promise((res, rej) => { setTimeout(() => rej(new Error("Oops")), 1000)}),
  new Promise((res, rej) => { setTimeout(() => res(1), 2000)}),
]).then((result) => {
  console.log(result); //1
}).catch((err) => {
  console.warn(err.message); //Not executed: Promise.any is resolved
});
```

PROMISE API: PROMISE.ANY()

- Se tutte le promesse vengono rifiutate, Promise.any rifiuta con un AggregateError contenente tutti gli errori delle promesse di input

```
let p = Promise.any([
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Woah")), 2000)},
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Oops")), 1000)},
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Ouch")), 1000)},
]).then((result) => {
  console.log(result); //not executed, Promise.any rejected
}).catch((err) =>{
  console.warn(err.message); //No promise in Promise.any was resolved
  console.warn(err.errors[0].message); //Woah
  console.warn(err.errors[1].message); //Oops
  console.warn(err.errors[2].message); //Ouch
});
```

PROMISE API: PROMISE.RESOLVE / REJECT

- `Promise.resolve(result)` e `Promise.reject(error)` creano Promesse già liquidate (risp. Resolved o Rejected)

```
let p = Promise.resolve(1); //same as p = new Promise( resolve => resolve(1) )
p.then( result => console.log(result) );
```

- A volte vengono utilizzati per la compatibilità (ad esempio: quando ci si aspetta che una funzione restituisca una Promise)

JAVASCRIPT: ASYNC/AWAIT

Il JavaScript moderno include una sintassi speciale per lavorare con le promesse
Si compone di due parole chiave: `async` e `await`

La parola chiave `async` può essere inserita prima di una funzione per garantire che la funzione restituisca sempre una promessa

Qualsiasi altro valore restituito da una funzione asincrona viene encapsulato in modo implicito in una promessa risolta

Gli errori generati vengono racchiusi in una promessa rifiutata

```
async function f(){
  return new Promise(...);
}
```

```
async function g(){
  return 1; //returns Promise.resolve(1)
}
```

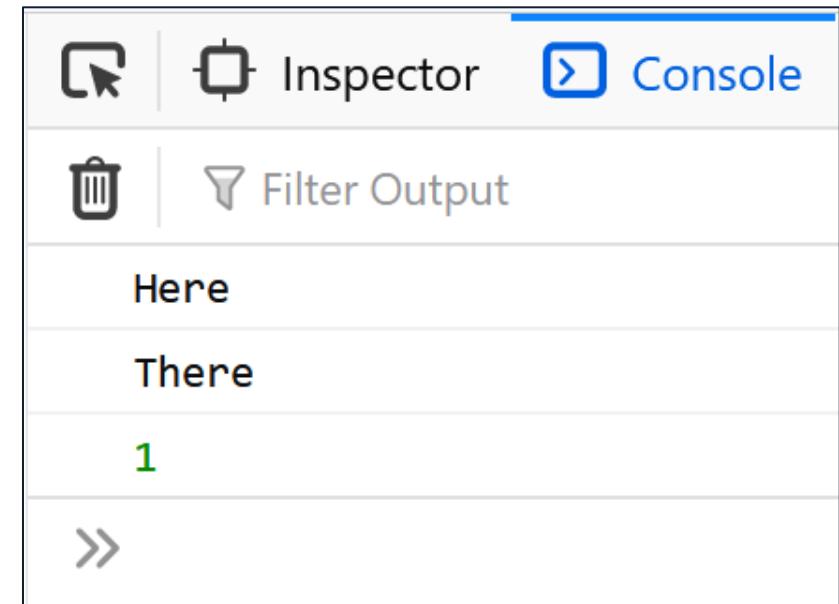
JAVASCRIPT: ASYNC

```
console.log("Here");

async function f(){
  //throw new Error("Ouch");
  return 1;
}

f().then(console.log).catch(console.warn);

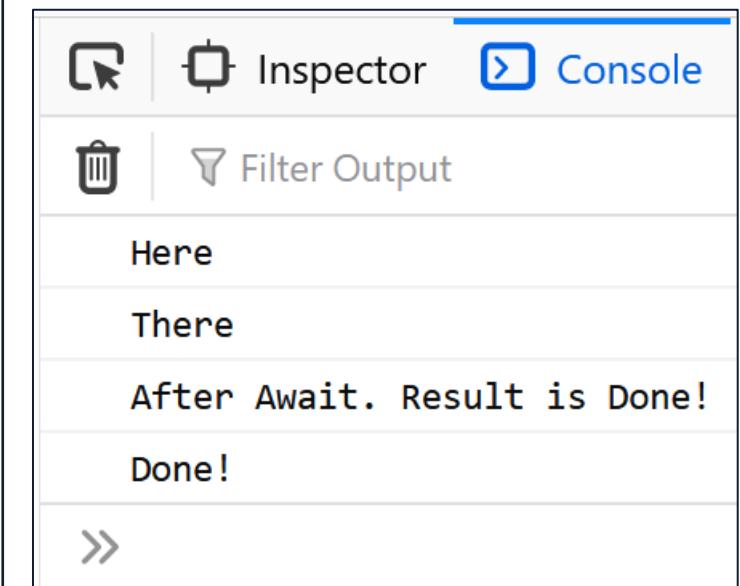
console.log("There");
```



JAVASCRIPT: AWAIT

- La parola chiave await può essere utilizzata solo nelle funzioni asincrone
- Sospende l'esecuzione del codice fino a quando non viene risolta una Promise

```
console.log("Here");
async function f(){
  let promise = new Promise(function(res, rej){
    setTimeout(() => {res("Done!")}, 2000);
  });
  let result = await promise; //execution pauses here
  console.log(`After Await. Result is ${result}`);
  return result;
}
f().then(console.log);
console.log("There");
```



NETWORK REQUESTS

NETWORK REQUESTS

- JavaScript can also fetch data from the internet
- In old JavaScript, programmers used XMLHttpRequest objects
 - Callbacks were used to handle status events

```
<h1>Cat Facts</h1>
<button id="btn">Click Here To Load a Cat Fact</button> <p id="fact"></p>
<script>
  let btn = document.getElementById("btn");
  btn.onclick = () => {
    let req = new XMLHttpRequest(); req.onload = handleFact;
    req.open("GET", "https://catfact.ninja/fact"); req.send();
    function handleFact(result) {
      let fact = JSON.parse(this.responseText);
      document.getElementById("fact").innerHTML = fact.fact;
    }
  }
</script>
```

NETWORK REQUESTS: THE FETCH API

- Il modo moderno di eseguire richieste HTTP all'interno di JavaScript è `fetch()`
- `fetch()` prende come input un URL e un array opzionale di opzioni

```
let promise = fetch("url", [options]);
```
- Le opzioni possono essere utilizzate per specificare il metodo HTTP, le intestazioni delle richieste, ecc...
- Viene restituito un oggetto Promise, che si risolve in un oggetto Response

FETCH API: WORKING WITH RESPONSES

- La Promise restituita da fetch viene risolta non appena vengono ricevute le intestazioni di risposta
- In questa fase, possiamo controllare lo stato HTTP, le intestazioni, ma il corpo della risposta potrebbe non essere ancora disponibile

FETCH API: WORKING WITH RESPONSES

```
async function f(){
  let response = await fetch("./example.json"); //there is no such file
  //get one specific header
  console.log(response.headers.get('Content-Type')); // application/json
  //iterate over all response headers
  for(let [key, value] of response.headers){
    console.log(`Header ${key}: ${value}`);
  }
  let status = response.status;
  console.log(status);
  if(response.ok){
    console.log("Request was successful");
  } else {
    console.log("Some error occurred");
  }
}
```

FETCH API: GETTING THE RESPONSE BODY

- Quando una promessa di recupero viene risolta, il corpo potrebbe non essere ancora presente
- La risposta fornisce diversi metodi basati su promesse per accedere al corpo, in vari formati. Ad esempio, `.json()` analizza il corpo come JSON non appena diventa disponibile.

```
async function f(){
  let response = await fetch("https://catfact.ninja/fact");
  if(response.ok){
    let fact = await response.json();
    document.getElementById("fact").innerHTML = fact.fact;
  } else {
    console.log("Some error occurred");
  }
}
```

FETCH API: GETTING THE RESPONSE BODY

- Some other methods to access the body are:

.text()	Read the response body and return it as text
.json()	Parse the response body as JSON
.blob()	Return a Blob (Binary data with type)
.arrayBuffer()	Return an ArrayBuffer (low level repr. of binary data)
.formData()	Return a FormData object (represents data submitted via forms)

- **Importante: possiamo scegliere un solo metodo di lettura del corpo! Se hai già chiamato response.text(), response.json() non funzionerà perché il contenuto del corpo è già stato consumato!**

FETCH API: BLOB EXAMPLE

```
<input id="msg" placeholder="Insert your message here" type="text">
<button id="btn">Generate QR</button><img id="img">
<script>
document.getElementById("btn").onclick = () => {
  let msg = document.getElementById("msg").value;
  let url = `https://image-charts.com/chart?chs=200x200&cht=qr&chl=${msg}&choe=UTF-8`;
  console.log(url);
  let promise = fetch(url);
  promise.then((response) => {
    return response.blob();
  }).then((blob) => {
    let img = document.getElementById("img");
    img.src = URL.createObjectURL(blob);
  });
}
</script>
```

REFERENCES

- **The Modern JavaScript Tutorial**

Freely available at <https://javascript.info/> or on [GitHub](#)

Part 2: Storing data in the browser (4.1, 4.2)

Part 1: Promises and async/await (11.1 to 11.5, 11.8)

Part 3: Network requests (3.1, 3.8)

- **Eloquent JavaScript (3rd edition)**

By Marijn Haverbeke

Freely available at <https://eloquentjavascript.net/>

Chapters 11, 18

- **JavaScript Reference**

MDN web docs

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>



UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
WEB TECHNOLOGIES – LECTURE 09

WELCOME TO THE SERVER-SIDE OF THE WEB

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



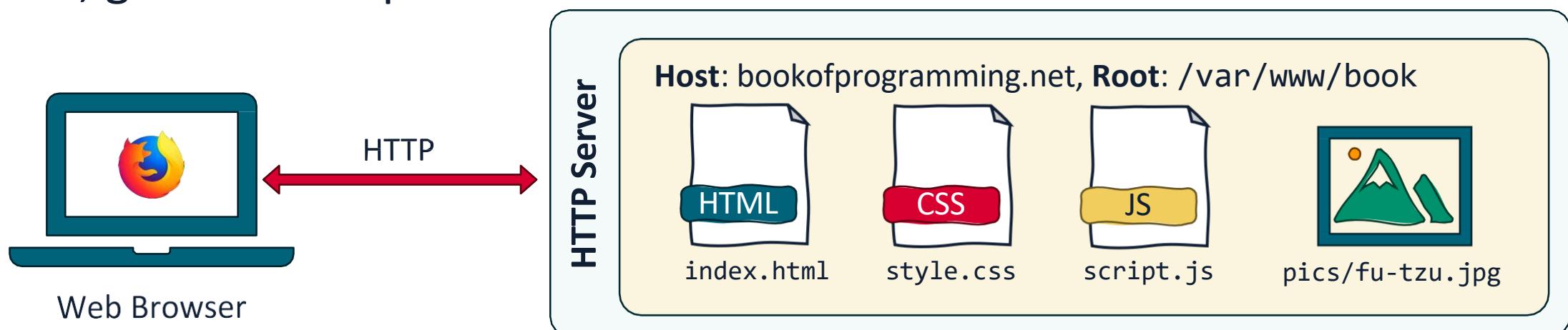
PREVIOUSLY, ON WEB TECHNOLOGIES

So far, we've learned a good deal about the **core web technologies: HTML, CSS, and JavaScript**. Let's contemplate our progress so far:

- We learned to design **modern, beautiful, responsive** web pages.
- We learned to make web pages **dynamic**, using JavaScript:
 - Handling events, dynamically changing the DOM
 - Making async HTTP network requests to fetch data to show in our pages

STATIC WEB APPS

- Le nostre pagine web possono avere una certa dinamicità, grazie a JavaScript
- Questo dinamismo avviene all'interno dei browser web
- Tuttavia, in un certo senso, sono ancora intrinsecamente statici
- Ogni volta che un browser invia una richiesta a una delle nostre pagine web, gli verrà sempre servito lo stesso documento identico



LIMITATIONS OF STATIC WEB APPS

- L'approccio web «statico» che abbiamo visto finora è semplice ed efficace
- Funziona alla grande per le applicazioni web costituite da un numero limitato di pagine, che cambiano abbastanza raramente. È interessato da alcune limitazioni:
- Cosa succede se vogliamo personalizzare una pagina per un utente specifico?
- Cosa succede se il contenuto di una pagina cambia molto frequentemente?
- Cosa succede se il nostro sito web è composto da milioni di pagine?
- Cosa succede se dobbiamo agire in base a una richiesta (ad esempio: salvare un ordine, ...)?

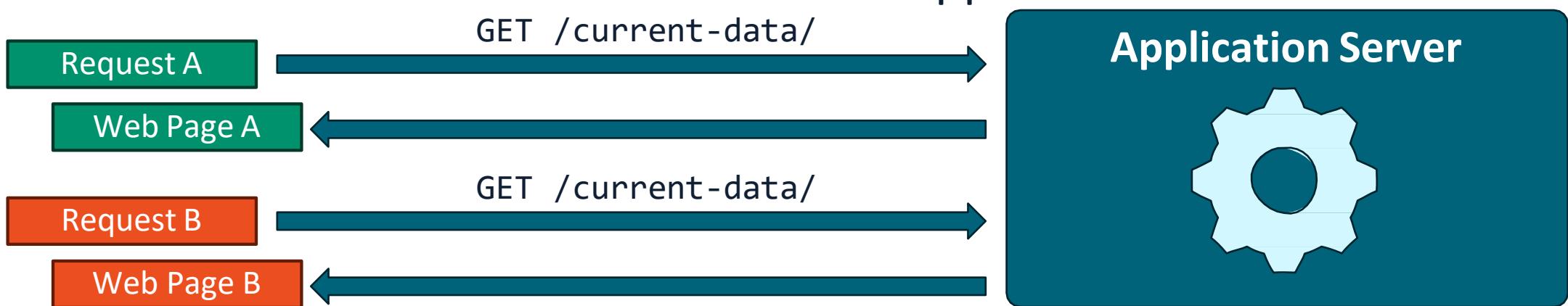
AN EXAMPLE: WIKIPEDIA

- Wikipedia features **~60 million** web pages
 - All with the same structure, same style...
 - ...but different contents
- Let's say the base structure in a Wikipedia page is **~150k chars**
- That's **150kB** of data **repeated** in **every** page
- Multiply that for the 60 million web pages, you get **9 TeraBytes** of repeated data
- What if we need to change the navbar?



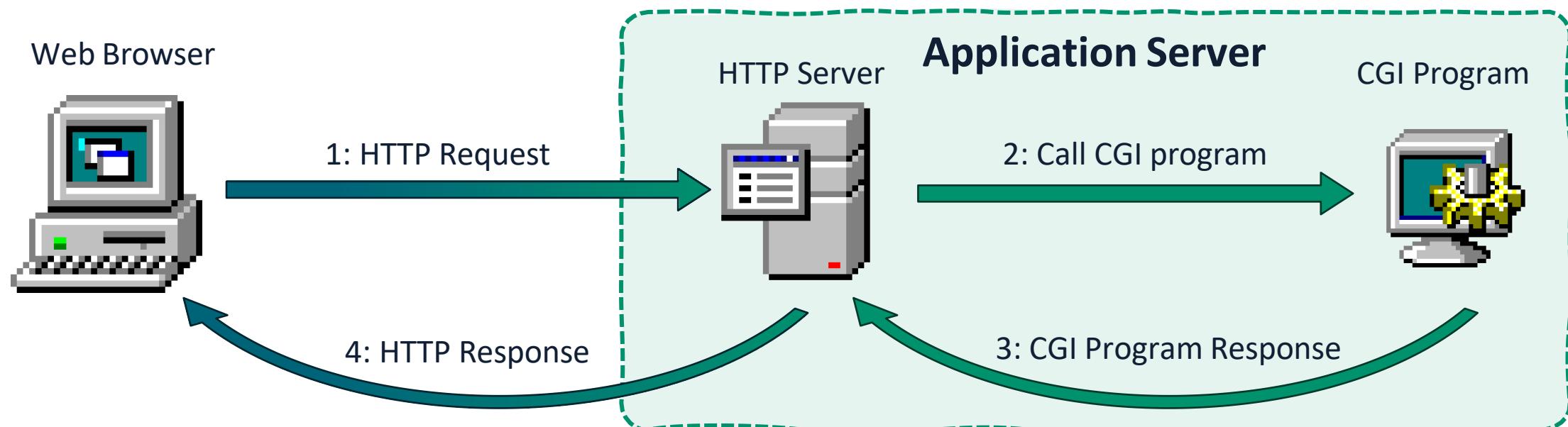
INTRODUCING SERVER-SIDE PROGRAMMING

- Fino ad ora, i server Web servono solo i file dalla radice del documento
- Un server web può fare molto di più!
- Per cominciare, può generare pagine Web al volo
- In genere in base ai parametri ricevuti nella richiesta HTTP
- Questa è l'idea chiave alla base della programmazione lato server
- Il server del browser diventa il server di tutto l'applicativo



SERVER-SIDE PROGRAMMING: CGI

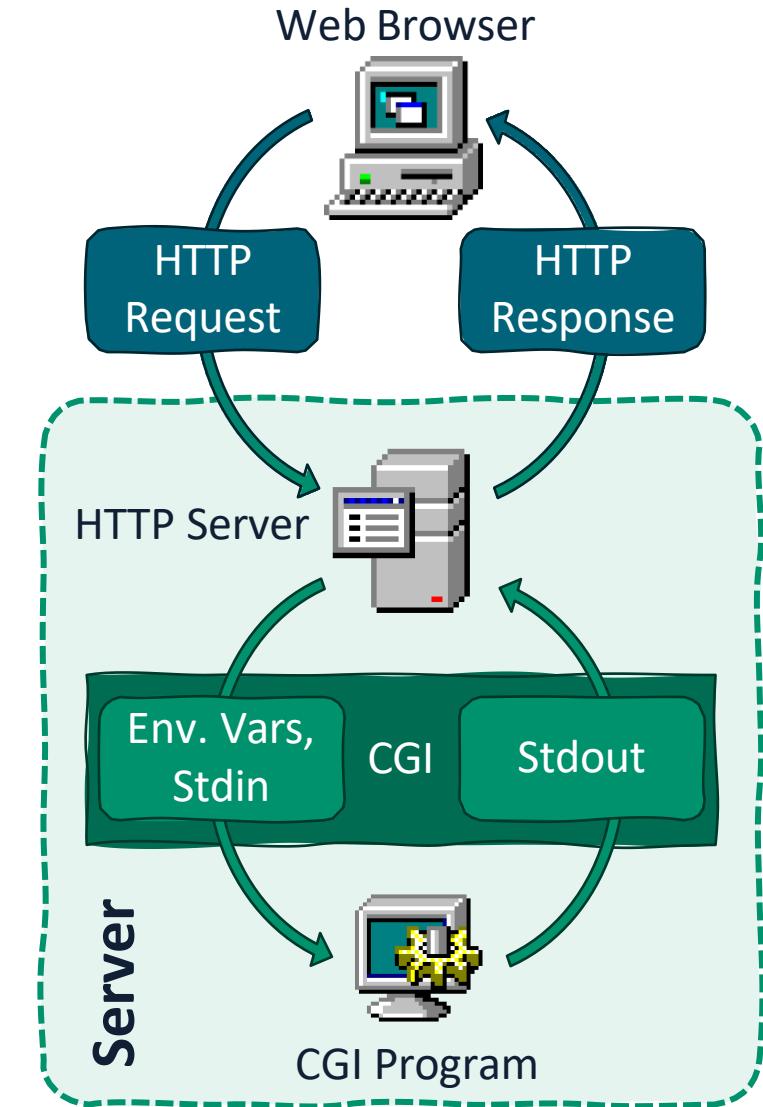
- In the early 1990, before JavaScript even existed, dynamic web pages could be achieved using the **Common Gateway Interface (CGI)**
- CGI is an interface specification allowing web servers to execute external programs to process an HTTP request



THE COMMON GATEWAY INTERFACE

Defines how web servers and CGI-compliant programs communicate

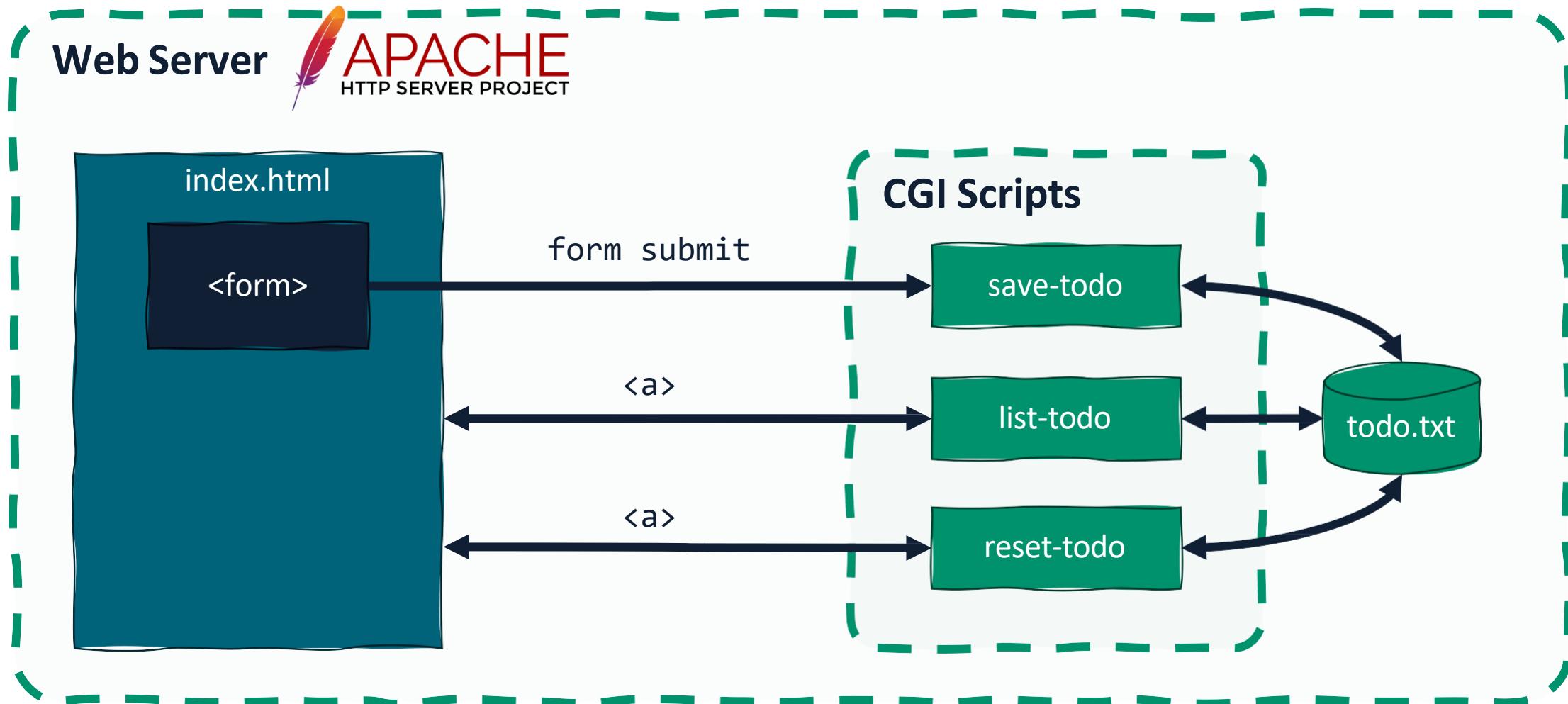
- How a CGI program should access its inputs
 - Request headers and Query String are available via **Environment Variables**
 - Request body is available to the program on the **stdin** stream
- How a web server should build a response
 - The program output on **stdout** is the body of the HTTP response



OUR VERY FIRST CGI SCRIPT

```
#!/bin/bash
echo "Content-type: text/html; charset=utf-8"
echo ""
echo "<!DOCTYPE html>"
echo "<html><head><title>Hello CGI!</title></head><body>"
echo "<h1>Hello CGI!</h1>"
echo "<p>This page was loaded on "
date +"%d/%m/%Y at %H:%M:%S. </p></body>"
```

EXAMPLE: CGI-BASED TODO LIST WEB APP



EXAMPLE: CGI-BASED TO - DO LIST

WITH BASH

- Live demo time!
- Repo: <https://github.com/luistar/cgi-to-do-list-with-bash>
- Will the teacher make it through the first demo of the course?



SERVER-SIDE SCRIPTING

- Scrivere programmi CGI non è un'esperienza molto piacevole
- Scrivere interi documenti HTML su Stdout è complicato
- L'analisi manuale delle stringhe di query e dei corpi delle richieste non è molto divertente
- La generazione di un nuovo processo per ogni richiesta non è molto efficiente in termini di risorse
- Fortunatamente, sono emersi linguaggi di programmazione e framework specializzati
- PHP (acronimo ricorsivo di PHP Hypertext Preprocessor), nel 1995
- ASP (Active Server Pages), il linguaggio di scripting lato server di Microsoft, nel 1997
- JSP (Java Server Pages), originariamente pubblicato da Sun nel 1999
- L'idea era quella di «mescolare» il codice lato server e l'HTML, con uno speciale interprete che analizza il mix per produrre codice HTML «puro»

SERVER-SIDE SCRIPTING: PHP

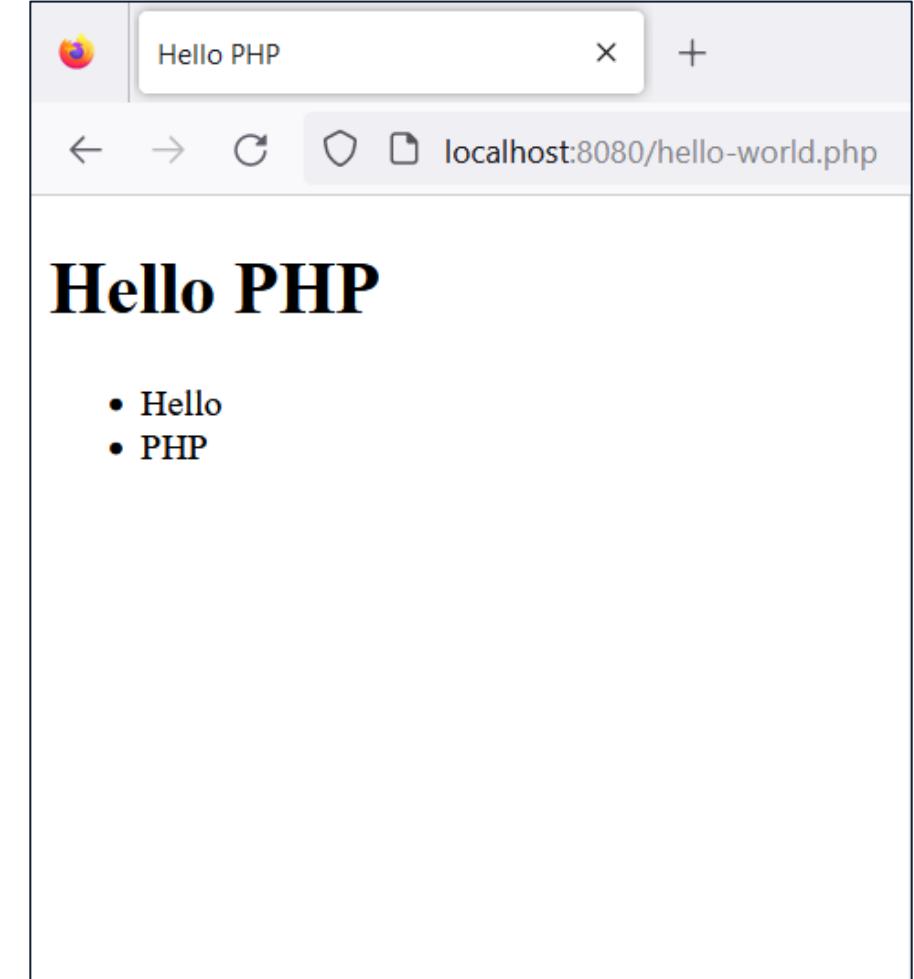
- Delimitatori speciali <?php ... ?> vengono utilizzati per separare il codice e l'HTML

```
<!-- hello-world.php file -->
<!DOCTYPE html>
<html>
  <head>
    <title><?php echo "Hello PHP" ?></title>
  </head>
  <body>
    <h1>Hello PHP</h1>
    <?php $array = ["Hello", "PHP"] ?>
    <ul>
      <?php foreach($array as $item): ?>
        <li><?= $item ?></li>
      <?php endforeach; ?>
    </ul>
  </body>
</html>
```

- Il testo all'esterno dei delimitatori rimane così com'è nell'output
- Il codice all'interno dei delimitatori viene interpretato e l'output risultante viene inserito nel file html
- <?= è una scorciatoia per <?php echo>

SERVER-SIDE SCRIPTING: PHP

```
<!-- hello-world.php file -->
<!DOCTYPE html>
<html>
  <head>
    <title><?php echo "Hello PHP" ?></title>
  </head>
  <body>
    <h1>Hello PHP</h1>
    <?php $array = ["Hello", "PHP"] ?>
    <ul>
      <?php foreach($array as $item): ?>
        <li><?= $item ?></li>
      <?php endforeach; ?>
    </ul>
  </body>
</html>
```



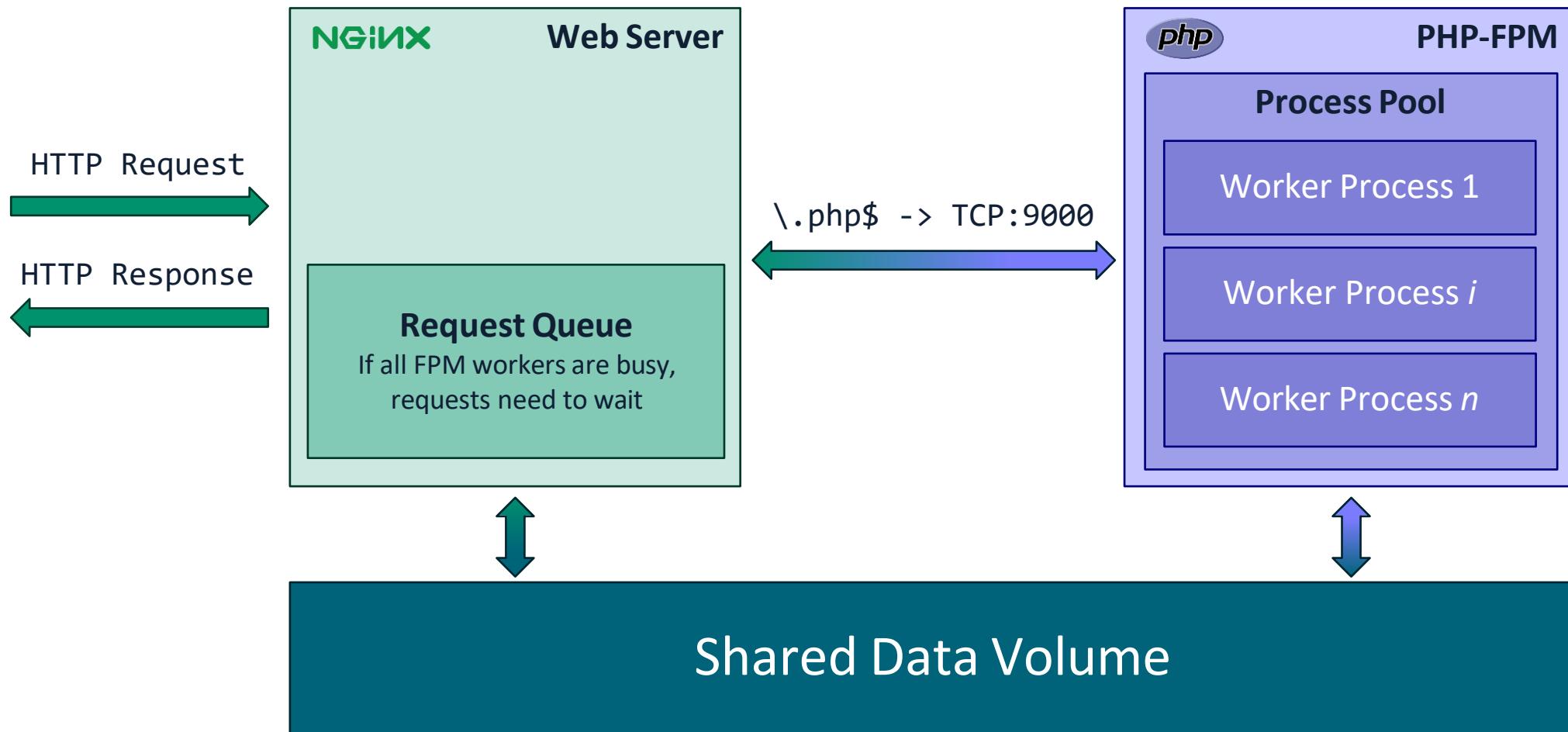
SERVER-SIDE SCRIPTING: PHP

- Le soluzioni di scripting lato server in genere supportano gli sviluppatori in molte attività noiose tipiche della gestione delle richieste HTTP
- Le richieste vengono pre-elaborate automaticamente
- In PHP, i parametri di richiesta sono disponibili in array associativi (ad esempio: `$_GET`,
`$_POST`).
- Ad esempio, `$_POST['todo']` può essere utilizzato per accedere al parametro di richiesta todo.
- In PHP, i cookie sono già analizzati nell'array associativo `$_COOKIES`
- Non è necessario emettere esplicitamente su `stdout` l'intera risposta

EXAMPLE: PHP-BASED TO-DO LIST

- We will re-implement our To-do list web app using PHP 8
- Since we're making a leap forward towards modern web development, we'll make our app slightly more interesting
 - We'll manage persistency using an actual database (SQLite)
 - We'll use Bootstrap to make our pages slightly less ugly and responsive
 - We'll introduce a footer and a navbar, as template parts
- We'll use a modern architecture leveraging NGINX and PHP-FPM

EXAMPLE: PHP-BASED TO-DO LIST



EXAMPLE: PHP TO - DO LIST

- Live demo time! (again!)
- Repo: <https://github.com/luistar/php-to-do-list>



REFERENCES (1/2)

- **Server-side programming**

MDN web docs

<https://developer.mozilla.org/en-US/docs/Learn/Server-side>

https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps

https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction

https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Client-Server_overview

- **CGI**

IBM Docs

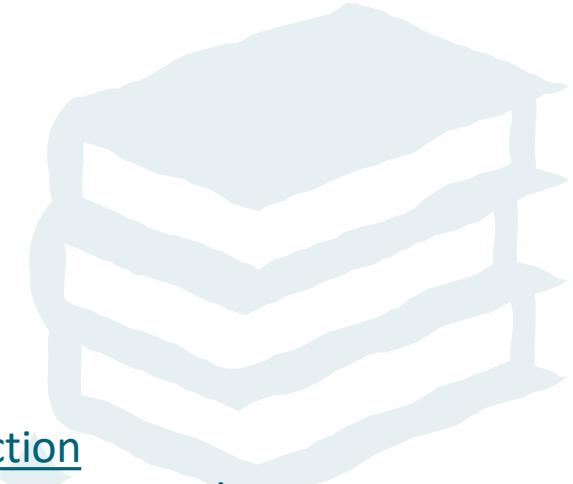
<https://www.ibm.com/docs/en/i/7.5?topic=functionality-cgi>

RFC 3875: The Common Gateway Interface (CGI) Version 1.1 (2004)

<https://datatracker.ietf.org/doc/html/rfc3875>

Apache HTTPD documentation (in case you want to take a look at how to configure a web server for CGI)

<https://httpd.apache.org/docs/2.4/howto/cgi.html>



REFERENCES (2/2)

- **PHP**

Official website

<https://www.php.net/>

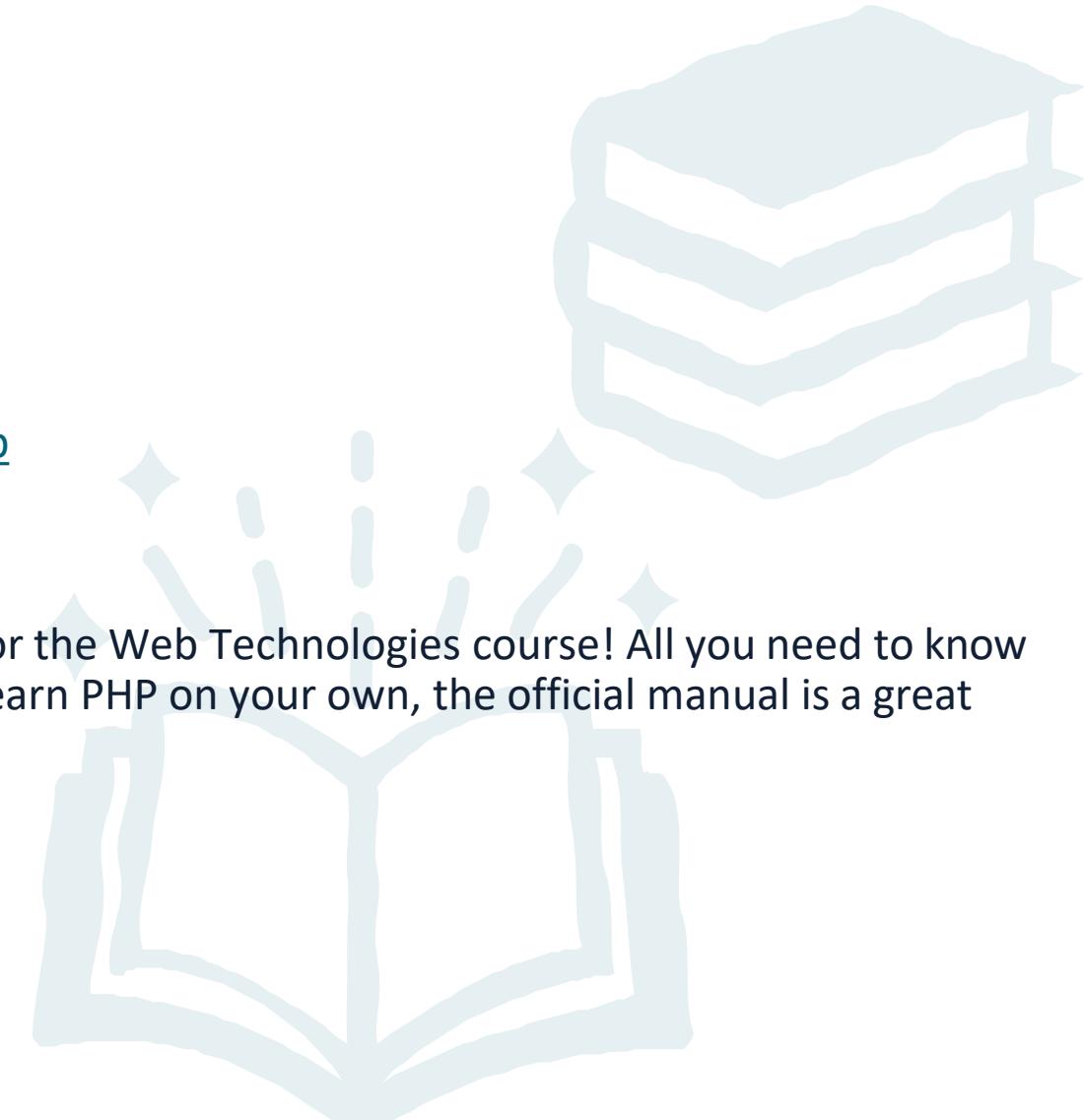
Getting started page

<https://www.php.net/manual/en/getting-started.php>

Official manual

<https://www.php.net/manual/en/>

⚠ You are **not** required to learn the PHP language for the Web Technologies course! All you need to know is what's included in the slides. In case you want to learn PHP on your own, the official manual is a great starting point.



UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
WEB TECHNOLOGIES – LECTURE 10

JAVASCRIPT IN A SERVER ENVIRONMENT: NODE.JS

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

<https://docenti.unina.it/luigiliberolucio.starace>

<https://luistar.github.io>

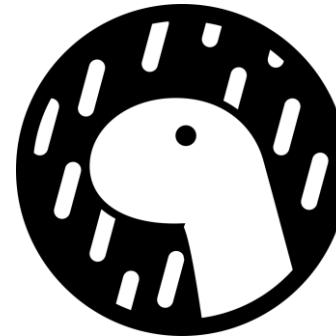


PREVIOUSLY, ON WEB TECHNOLOGIES

- We now know **server-side scripting**
 - Web pages can be generated by programs on-the-fly
- We've learned a good deal about JavaScript in the first lectures
 - So far, we've only used it in a browser environment
 - Well, we can also use it in a **server environment!**
 - Several different runtimes allow us to do so...



[Node.js](#)



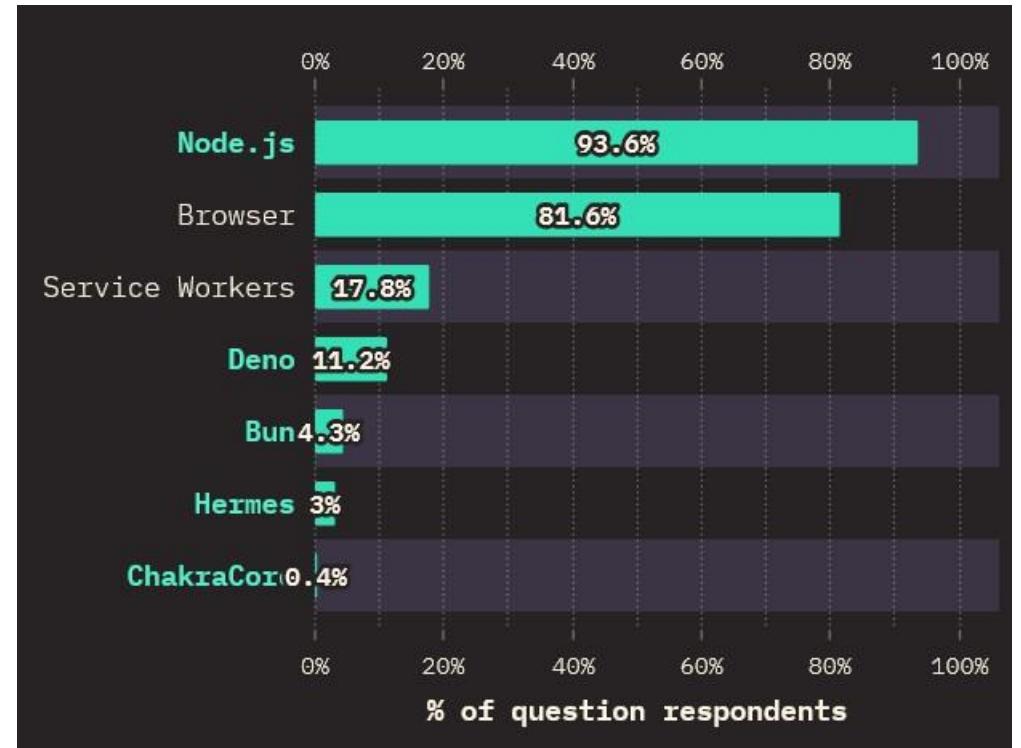
[Deno](#)



[Bun](#)

NODE.JS

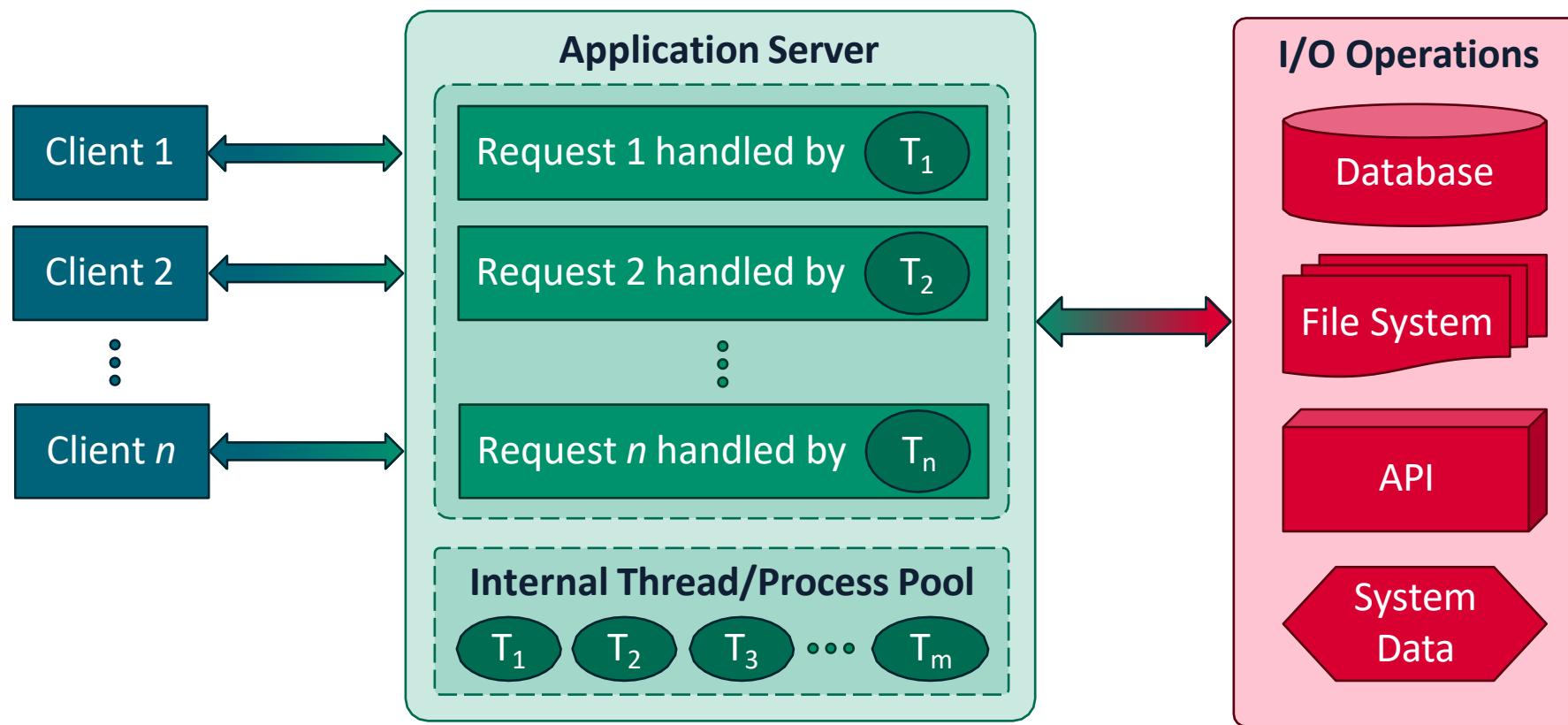
- Open-source e multiplattforma
- Ambiente di runtime JavaScript
- Esegue il motore JavaScript V8 di Google dal progetto Chromium
- Modello I/O basato su eventi, asincrono per impostazione predefinita, non bloccante
- Di gran lunga l'ambiente di esecuzione JS più popolare



[State of JavaScript 2022 Report](#)

MULTI-THREADED REQUEST HANDLING

In altri ambienti/linguaggi di esecuzione, una richiesta viene gestita da un thread/processo dedicato



EXPENSIVE, BLOCKING I/O OPERATIONS

Quando gestiscono le richieste, le applicazioni Web devono spesso eseguire operazioni di I/O che richiedono molto tempo e bloccano

Accedere a un database

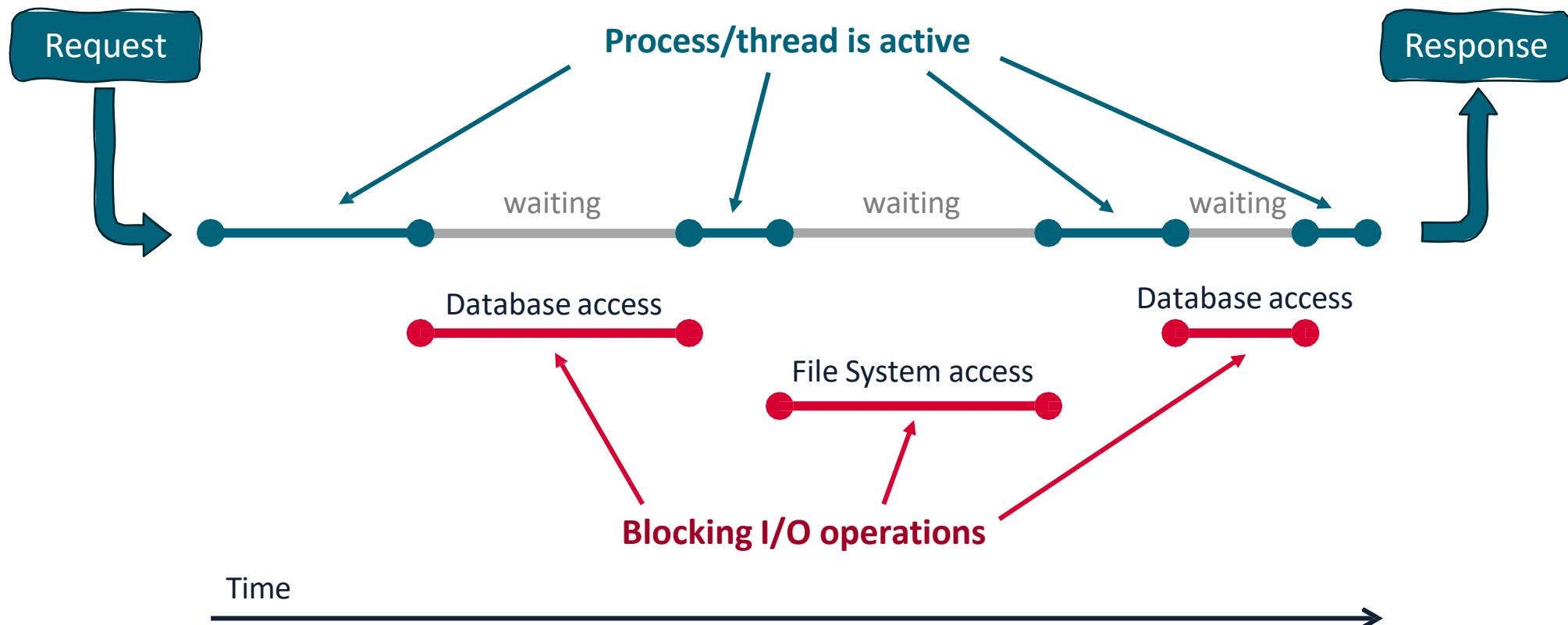
Accedere a un'API esterna (ad esempio: utilizzando `fetch()`)

Leggere un file dal file system locale

Leggi il corpo della richiesta

Di conseguenza, i thread/processi che gestiscono una richiesta trascorrono molto tempo in attesa del completamento di queste operazioni

BLOCKING I/O EXAMPLE



MULTI-THREADED REQUEST HANDLING

- Può essere inefficiente (molto tempo di attesa del thread)
- Scalabilità limitata (numero di richieste simultanee gestite)
- Il pool di thread è preallocato in base alle dimensioni predefinite.
Non è possibile gestire più richieste simultanee di quelle consentite dal pool di thread.
- Pensiamoci con un esempio.
- Un ristorante ha 10 tavoli. La direzione decide di assumere 10 camerieri. Ogni cameriere è assegnato esattamente a un tavolo.

MULTI-THREADED REQUEST HANDLING

- Quando un cliente arriva, viene fatto accomodare al primo tavolo libero e gli viene dato un menu dal cameriere.
- I clienti impiegano dai 5 ai 10 minuti per decidere cosa vogliono. Durante questo periodo, il cameriere rimane inattivo.
- Quando un cliente decide cosa vuole, il cameriere prende la sua ordinazione e la porta allo chef.
- Lo chef impiega dai 15 ai 30 minuti per preparare il cibo. Durante questo periodo, il cameriere rimane inattivo.

MULTI-THREADED REQUEST HANDLING

- Quando il cibo è pronto, il cameriere lo porta al cliente. Il cliente impiega dai 15 ai 30 minuti per mangiare il cibo. Durante questo periodo, il cameriere rimane inattivo.
- Una volta che il cliente ha finito di mangiare, il cameriere pulisce il tavolo e chiede a un manager di preparare il conto. La preparazione del conto richiede dai 2 ai 5 minuti. Durante questo periodo, il cameriere rimane inattivo.
- Una volta che il conto è pronto, il cameriere lo porta al cliente, che paga e lascia lo stabilimento.
- Un nuovo cliente può ora essere servito dallo stesso cameriere allo stesso tavolo.

MULTI-THREADED REQUEST HANDLING

- Nel nostro esempio, i camerieri sono thread che servono i clienti (richieste).
- Al massimo 10 clienti (richieste) possono essere serviti in qualsiasi momento.
- Un cliente che decide cosa ordinare, un cuoco che prepara il cibo, un cliente che lo mangia e un manager che prepara il conto stanno bloccando le operazioni. Il cameriere rimane inattivo durante queste operazioni.
- Le risorse non vengono utilizzate in modo molto efficiente...
- Cosa succede se vogliamo servire un'ulteriore richiesta alla volta? Dobbiamo assumere un nuovo cameriere.

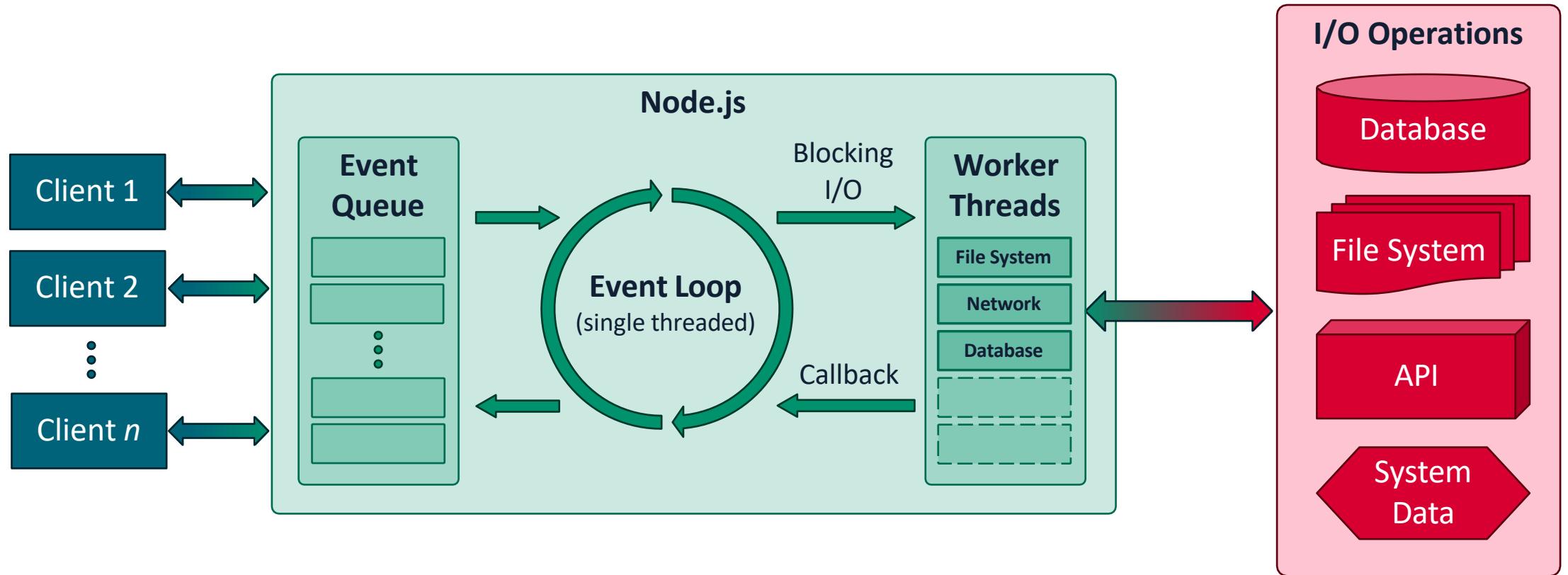
IMPROVING EFFICIENCY AT OUR RESTAURANT

- Il ristorante assume un solo cameriere per assistere ai suoi 10 tavoli
- Quando arriva un cliente, il cameriere lo saluta e lo fa sedere al suo tavolo, dandogli un menu. Mentre il cliente sceglie, il cameriere è disponibile a soddisfare le esigenze degli altri clienti.
- Una volta che il cliente decide l'ordine, il cameriere lo porta al cuoco. Mentre il cuoco prepara il cibo, il cameriere torna in sala da pranzo ed è disponibile per assistere gli altri clienti.
- Una volta che il cibo è pronto (e suona il campanello della cucina), il cameriere porta il cibo al cliente. Mentre il cliente mangia, il cameriere è disponibile a occuparsi degli altri clienti.

NODE.JS: PHILOSOPHY

- Node.js si basa su una filosofia che è abbastanza simile al nostro esempio di ristorante più efficiente
- Un programma Node.js viene eseguito in un unico processo
- Non è necessario creare nuovi thread per ogni richiesta
- Il runtime Node.js fornisce un set di primitive di I/O asincrone nella libreria standard che impediscono il blocco del codice JavaScript
- La maggior parte delle librerie sfrutta il paradigma non bloccante quando possibile
- Ampia adozione del modello di callback
- In genere, le operazioni sincrone sono un'eccezione piuttosto che la norma

THE SINGLE-THREADED EVENT LOOP



INSTALLING NODE.JS

To install Node.js you can:

- use the installers available on the [official website](#)
- use your favourite [package manager](#)
- use a Node.js version manager such as [nvm](#), [nvs](#), [nvm4w](#)
 - Nice if you want to experiment with different versions and switch frequently

In the Web Technologies course (2023/24), we'll use **Node.js 22.14.0**

- As long as you use a supported version, there should be no issues

INSTALLING NODE.JS

```
@luigi → D/0/T/W/2/e/10-Node.js $ nvs
```

```
-----  
| Select a node version |  
+-----+  
| a) node/21.2.0 |  
| b) node/21.1.0 |  
| c) node/21.0.0 |  
| d) node/20.11.1 (Iron) |  
| e) node/20.11.0 (Iron) |  
| f) node/20.10.0 (Iron) |  
| [g] node/20.9.0 (Iron) |  
| h) node/20.8.1 |  
| i) node/20.8.0 |  
'-- \-----'
```

Type a hotkey or use Down/Up arrows then Enter to choose an item.

INSTALLING NODE.JS

```
@luigi → D/0/T/W/2/e/10-Node.js $ nvs
-----
| Select a version
+-----+
| [a] node/20.9.0/x64 (Iron) [current] [default]
| b) node/20.8.1/x64
|
| ,) Download another version
| .) Don't use any version
'-----'

Type a hotkey or use Down/Up arrows then Enter to choose an item.
```

INSTALLING NODE.JS

- To check that everything is ok, you can run: `node --version`

```
@luigi → D/0/T/W/2/e/10-Node.js $ node --version  
v20.9.0
```

```
@luigi → D/0/T/W/2/e/10-Node.js $
```

NODE.JS: HELLO WORLD

```
//hello-world.js
"use strict"
console.log("Hello Node.js!");
```

```
@luigi → D/O/T/W/2/e/10-Node.js $ node .\hello-world.js
Hello Node.js!
```

```
@luigi → D/O/T/W/2/e/10-Node.js $
```

NODE.JS VS BROWSER ENVIRONMENT

- In un ambiente browser, utilizziamo principalmente JavaScript per manipolare il DOM. Avevamo finestre, documenti, API della piattaforma Web come cookie, LocalStorage, ecc...
- In Node.js, questi oggetti e API non sono disponibili
- D'altra parte, in Node.js, possiamo sfruttare la libreria standard e i suoi numerosi moduli per l'accesso al file system e l'accesso alla rete
- È possibile leggere e scrivere file sul filesystem
- È possibile ascoltare su un socket le richieste in arrivo

NPM: THE NODE PACKAGE MANAGER

- Uno dei principali motori del successo di Node.js è npm
- Il più grande archivio in un solo linguaggio al mondo
- C'è un pacchetto per (quasi!) tutto
- NPM fornisce modi per scaricare e gestire le dipendenze per i progetti Node.js

NPM: GETTING STARTED

- **npm** should be installed by default with Node.js
- You can check that npm is available using: **npm --version**

```
@luigi → D/0/T/W/2/e/10-Node.js $ npm --version  
10.2.2
```

```
@luigi → D/0/T/W/2/e/10-Node.js $
```

- To create an npm package in the current directory, run: **npm init**
- To create an ES compatible module, use **npm init es6**

NPM: CREATING A NEW PACKAGE

- npm init chiederà alcune informazioni sul tuo pacchetto/progetto
- Nome del pacchetto, versione, repository (se presente), licenza, autore,...
- Entry point (il primo file javascript da eseguire), comando di test (se presente)
- Alla fine, npm crea un file project.json nella stessa directory
- Il file project.json contiene tutte le informazioni sul pacchetto, incluse le sue dipendenze (se presenti) e i comandi per eseguirlo.
- Pensalo come la versione di npm del pom.xml di Maven
- Poiché lavoreremo con i moduli ES, utilizzare npm init es6 durante la creazione di un nuovo pacchetto

THE PACKAGE.JSON FILE

```
{  
  "name": "hello-web-technologies",  
  "version": "0.0.1",  
  "description": "Web Technologies' first npm package",  
  "main": "app.js",  
  "type": "module", //inserito con es6  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "keywords": [  
    "greetings"  
  ],  
  "author": "Luigi Libero Lucio Starace",  
  "license": "MIT"  
}
```

PACKAGE.JSON: MAIN FILE AND TASKS

- Il file main (app.js nel nostro esempio) viene eseguito quando il codice client importa il nostro pacchetto (o progetto). Generalmente esporta variabili/funzioni pubbliche
- La proprietà scripts può essere utilizzata per specificare le attività della riga di comando
- L'attività di test è stata definita per impostazione predefinita e non esegue molte operazioni
- Le attività possono essere eseguite eseguendo: npm <taskname>
- Possiamo aggiungere un'attività di avvio modificando la proprietà scripts come segue:

```
"scripts": {  
  "start": "node app.js",  
  "test": "echo \\"$Error: no test specified\\" && exit 1"  
},
```

NPM: RUNNING TASKS

```
//from package.json
"scripts": {
  "start": "node app.js",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

```
//app.js file
"use strict";
console.log("Hello Web Technologies!");
```

```
@luigi → D/O/T/W/2/e/10-Node.js $ npm start
```

```
> hello-web-technologies@0.0.1 start
> node app.js
```

```
Hello Web Technologies!
```

```
@luigi → D/O/T/W/2/e/10-Node.js $
```

NPM: INSTALLING DEPENDENCIES

- Let's make our package more interesting
- We search the npm library, and find [this interesting package: cowsay](#)
- Let's use this package
- To install a dependency, run **npm install <packagename>**

NPM: INSTALLING DEPENDENCIES

- In our case, **npm install cowsay**
- Two things happen:
 1. npm keeps track of the new dependency, by adding a new section in the **package.json** file

```
"dependencies": {  
    "cowsay": "^1.5.0"  
}
```

2. Dependencies are downloaded in the **node_modules** directory
 - Notice that npm downloaded not only the package we requested, but also its (recursive) dependencies!

NPM: IMPORTING DEPENDENCIES

```
//app.js file
import cowsay from 'cowsay';

console.log(cowsay.think({
  text: "Hello Web Tech!"
}));
```

```
@luigi → D/0/T/W/2/e/10-Node.js $ npm start
> hello-web-technologies@0.0.1 start
> node app.js
```

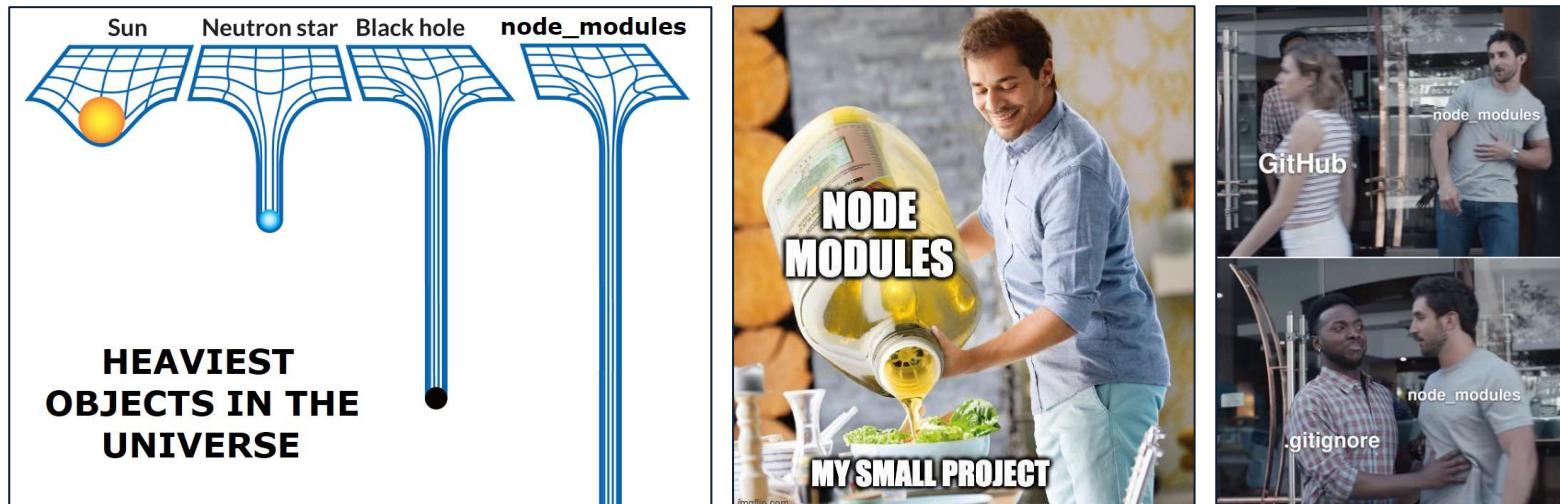
(Hello Web Tech!)



```
@luigi → D/0/T/W/2/e/10-Node.js $
```

DISTRIBUTING AN NPM PACKAGE

- Quando distribuiamo un pacchetto npm, dobbiamo solo fornire il nostro codice e un descrittore package.json.
- Non c'è bisogno di aggiungere /node_modules/ al controllo delle versioni!
- Il comando npm install può essere utilizzato per installare tutte le dipendenze elencate nel file package.json



A FIRST WEB APPLICATION WITH NODE.JS

```
import http from 'http';

const PORT = 3000;

let server = http.createServer(function(request, response){
    let course = "Web Technologies";
    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(`<!DOCTYPE html>
<html><body>
    <h1>Hello ${course}</h1>
    <p>Current date is ${new Date().toString()}</p>
</body></html>`);
    response.end();
}).listen(PORT);

console.log(`Web app listening on port ${PORT}`);
```



DEBUGGING A NODE.JS APP IN VS CODE

- Il modo più semplice è abilitare la funzione di auto attach
- In questo modo verrà collegato un debugger ai processi avviati Node.js in VSCode
- Per abilitare l'attacco automatico, utilizzare CTRL+MAIUSC+P per visualizzare il riquadro dei comandi VSCode, quindi cercare «Attiva/disattiva collegamento automatico»
- Ti suggerisco quindi di selezionare la modalità «Smart», che si collega automaticamente solo agli script che si trovano al di fuori di /node_modules/

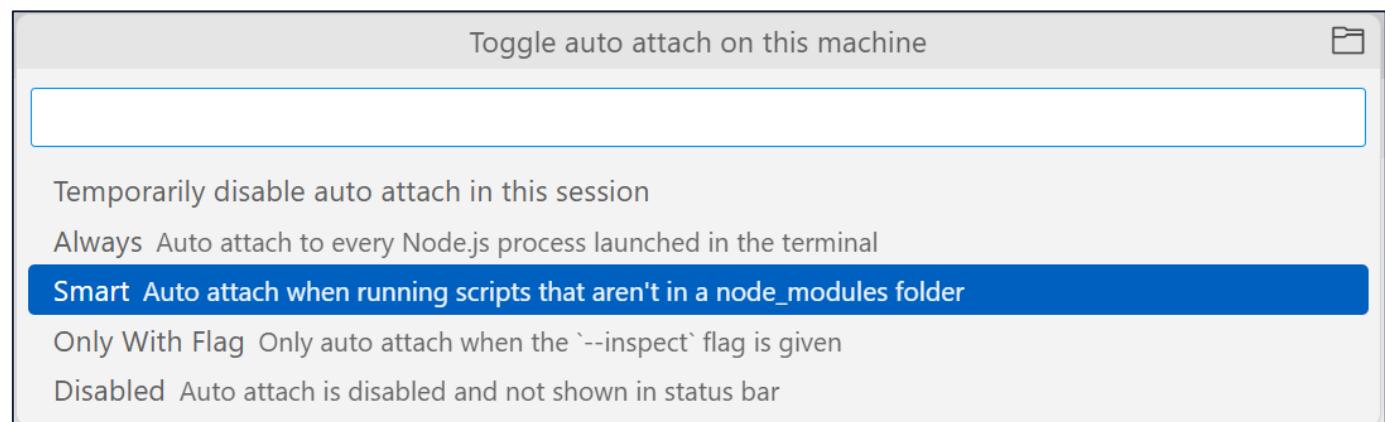
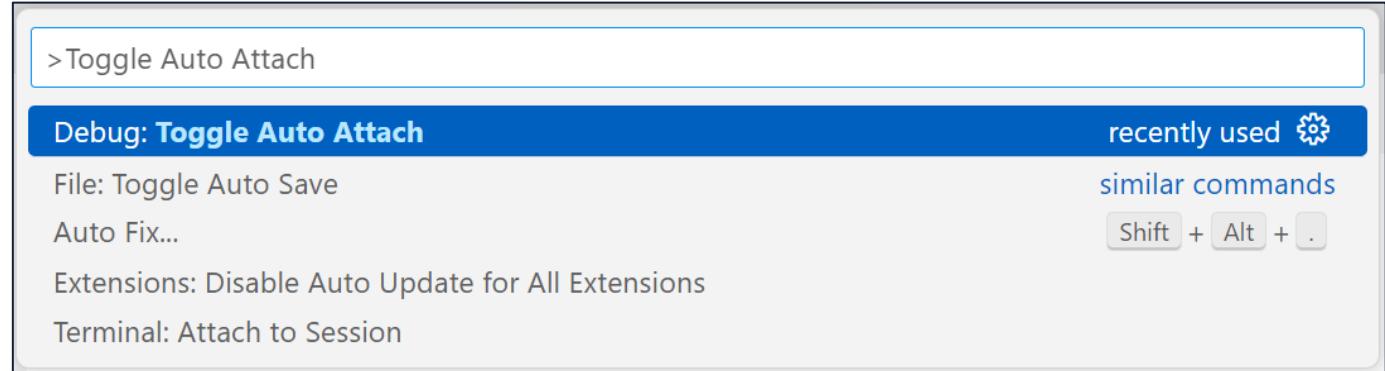
DEBUGGING A NODE.JS APP IN VS CODE

CTRL + ⌘ + P

Windows/Linux

⌘ + ⌘ + P

Mac



DEBUGGING A NODE.JS APP IN VS CODE

Quando avvii l'app Node.js dal terminale VSCode, un debugger si collegherà
Il pannello Esegui ed Esegui debug verrà attivato e potrai avviare correttamente il debug dell'app

The screenshot shows the VS Code interface with the following elements:

- Explorer View:** Shows a project structure for "WEB-APP" containing files like .vscode, node_modules, app.js, index.js, jsconfig.json, package-lock.json, and package.json. A yellow arrow points to the "Run" icon in the sidebar.
- Code Editor:** Displays the content of "index.js".

```
1 import http from 'http';
2
3 const PORT = 3000;
4
5 let server = http.createServer(function(request, response){
6   response.writeHead(200, {'Content-Type': 'text/html'});
7   response.write("<!DOCTYPE html><html><body><h1>Hello Web Tech</h1></body></html>");
8   response.end();
9 }).listen(PORT);
10
11 console.log(`Web app listening on port ${PORT}`);
```

A yellow arrow points to the green "Run" button in the top right corner of the editor.
- Terminal:** Shows the command being run: `@luigi → D/O/T/L/T/2/e/1/web-app $ npm start`. The output shows the application starting and the debugger connecting.

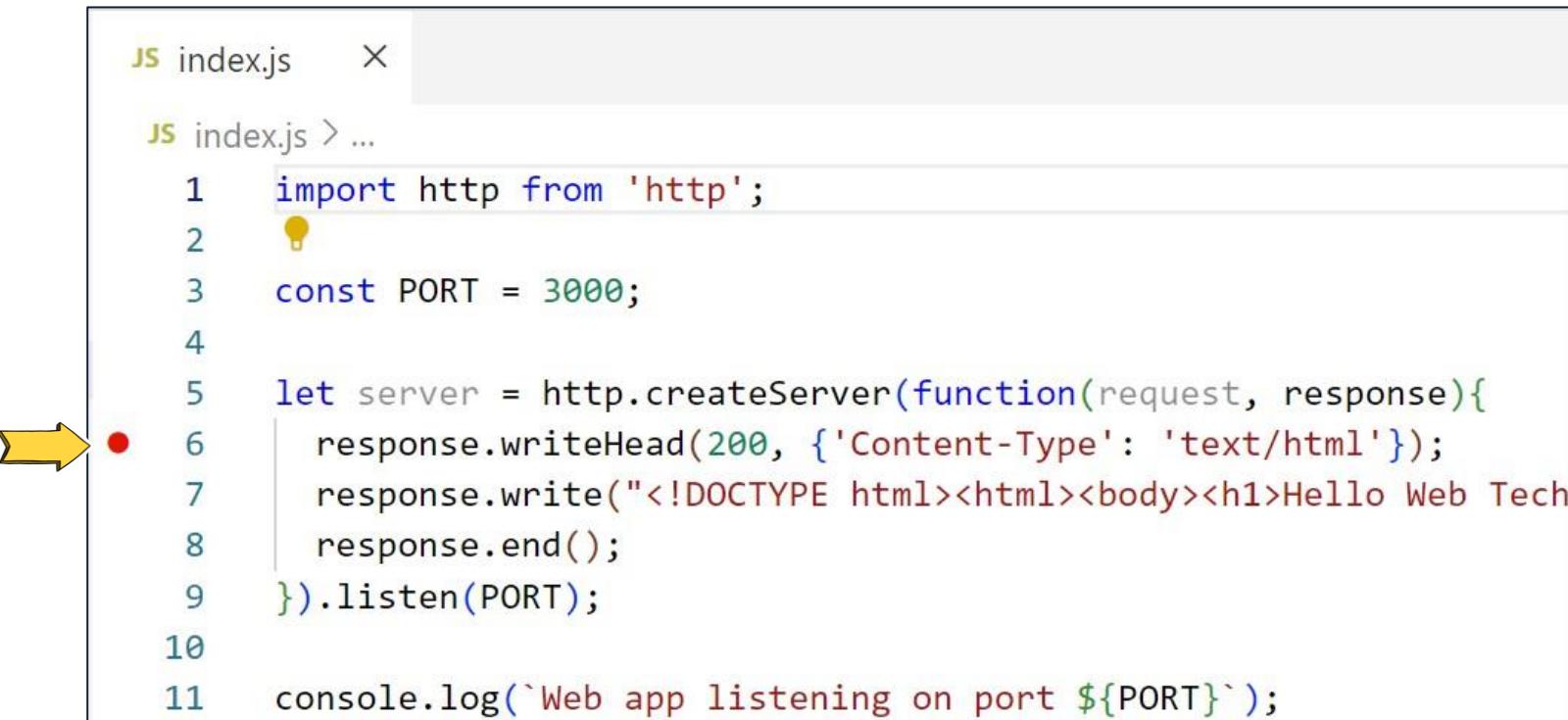
```
> web-app@0.0.1 start
> node index.js

Debugger listening on ws://127.0.0.1:58896/06d4d583-e2b3-4a5f-baca-8635eb9dbedf
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
Web app listening on port 3000
```

Yellow arrows point to the terminal output and the "Debugger attached." message.
- Bottom Status Bar:** Shows file information (Ln 11, Col 50), code style settings (Spaces: 2, CRLF), and language (JavaScript).

DEBUGGING A NODE.JS APP IN VS CODE

- È possibile aggiungere punti di interruzione come di consueto, facendo clic accanto al numero di riga della riga di codice su cui si desidera inserire il breakpoint



```
JS index.js ×  
JS index.js > ...  
1 import http from 'http';  
2  
3 const PORT = 3000;  
4  
5 let server = http.createServer(function(request, response){  
6   response.writeHead(200, { 'Content-Type': 'text/html' });  
7   response.write("<!DOCTYPE html><html><body><h1>Hello Web Tech</h1></body></html>");  
8   response.end();  
9 }).listen(PORT);  
10  
11 console.log(`Web app listening on port ${PORT}`);
```

DEBUGGING A NODE.JS APP IN VS CODE

- L'esecuzione del codice viene sospesa quando raggiunge un breakpoint
- Il pannello Esegui e debug ci consente di ispezionare le variabili, di osservare le espressioni, di esaminare lo stack di chiamate e molto altro ancora
- Decisamente più efficace di console.logging la nostra via d'uscita dal bug!
- Possiamo anche esercitare un controllo a grana fine del flusso di esecuzione con azioni di debug

DEBUGGING: ACTIONS



Action	Description
	Resume the normal execution flow (until the next breakpoint)
	Execute the next statement as a single unit, without inspecting its inner component steps
	Enter the next statement to follow its execution line-by-line.
	When inside a method or subroutine, return to the earlier execution context by completing remaining lines of the current method as though it were a single command.
	Terminate the current program execution and start debugging again using the current run configuration (does not work with auto attach!).
	Terminate the current debugging session.

DEBUGGING A NODE.JS APP IN VS CODE

The screenshot shows the Visual Studio Code (VS Code) interface with the following details:

- File Bar:** File, Edit, Selection, ...
- Search Bar:** web-app
- Editor:** index.js (JavaScript file)
- Code:**

```
1 import http from 'http';
2
3 const PORT = 3000;
4
5 let server = http.createServer(function(request, response){
6   response.writeHead(200, {'Content-Type': 'text/html'});
7   response.write("<!DOCTYPE html><html><body><h1>Hello Web Tech</h1></body></html>");
8   response.end();
9 }).listen(PORT);
10
11 console.log(`Web app listening on port ${PORT}`);
```
- Run and Debug View:** No Configured
- Variables View:** Local, Module, Global
- Watch View:** response.statusCode: 200
- Terminal:**

```
@luigi → D/O/T/L/T/2/e/1/web-app $ npm start
> web-app@0.0.1 start
> node index.js

Debugger listening on ws://127.0.0.1:59170/0c5c161d-714e-427f-a328-c75b2ebcb2e7
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
Web app listening on port 3000
```
- Status Bar:** Ln 6, Col 12, Spaces: 2, UTF-8, CRLF, {}, JavaScript, Auto Attach: Smart

LIVE RELOADING IN NODE.JS

- Ogni volta che apportiamo una modifica al nostro codice, dobbiamo riavviare l'intero server (npm start, o node app.js, ecc...)
- Per rendere lo sviluppo più piacevole, possiamo utilizzare strumenti che monitorano la nostra base di codice per le modifiche e riavviare il server di sviluppo quando necessario (ovvero il ricaricamento in tempo reale).
- Una di queste utilità è nodemon
- Puoi installarlo usando npm (npm install nodemon)
- Quindi devi solo sostituire il node app.js con nodemon app.js

LIVE RELOAD WITH NODEMON (EXAMPLE)



The screenshot shows a terminal window with the following interface elements:

- Top bar: PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (underlined), PORTS.
- Right side: A tab bar with "node" selected, followed by "+" (new tab), "x" (close), "...", and other icons.

The terminal output is as follows:

```
○ @luigi → D/O/T/L/T/2/e/1/web-app $ npm start
> web-app@0.0.1 start
> nodemon app.js

[nodemon] 3.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node app.js`
Debugger listening on ws://127.0.0.1:62283/30cd1607-a3ed-4b2d-ac76-e910303ebbe0
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
Web app listening on port 3000
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
Debugger listening on ws://127.0.0.1:62287/76349d46-d3af-4b18-9a1b-a2a156ec7b97
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
Web app listening on port 3000
```

A callout box highlights the "start" script from the package.json file:

```
//from package.json
"scripts": {
  "start": "nodemon app.js"
},
```

REFERENCES (1/2)

- **Getting started**

Node.js Docs

<https://nodejs.org/en/learn/>

Relevant parts: Introduction to Node.js, How to install Node.js, Differences between Node.js and the Browser, The V8 JavaScript Engine, An introduction to the npm package manager, ECMAScript 2015 (ES6) and beyond.

- **Eloquent JavaScript (3rd edition)**

By Marijn Haverbeke

Freely available at <https://eloquentjavascript.net/>

Chapter 20

- **Mixu's Node book: A book about using Node.js**

By Mikito Takada

Freely available at <http://book.mixu.net/node/single.html>

You can also download it in PDF, epub, mobi formats from <http://book.mixu.net/>

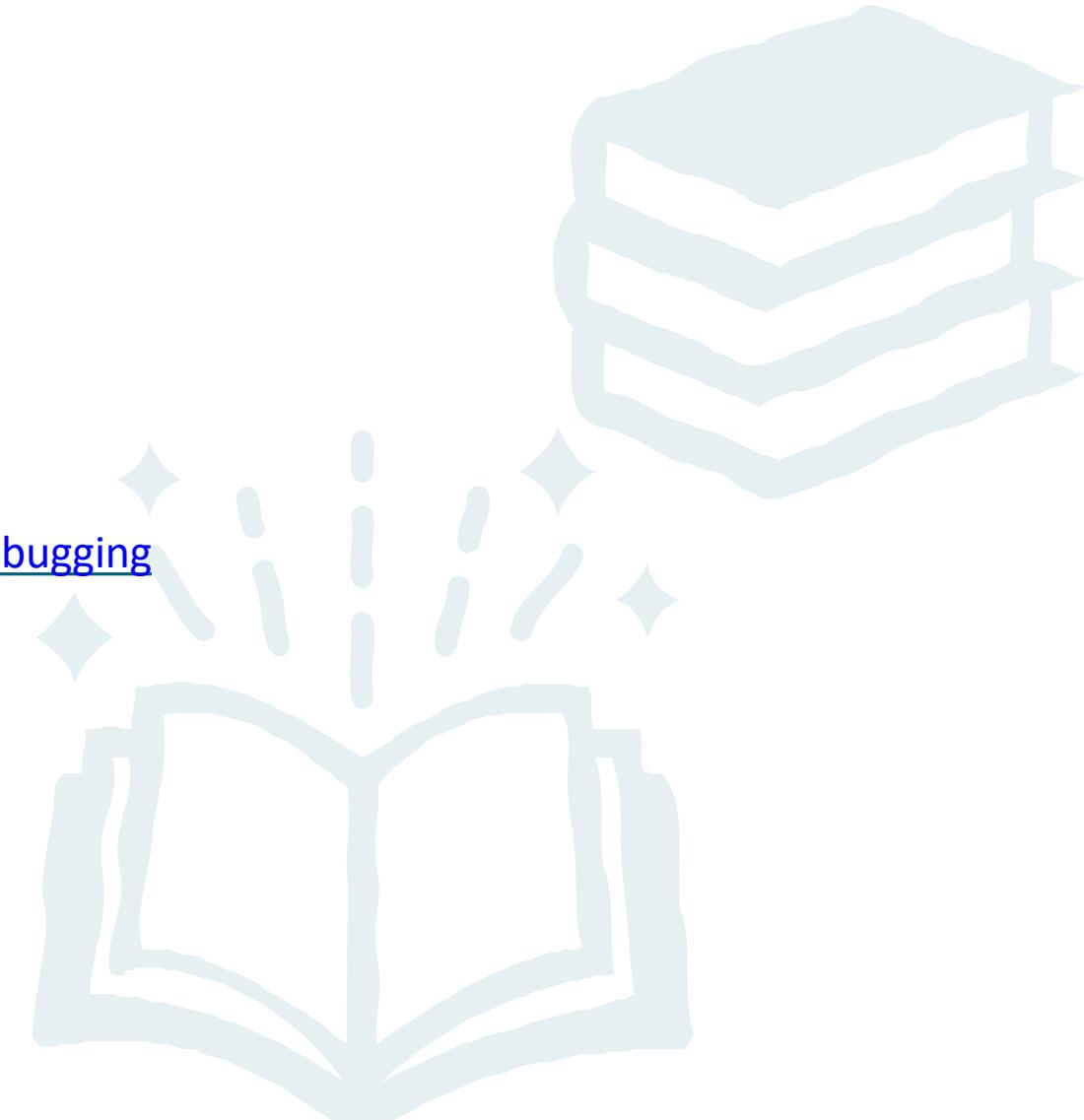
Relevant parts: Chapter 2 (What is Node.js?), Chapter 8, Section 8.2 (NPM)

REFERENCES (2/2)

- **Nodemon**
- **Node.js debugging in VS Code**

Nodemon official website
<https://nodemon.io/>

Visual Studio Code Docs
<https://code.visualstudio.com/docs/nodejs/nodejs-debugging>



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
WEB TECHNOLOGIES – LECTURE 11**

IMPLEMENTING WEB APPS WITH NODE.JS

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

<https://luistar.github.io>

<https://docenti.unina.it/luigiliberolucio.starace>



TO - DO LIST WEB APP

Will this be less painful than the infamous CGI+Bash example?



THE TO - DO LIST WEB APP

Node.js To-do List

Home To-do list Reset app

 Node.js To-do App

A simple web app built using Node.js 20.9.0. The app uses only standard Node.js modules and the [Plug template engine](#). A session tracking mechanism, implemented from scratch, is included. Source code and Docker environment available on [Github](#).

Beware: This web app is intended as a basic first example to showcase server-side scripting. It is not representative of modern web development practices!

[Manage your To-do List](#)

Node.js To-do List

Home To-do list Reset app

Welcome to your To-do List

Welcome to our Node.js To-do list web app! Links to reset the list or to go back to the homepage are on the navbar above. Use the form below to add items to the To-Do list!

To-do Item

Save To-do

The To-do list is currently empty! Add the first item using the form above.

© Web Technologies Course

B.Sc. in Computer Science program,
Università degli Studi di Napoli Federico II

Course held by [Luigi Libero Lucio Starace, Ph.D.](#)

Node.js To-do List

Home To-do list Reset app

To-do list app has been reset

The To-do list app has been successfully reset and all items and users have been deleted. [Go back to the homepage](#).

© Web Technologies Course

B.Sc. in Computer Science program,
Università degli Studi di Napoli Federico II

Course held by [Luigi Libero Lucio Starace, Ph.D.](#)



TO-DO LIST APP: HOMEPAGE

Node.js To-do List

Home To-do list Reset app

 node
js

Node.js To-do App

A simple web app built using Node.js 20.9.0. The app uses only standard Node.js modules and the [Pug template engine](#). A session tracking mechanism, implemented from scratch, is included. Source code and Docker environment available on [Github](#).

Beware: This web app is intended as a basic first example to showcase server-side scripting. It is not representative of modern web development practices!

Manage your To-do List

- Landing page
- Includes links to the list page and the reset app page.

TO - DO LIST APP: THE LIST

The screenshot shows a web application titled "Node.js To-do List". The main content area displays a welcome message: "Welcome to your To-do List". Below this, there is a form with a red dashed border containing a text input field labeled "To-do Item" and a blue "Save To-do" button. A list of saved items is shown below the form, each with a teal dashed border: "Learn Node.js" and "Learn Express". At the bottom of the page, there is footer information: "© Web Technologies Course", "B.Sc. in Computer Science program, Università degli Studi di Napoli Federico II", "Course held by [Luigi Libero Lucio Starace, Ph.D.](#)", and social media icons for globe, Facebook, LinkedIn, Instagram, and Telegram.

Form for saving new To-do items. Submits using POST to the same url as the page.

List showing the saved To-do list items

TO - DO LIST APP: RESET PAGE

Node.js To-do List

Home To-do list Reset app

To-do list app has been reset

The To-do list app has been successfully reset and all items and users have been deleted. [Go back to the homepage.](#)

© Web Technologies Course

B.Sc. in Computer Science program,
Università degli Studi di Napoli Federico II

Course held by [Luigi Libero Lucio Starace, Ph.D.](#)



- Resets the app (i.e., deletes all saved To-do items)

IMPLEMENTING A TO-DO LIST IN NODE.JS

- We start by creating an http server

```
import http from 'http';

const PORT = 3000;

let server = http.createServer();
server.listen(PORT);

server.on('request', handleRequest);

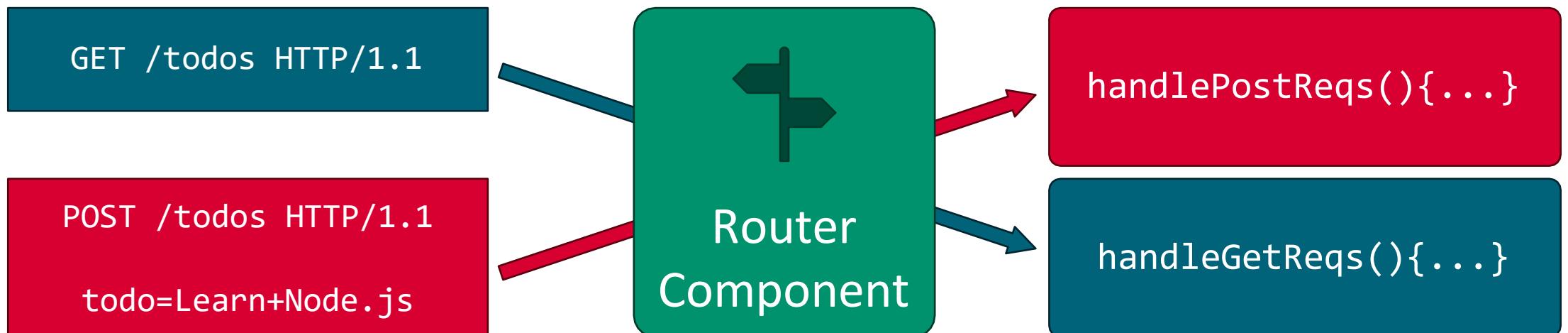
function handleRequest(request, response){
  /* handle request */
}
```

WEB APPS WITH BARE NODE.JS

- Node.js ci consente di implementare in modo semplice ed efficiente un server http sfruttando i moduli integrati e il suo ciclo di eventi a thread singolo
- Node.js esegue anche una pre-elaborazione sulle richieste (ad esempio: analizza le intestazioni) e fornisce un'astrazione per le richieste e le risposte.
- A parte questo, siamo da soli quando si tratta di implementare l'app To-do list
- Le prime tre questioni che dobbiamo affrontare sono le seguenti:
 - Routing
 - Templating
 - Parsing request bodies

ROUTING

- La nostra applicazione web gestirà molti tipi diversi di richieste HTTP (ad esempio: mostra homepage, mostra elenco, ripristina app, ...)
- Un primo problema fondamentale da affrontare è il **routing**
- **Given a request, what code should I execute to serve that request?**



ROUTING – PATHS

- È necessario gestire le richieste a percorsi diversi
- Una homepage, una pagina in cui si trovano le cose da fare
- sono elencati, una pagina di ripristino, ...
- Le richieste a ciascun percorso possono essere gestite diversamente

```
function handleRequest(request, response){  
  switch(request.url){  
    case "/":  
      renderHomepage(request, response);  
      break;  
    case "/reset":  
      handleResetRequest(request, response);  
      break;  
    case "/todo":  
      handleTodoListRequest(request, response);  
      break;  
    /* and so on... */  
    default:  
      handleError(request, response, 404);  
  }  
}
```

ROUTING – HANDLING HTTP METHODS

Le richieste a un determinato percorso possono anche essere gestite in modo diverso a seconda del metodo HTTP utilizzato

Le richieste GET a /todos ottengono l'elenco degli elementi da fare
Le richieste POST a /todos tentano di salvare un nuovo elemento

```
function handleTodoListRequest(request, response, context={}){
  switch(request.method){
    case "GET":
      handleTodoListRequestGet(request, response, context); break;
    case "POST":
      handleTodoListRequestPost(request, response, context); break;
    default:
      handleError(request, response, 405, "Unsupported method");
  }
}
```

SERVING STATIC FILES

- Se il nostro output HTML include file statici (ad esempio: immagini, file css o js), dobbiamo configurare il componente Router per usare questi file

```
switch(request.url){  
    /* ... */  
    case "/css/bootstrap.css":  
        fs.readFile('./static/css/bootstrap.css', function(err, data){  
            response.writeHead(200, {'Content-Type': 'text/css'});  
            response.end(data, 'utf-8');  
        }); break;  
    case "/img/node.png":  
        fs.readFile('./static/img/node.png', function(err, data){  
            response.writeHead(200, {'Content-Type': 'image/png'});  
            response.end(data, "binary");  
        }); break;  
}
```

HANDLING ERRORS

- Quando nessuno dei percorsi conosciuti si applica, possiamo restituire un 404

```
switch(request.url){  
  /* other routes */  
  default:  
    handleError(request, response, 404, "Web page not found");  
}
```

```
function handleError(request, response, statusCode, message){  
  let renderedContent = pug.renderFile("./templates/errorPage.pug",  
    {"code": statusCode, "description": message});  
  response.writeHead(statusCode, {"Content-Type": "text/html"});  
  response.end(renderedContent);  
}
```

ROUTING: THERE'S MORE TO THAT

Il routing sembra abbastanza semplice, ma tieni presente che può diventare più complesso man mano che le app crescono (vedremo presto!):

Alcuni percorsi potrebbero essere accessibili solo ad alcuni utenti (ad esempio: amministratori)

Alcuni percorsi possono dipendere da modelli o parametri nel percorso della richiesta (ad esempio: GET /users/42/friends, dove 42 potrebbe essere qualsiasi ID utente)

Potrebbe essere necessario eseguire due o più parti di codice per gestire una richiesta (ad esempio, una funzione che registra solo i dati e una funzione diversa che prepara la risposta)

TEMPLATE ENGINES

PRODUCING HTML OUTPUTS

- A un certo punto, la nostra app Web dovrà produrre del codice HTML da inviare nel corpo della risposta, in modo che i browser possano eseguirne il rendering.
- Nel nostro primo esempio, abbiamo scritto l'HTML manualmente

```
let server = http.createServer(function(request, response){  
    let course = "Web Technologies";  
    response.writeHead(200, {"Content-Type": "text/html"});  
    response.write(`<!DOCTYPE html>  
    <html><body>  
        <h1>Hello ${course}</h1>  
        <p>Current date is ${new Date().toString()}</p>  
    </body></html>`);  
    response.end();  
}).listen(PORT);
```

TEMPLATE ENGINES

- Alcune parti dell'HTML che produciamo possono essere ripetute in più pagine (ad esempio: barra di navigazione, piè di pagina, ecc.)
- Man mano che le pagine diventano sempre più complesse, la creazione di una risposta manipolando e concatenando le stringhe diventa rapidamente irrealizzabile.
- I motori di template sono ottimi per risolvere questo problema!

TEMPLATING WITH PUG (FORMERLY JADE)

- Pug è un motore di template ad alte prestazioni implementato in JavaScript (port disponibili in molte altre lingue)
- Sintassi sensibile agli spazi bianchi (simile a Python) per la scrittura di modelli HTML
- I modelli di pug possono essere facilmente
- «compilato» in codice HTML



INSTALLING AND USING PUG

- Installing Pug is as simple as running `npm install pug`
- The simplest way to use Pug is as follows:

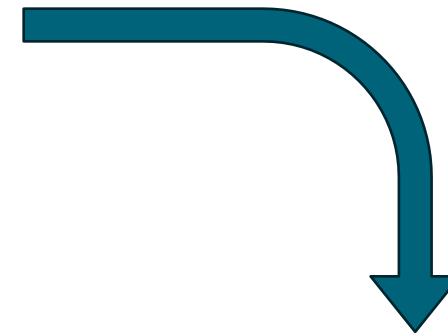
```
import pug from "pug"

//basic_example.pug is the file with the template to render
let html = pug.renderFile("./templates/basic_example.pug");

console.log(html); //html contains the generated HTML code
```

A FIRST PUG TEMPLATE

```
doctype html
html(lang="en")
head
  title Hello Pug!
body
  h1(class="foo") First Pug template
  p This is our first Pug template!
```



```
<!DOCTYPE html>
<html lang="en">
<head><title>Hello Pug!</title></head>
<body>
  <h1 class="foo">First Pug template</h1>
  <p>This is our first Pug template!</p>
</body>
</html>
```

TEMPLATING WITH PUG: INCLUDES

- Pug è molto più di un modo diverso di scrivere HTML...
- Per cominciare, un modello può includere altri modelli
- Questo è di grande aiuto per il riutilizzo dei template!

```
//- index.pug
doctype html
html
  include partials/head.pug
  body
    h1 Hello Pug!
    include partials/footer.pug
```

```
//- partials/head.pug
head
  title Hello Pug
  script(src='/js/script.js')
  link(rel='stylesheet' href='/style.css')
```

```
//- partials/footer.pug
footer#footer
  p Copyright (c) Web Tech
```

TEMPLATING WITH PUG: INCLUDES

```
//- index.pug
doctype html
html
  include partials/head.pug
  body
    h1 Hello Pug!
    include partials/footer.pug
```

```
//- partials/head.pug
head
  title Hello Pug
  script(src='/js/script.js')
  link(rel='stylesheet' href='/style.css')
```

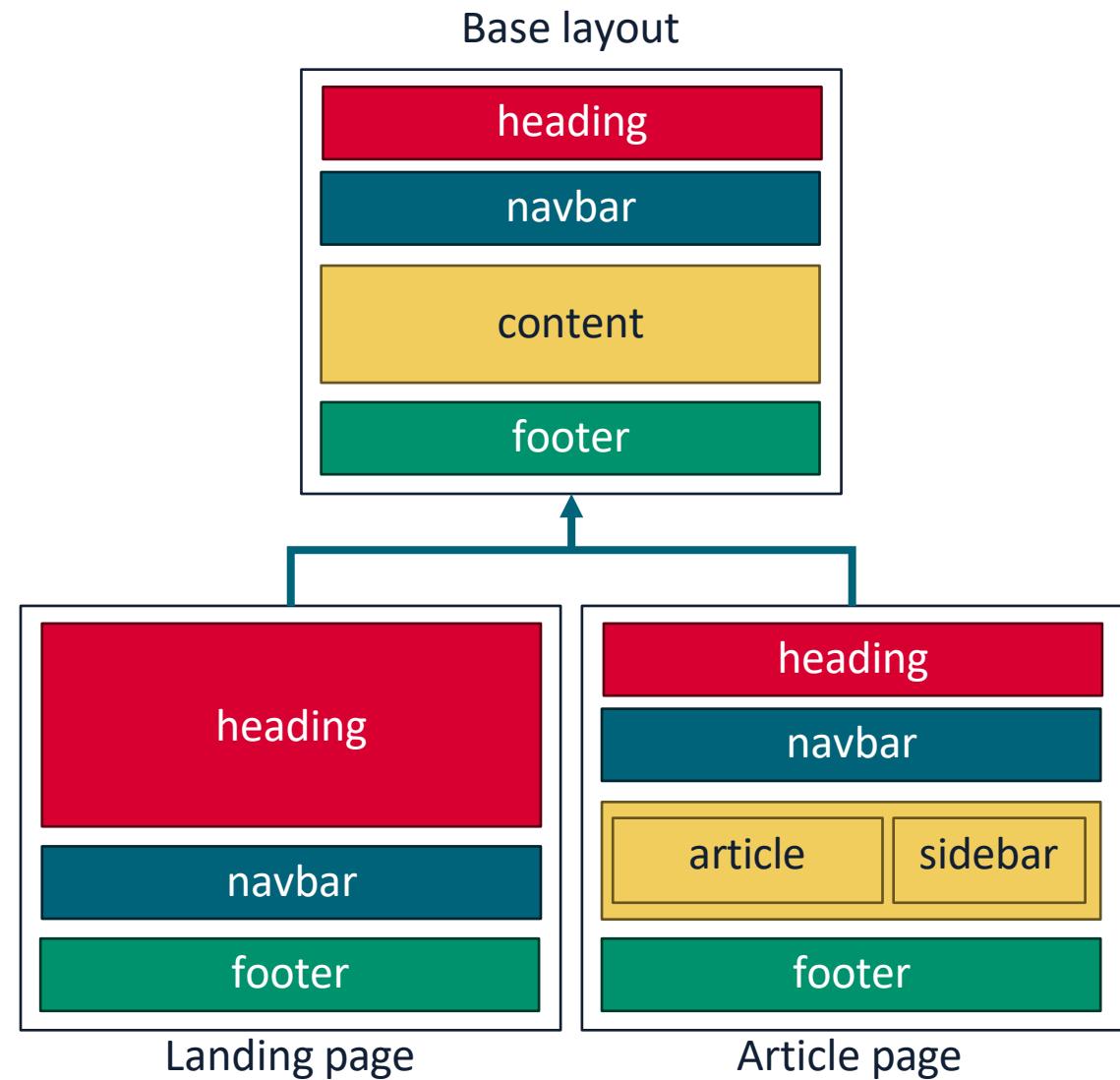
```
//- partials/footer.pug
footer#footer
  p Copyright (c) Web Tech
```



```
<!DOCTYPE html>
<html>
<head>
  <title>Hello Pug</title>
  <script src="/js/script.js">
  </script>
  <link rel="stylesheet"
        href="/style.css"/>
</head>
<body>
  <h1>Hello Pug!</h1>
  <footer id="footer">
    <p>Copyright (c) Web Tech</p>
  </footer>
</body>
</html>
```

TEMPLATING WITH PUG: INHERITANCE

- Quando si progettano applicazioni Web, è comune avere un layout comune condiviso da tutte le pagine Web
- Ogni pagina web può eventualmente sovrascrivere alcune parti del modello comune
- In Pug, tali scenari sono supportati con l'ereditarietà dei modelli



TEMPLATING WITH PUG: INHERITANCE

- Pug supporta l'ereditarietà dei modelli tramite il blocco ed estende le parole chiave
- Un blocco è un «blocco» del modello Pug che i modelli figlio possono sostituire
- I blocchi possono anche contenere contenuti predefiniti, se appropriato
- L'esempio a destra definisce tre blocchi: intestazione, contenuto e piè di pagina. L'intestazione e il piè di pagina contengono il contenuto predefinito.

```
//- base-layout.pug
doctype html
html
  head
    title Pug Inheritance
  body
    block heading
      h1 Pug Inheritance
    block content
    block footer
      footer This is the default footer.
```

TEMPLATING WITH PUG: INHERITANCE

- Un modello Pug può estendere altri modelli utilizzando la parola chiave `extends`
- I modelli che estendono altri modelli possono sovrascrivere alcuni dei modelli definiti dal modello principale
- Nell'esempio, i blocchi del contenuto e del piè di pagina vengono sostituiti
- Il blocco di intestazione non viene sostituito e visualizzerà il valore predefinito.

```
extends ./base-layout.pug

block content
  p The actual content of the homepage.

block footer
  footer This is a specialized footer.
```

TEMPLATING WITH PUG: INHERITANCE

```
//- base-layout.pug
doctype html
html
  head
    title Pug Inheritance
  body
    block heading
      h1 Pug Inheritance
    block content
    block footer
      footer This is the default footer.
```

```
extends ./base-layout.pug

block content
  p The actual content of the homepage.

block footer
  footer This is a specialized footer.
```

```
<!DOCTYPE html>
<html>
  <head>
    <title>Pug Inheritance </title>
  </head>
  <body>
    <h1>Pug Inheritance </h1>
    <p>
      The actual content of the
      homepage.
    </p>
    <footer>
      This is a specialized
      footer.
    </footer>
  </body>
</html>
```

TEMPLATING WITH PUG: LOCALS

- I modelli Pug possono anche eseguire il rendering del contenuto in base a un oggetto dati fornito (noto anche come «locals»), passato alle funzioni render() o renderFile()

```
import pug from "pug";

let html = pug.renderFile("./locals-example.pug", {
  "product": {
    "name": "Samsung S24",
    "description": "Smartphone"
  }
});
console.log(html);
```

- let footer = "a locally defined variable"

h1 #{product.name.toUpperCase()}
p Description: #{product.description}
footer #{footer}

TEMPLATING WITH PUG: LOCALS

- Il codice contenuto in #{ e } viene valutato, sottoposto a escape e memorizzato nel buffer nell'output

```
import pug from "pug";

let html = pug.renderFile("./locals.pug", {
  "product": {
    "name": "Samsung S24",
    "description": "Smartphone"
  }
});
console.log(html);
```

```
//- locals.pug
- let footer = "a nice string"

h1 #{product.name.toUpperCase()}
p Description: #{product.description}
footer #{footer}
```

```
<h1>SAMSUNG S24</h1>
<p>Description: Smartphone</p>
<footer>a nice string</footer>
```

TEMPLATING WITH PUG: CONDITIONALS

- Spesso, il rendering dei modelli viene eseguito in modo diverso a seconda di determinate condizioni
- A tal fine, Pug supporta un familiare costrutto if/else

```
let user = {"isAdmin": true, "name": "Brendan"};
let html = pug.renderFile("./conditionals.pug", {"user": user});
```

```
if !user
  a(href="/login") Please, authenticate yourself.
else if user.isAdmin
  p At your command, #{user.name}
else
  p Welcome back, #{user.name}
```

```
<p>At your command, Brendan</p>
```

TEMPLATING WITH PUG: ITERATIONS

- Un'attività comune quando si lavora con i modelli è l'iterazione su sequenze di dati

```
let items = [
  {"name": "Margherita", "price": 5.50}, {"name": "Marinara", "price": 5.00},
  {"name": "Capricciosa", "price": 6.50}
]
let html = pug.renderFile("./iteration.pug", {"items": items});
```

```
ul
  each item in items
    li #{item.name} - € #{item.price.toFixed(2)}
  else
    li No pizza is available at the moment!
```

```
<ul>
  <li>Margherita - € 5.50</li>
  <li>Marinara - € 5.00</li>
  <li>Capricciosa - € 6.50</li>
</ul>
```

OTHER TEMPLATE ENGINES

Pug è solo uno dei tanti motori di modelli disponibili. Altri motori di modelli ben noti includono:

- [SquirellyJS](#)
- [EJS](#) (formerly Jake): Syntax somewhat similar to JSP/PHP
- [Nunjucks](#)
- [LiquidJS](#)
- [Eta](#)
- [Haml](#) (Ruby)

PARSING REQUEST BODIES

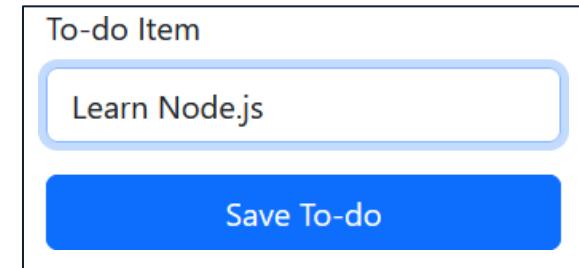
PARSING REQUEST BODIES

- Gli utenti compilano il modulo nella pagina dell'elenco delle cose da fare per salvare le nuove cose da fare
- Il modulo viene inviato utilizzando il metodo POST
- Abbiamo bisogno di analizzare il corpo, per ottenere i nomi dei parametri e il loro valore
- Non è così facile come potrebbe sembrare!

```
form(action="/todo" method="POST")
  label(for="todo") To-do Item
  input(type="text" id="todo" name="todo" required)
  button(type="submit") Save To-do
```

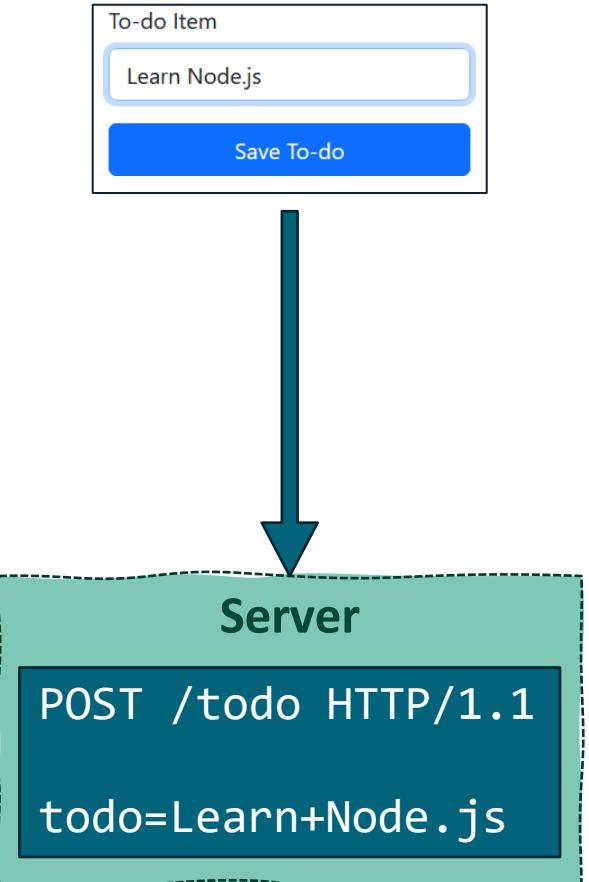
POST /todo HTTP/1.1

todo=Learn+Node.js



READING THE REQUEST BODY

- Il corpo della richiesta potrebbe non essere ancora disponibile durante l'elaborazione della richiesta. Forse l'utente ha un brutto connessione, o sta caricando alcuni file di grandi dimensioni, e il corpo sarà disponibile in un secondo momento
- L'oggetto richiesta (`http.IncomingMessage`) estende il flusso. Leggibile. Ciò significa che genera un evento di dati quando un nuovo blocco di dati diventa disponibile nel corpo della richiesta e un evento `End` quando non sono più presenti dati da utilizzare dal flusso.
- Per leggere il corpo della richiesta, dobbiamo operare in un modo asincrono, sfruttando questi due eventi



READING THE REQUEST BODY

- Some work is needed to read the body

```
function parseRequestBody(request){  
  return new Promise((resolve, reject) => {  
    let body = [];  
    request  
      .on('data', chunk => { body.push(chunk); })  
      .on('end', () => {  
        body = Buffer.concat(body).toString();  
        // at this point, `body` has the entire request body stored as a string  
        let data = parseRequestBodyString(body);  
        resolve(data);  
      });  
  });  
}
```

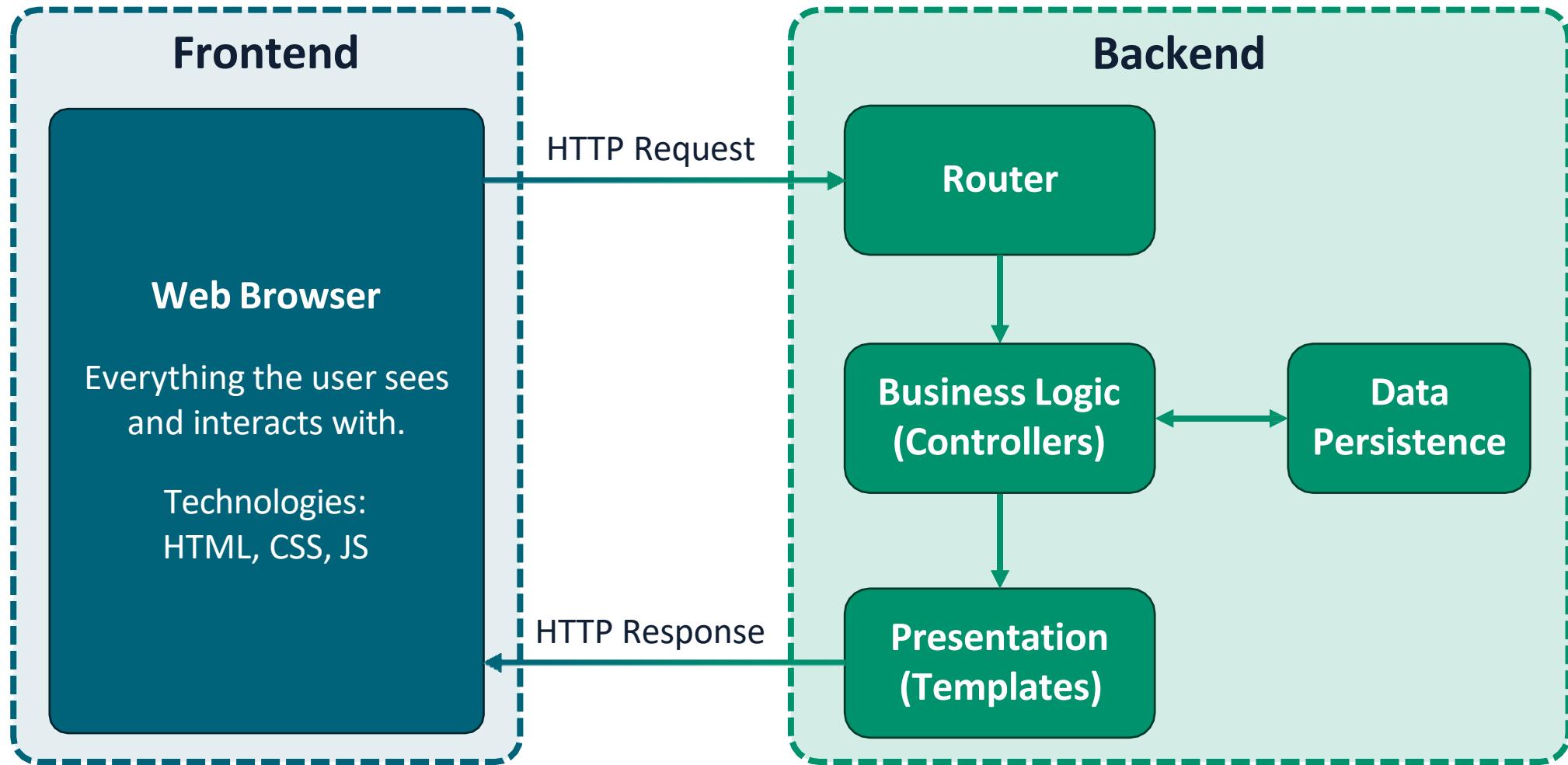
PARSING THE BODY

- Una volta letto il corpo come una stringa, è solo questione di dividerlo per ottenere i nomi e i valori dei parametri
- Ricordiamo che il corpo è una stringa della forma p1=v1&p2=v2&...

```
function parseRequestBodyString(body) {  
    let data = {};  
    let slices = body.split("&");  
    for (let slice of slices) {  
        let paramName = slice.split("=")[0];  
        paramName = decodeURIComponent(paramName.replace(/\+/g, " "));  
        let paramValue = slice.split("=")[1];  
        paramValue = decodeURIComponent(paramValue.replace(/\+/g, " "));  
        data[paramName] = paramValue;  
    }  
    return data;  
}
```

Recall that the body is URL encoded!
E.g.: «Prove that P=NP» is encoded as
«Prove+that+P%3DNP»!

ANATOMY OF A TYPICAL WEB APPLICATION



LET'S LOOK AT THE CODE

- Live demo time!
- We will take a look at the entire to-do list web app
- Source Code is available in the Course Materials on Teams
 - You should check the code out, and try to run (and debug) the web app



MINI-ASSIGNMENT

- Download and run the To-do List Web App we discussed in this lecture
- Feel free to try and add some new feature to our Node.js To-do app
 - For example, you may implement the possibility of deleting To-do items
 - Think about it and try to come up with a solution, using the tools and approaches we've seen so far!
- **Some hints:**
 - a possible way to do this is to add a new path in our app (e.g.: /delete)
 - you may pass an **id** of the to-do item to delete as a query parameter (e.g.: /delete?id=X)
 - you can specify the delete url for each to-do item when displaying the list in /todo, so that when a user clicks on a given to-do item, that item gets deleted. Alternatively, you may add a dedicated delete link for each to-do item!

REFERENCES

- **Web Templating**

Wiki page from the EduTech wiki hosted at the University of Geneva, CH

Available at [https://edutechwiki.unige.ch/en/Web templating](https://edutechwiki.unige.ch/en/Web_templating) and archived [here](#).

- **Single page apps in depth**

By Mikito Takada

Available at <http://singlepageappbook.com/> and on [GitHub](#)

Relevant parts: Templating: from data to HTML (direct link [here](#))

- **Node.js v.20.X (LTS) documentation**

Available at <https://nodejs.org/docs/latest-v20.x/api/index.html>

 In case you want to learn more about the built-in Node.js methods we used in this lecture.

- **A beginner's Guide to Pug**

By James Hibbard

Available at <https://www.sitepoint.com/a-beginners-guide-to-pug/> and archived [here](#).

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
WEB TECHNOLOGIES – LECTURE 12

IMPLEMENTING WEB APPS WITH NODE.JS: SESSION MANAGEMENT

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

<https://luistar.github.io>

<https://docenti.unina.it/luigiliberolucio.starace>

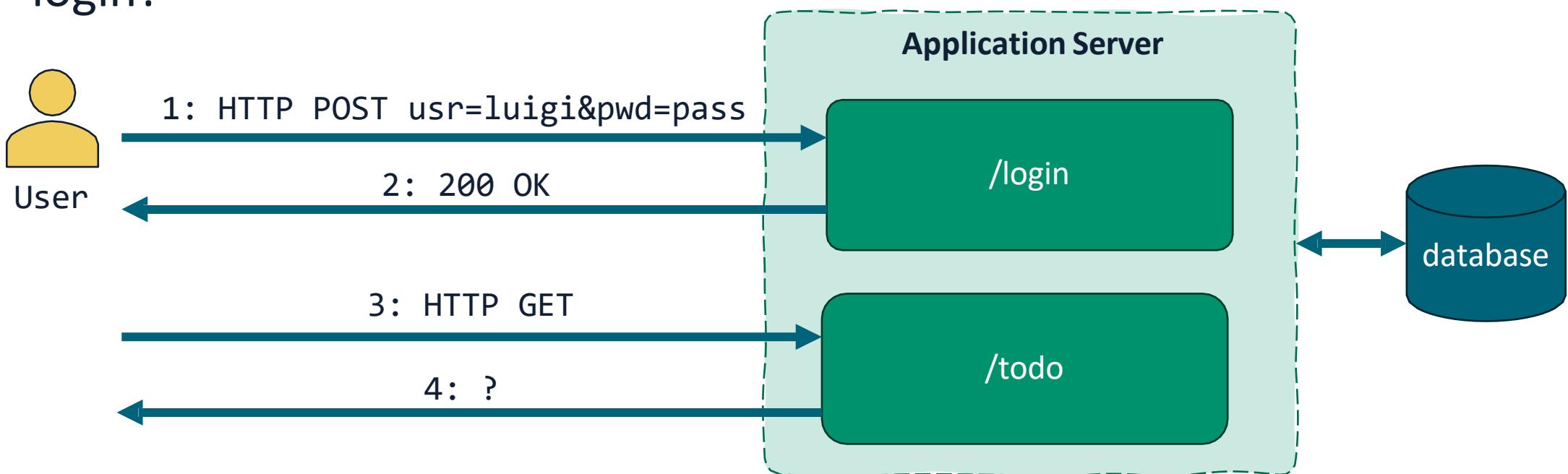
PREVIOUSLY, ON WEB TECHNOLOGIES

- We learned about server-side programming
- We implemented our first web application using Node.js
- It ain't much, but it's honest work :)
- Today, we'll add one more interesting feature to our app
 - Multiple users can **sign up**, **log in**, and manage their own To-do list
- We'll see soon enough that implementing these features requires a new concept: **session tracking**

SESSION TRACKING

HTTP IS STATELESS

- Una determinata richiesta, pronta all'uso, non contiene informazioni sulle richieste precedenti
- How can the server know that a user previously performed the login?



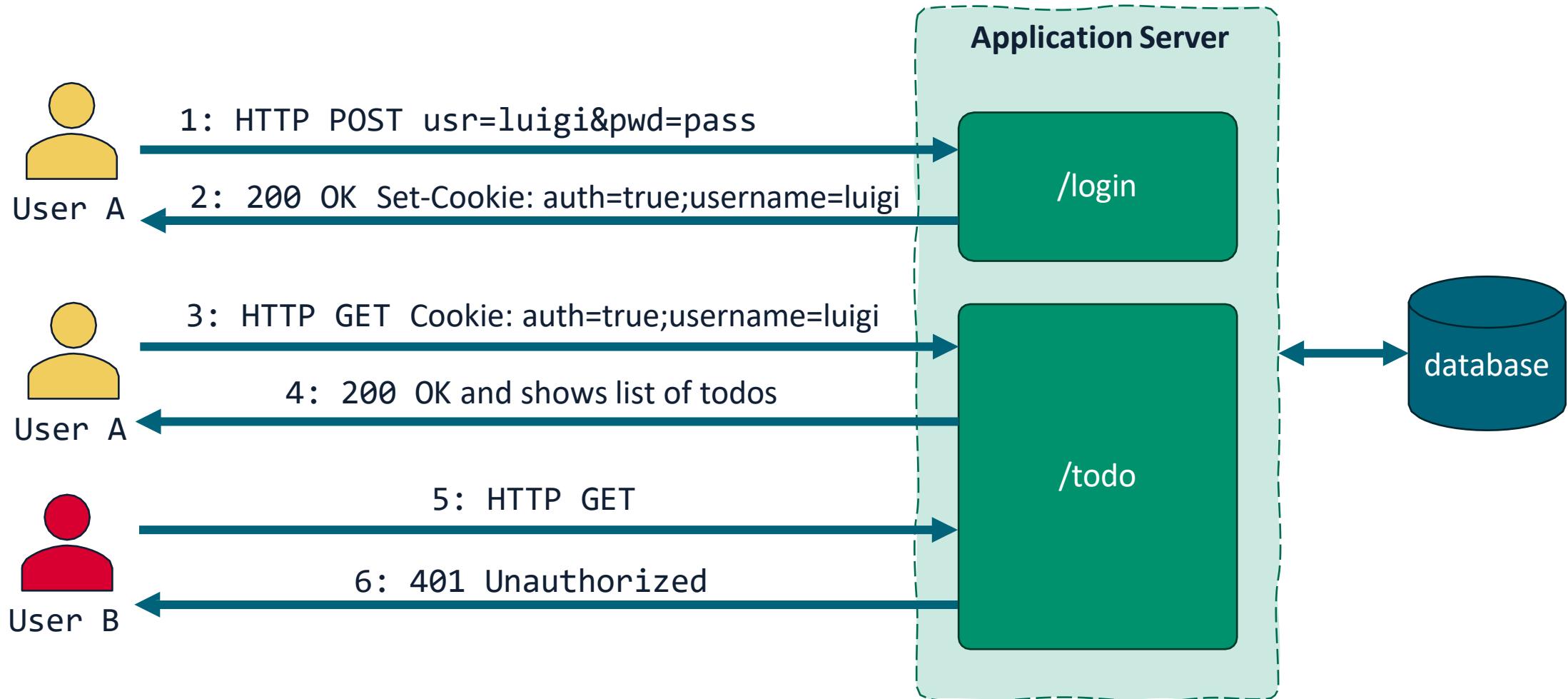
SESSION TRACKING

- Un aspetto cruciale dello sviluppo web
- L'obiettivo è mantenere le informazioni stateful sulle interazioni degli utenti
- su più richieste HTTP
- Consente ai server di «riconoscere» gli utenti e le loro azioni
- In questo modo, le risposte possono essere adattate a diversi utenti/situazioni!

A NAIIVE APPROACH: USING COOKIES

- Un modo che abbiamo visto per un server di «memorizzare» i dati in diverse richieste è l'impostazione dei cookie
- Idea:
- Gli utenti inviano una richiesta a una pagina dinamica, con nome utente e password come parametri
- La pagina dinamica verifica se le credenziali sono corrette
- Se sono corretti, il server imposta i cookie per tenere traccia dell'interazione, ad esempio: Set-Cookie: auth=true; Nome utente=Luigi
- Se non sono corretti, il server mostra una pagina di errore e non imposta alcun cookie
- Pagine dinamiche che richiedono il controllo dell'autenticazione per il cookie di autenticazione
- Se il cookie è impostato, mostrano il contenuto
- Se il cookie non è impostato, vengono reindirizzati alla pagina di accesso

NAIVE APPROACH: OVERVIEW



STORING SESSION DATA IN COOKIES

HANDLING LOGIN REQUESTS

1. Abbiamo un modulo di accesso con i campi usr e pwd, utilizzando POST
2. Analizziamo il corpo della richiesta come abbiamo fatto quando abbiamo salvato gli elementi To-do
3. Controlliamo se le credenziali sono corrette
 - In tal caso, reindirizza a «/»
 - Imposta due cookie
 - Uno memorizza il nome utente.
 - L'altro è solo un valore booleano
 - Entrambi scadono tra 1 ora

```
if(isAuthenticated){  
    response.writeHead(300, {  
        "Location": "/",  
        "Set-Cookie": [  
            `auth=true; max-age=${60*60}`,  
            `username=${user.username}; max-age=${60*60}`  
        ]  
    });  
    response.end();  
}
```

HANDLING LOGIN REQUESTS

- Se le credenziali non sono corrette, mostriamo la pagina di accesso e restituiamo un codice di errore 401.

```
if(!isAuthenticated){  
    let renderedContent = pug.renderFile("./templates/login.pug",  
        {"error": "Authentication failed. Check your credentials."});  
    response.writeHead(401, {"Content-Type": "text/html"});  
    response.end(renderedContent);  
}
```

RETRIEVING SESSION INFORMATION

- Vogliamo gestire le richieste in modo diverso a seconda che l'utente abbia effettuato l'accesso o meno.
- Ad esempio: se un utente non autorizzato tenta di visitare /todo, vogliamo servirgli una pagina di errore con un 401 (Non autorizzato). Se gli utenti registrati tentano di fare lo stesso, vogliamo che vedano il loro elenco di cose da fare.
- Per verificare se un utente ha effettuato l'accesso, dobbiamo controllare i cookie nelle richieste HTTP.

RETRIEVING SESSION INFORMATION

- We want something like the following example

```
function checkUserAuthentication(request){  
  let cookies = parseCookies(request);  
  if(cookies.auth){  
    return [true, cookies.username]  
  } else {  
    return [false, undefined];  
  }  
}
```

- Unfortunately, we're on our own again when parsing cookies!

PARSING COOKIES

```
function parseCookies(request){  
    const list = {};  
    const cookieHeader = request.headers?.cookie;  
    if (!cookieHeader) return list;  
  
    cookieHeader.split(`;`).forEach(function(cookie) {  
        let [ name, ...rest ] = cookie.split(`=`);
        name = name?.trim();  
        if (!name) return;  
        const value = rest.join(`=`).trim();  
        if (!value) return;  
        list[name] = decodeURIComponent(value);  
    });  
  
    return list;  
}
```

Recall that the Cookie header looks like this:

Cookie: auth=true; username=luigi

This code tries to be resilient when a cookie value contains “=”. Typically, cookies are URL-encoded, but it’s not required by specification

ACTING BASED ON SESSION STATUS

- Tutte le pagine della nostra app saranno influenzate dal fatto che l'utente sia autenticato o meno (ad esempio: la barra di navigazione mostrerà «Benvenuto, Utente» invece di «Accedi» quando un utente è autenticato)
- Vogliamo che queste informazioni siano disponibili a tutti i titolari del trattamento

ACTING BASED ON SESSION STATUS

- One way to do that, would be to modify the handleRequest method

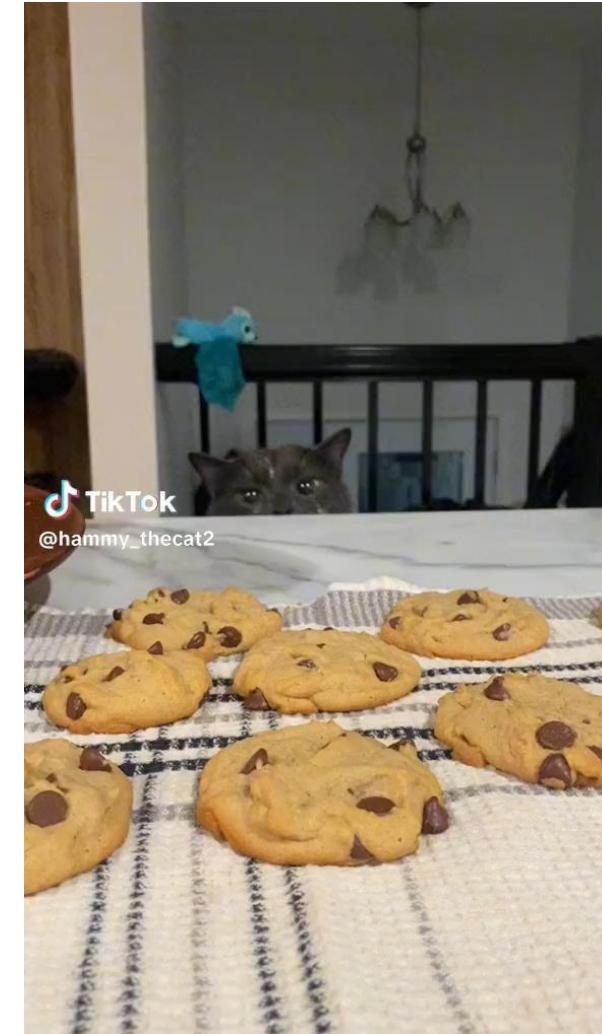
```
function handleRequest(request, response){  
  let [isUserAuthenticated, username] = checkUserAuthentication(request);  
  let context = {  
    isUserAuthenticated: isUserAuthenticated,  
    username: username  
  }  
  
  switch(request.url){  
    /* ... other routes ... */  
    case "/todo":  
      handleTodoListRequest(request, response, context); break;  
  }  
}
```

ACTING BASED ON SESSION STATUS

```
function handleTodoListRequest(request, response, context={}){
    if(!context.isUserAuthenticated){
        handleError(request, response, 401, "Unauthorized!");
    } else {
        switch(request.method){
            case "GET":
                handleTodoListRequestGet(request, response, context); break;
            case "POST":
                handleTodoListRequestPost(request, response, context); break;
            default:
                handleError(request, response, 405, "Unsupported method");
        }
    }
}
```

LET'S LOOK AT THE CODE

- Live demo time!
- We will take a look at the new version of the To-do list web app, with session data stored in **Cookies**
- Source Code is available in the Course Materials on Teams
 - You should check the code out, and try to run (and debug) the web app



SERVER-STORED SESSIONS

ISSUES WITH THE COOKIE APPROACH

- L'approccio dei cookie sembra funzionare ed è abbastanza semplice da implementare
- Purtroppo, è affetto da un problema critico
- I cookie sono interamente controllati dai clienti!
- È banale manipolarli, rendendo la nostra autenticazione del tutto inutile
- Un utente può cambiare il valore del proprio cookie «nome utente» con qualsiasi altro nome utente, oppure può impostare un cookie «nome utente» da solo, senza mai visitare la pagina di accesso.
- Questi problemi di manomissione possono essere risolti utilizzando i cookie firmati
- Il server firma i cookie utilizzando la sua chiave privata. Finché la chiave privata è sicura, il server è in grado di riconoscere i cookie manomessi

MORE ISSUES WITH COOKIES

- I problemi non sono limitati agli utenti che manomettono i dati dei cookie...
- I browser applicano limitazioni ai cookie per evitare problemi di prestazioni
- La dimensione dei cookie è generalmente limitata a 4096 byte
- Ogni host può generalmente memorizzare fino a 20-50 cookie per dominio
- Per saperne di più sui limiti del tuo browser, clicca qui:
- <http://browsercookielimits.iain.guru/>
- What if we need to store more session data?

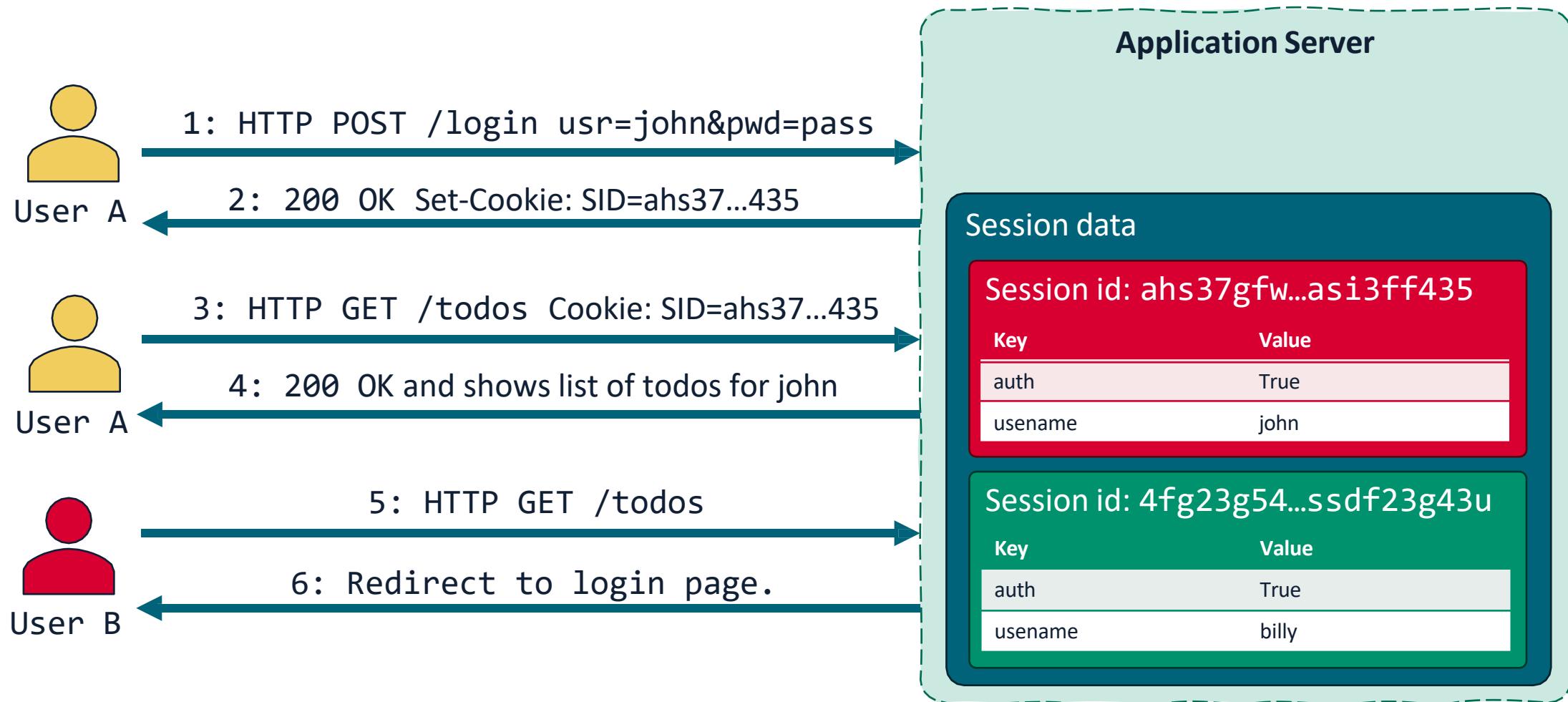
INTRODUCING SESSIONS

- So, how can we keep track of **sensitive** information from previous interactions and mitigate tampering issues?
- How can we store more data than Cookies allow us to?
- We can use the **Web Session** mechanism ([RFC 6265](#))

WEB SESSIONS: BASICS

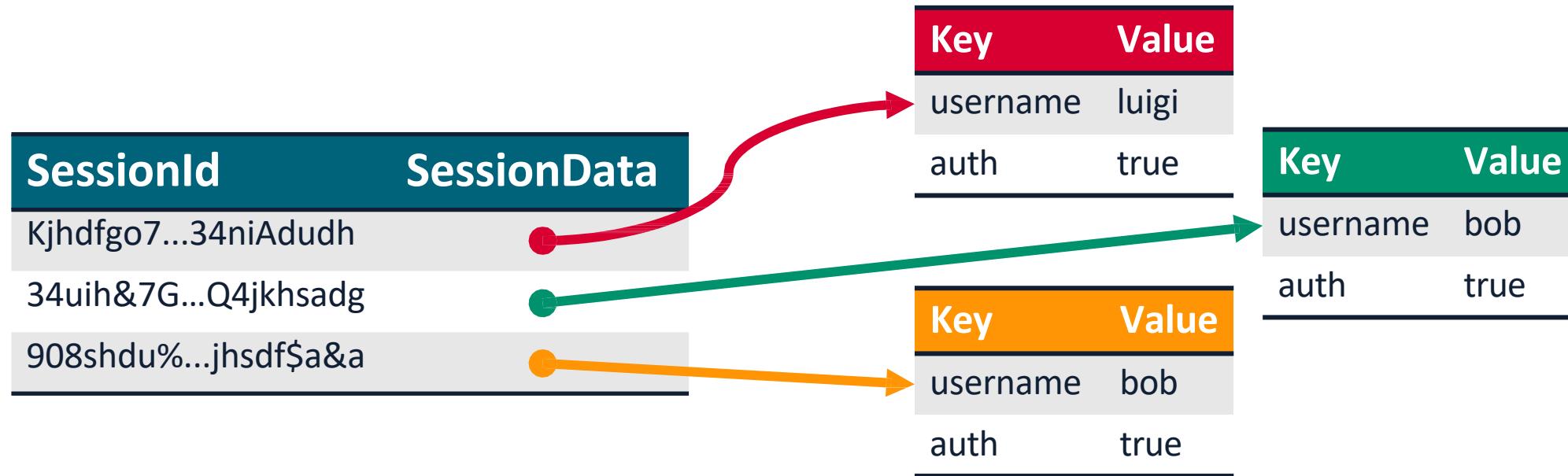
- Quando è necessario tenere traccia di alcune informazioni tra le richieste, il server Web crea una sessione per un determinato client
- Una sessione è una struttura dati che memorizza coppie chiave-valore, gestite e memorizzate sul server
- Una sessione è identificata da un ID di sessione univoco (noto anche come token di sessione)
- L'ID di sessione viene passato al client (generalmente come cookie)
- I dati della sessione rimangono sul server e il client non può modificarli

WEB SESSIONS: AUTHENTICATION EXAMPLE



IMPLEMENTING SESSIONS FROM SCRATCH

- Implementare un meccanismo di sessione di base in Node.js da zero è abbastanza semplice
- Possiamo utilizzare gli oggetti Map integrati in JavaScript. Una prima mappa associa ogni ID sessione a una sessione, che a sua volta è una mappa <chiave, valore>.



SESSIONS FROM SCRATCH IN NODE.JS

```
class Session {  
  
    constructor(){  
        this.sessionStore = new Map();  
    }  
  
    createSession(){  
        let sessionId = this.generateNewSessionId();  
        this.sessionStore.set(sessionId, new Map());  
        return sessionId;  
    }  
  
    storeSessionData(sessionId, key, value){  
        return this.getSessionById(sessionId)?.set(key, value);  
    }  
  
    /* other methods ... */  
}
```

See full example in Session.js

USING SESSIONS

- Quando un utente effettua l'accesso, creiamo una nuova sessione, memorizziamo i dati rilevanti al suo interno e passiamo l'ID di sessione come cookie

```
if(isAuthenticated){  
    //create a new session for the user  
    let sessionId = session.createSession();  
    session.storeSessionData(sessionId, "username", user.username);  
    session.storeSessionData(sessionId, "auth", true);  
    response.writeHead(300, {  
        "Location": "/",  
        "Set-Cookie": [  
            `sessionId=${sessionId}; max-age=${60*60}`  
        ]  
    });  
    response.end();  
}
```

USING SESSIONS

- When we need to access session data, we can

```
function handleRequest(request, response){  
  
    let userSession = session.getSessionFromRequest(request);  
    let context = {  
        isAuthenticated: userSession?.get("auth"),  
        username: userSession?.get("username")  
    }  
    /* rest of the code */  
}
```

```
getSessionFromRequest(request){  
    let requestSessionId = parseCookies(request)["sessionId"];  
    return this.getSessionById(requestSessionId);  
}
```

METHODS FOR WEB SESSION TRACKING

- L'utilizzo di un cookie per memorizzare l'ID di sessione è l'approccio più popolare
- Semplice, ampiamente supportato
- Esistono altri approcci:
- Riscrittura URL: l'ID sessione viene aggiunto a tutti gli URL come parametro di query
- Campi di modulo nascosti: gli ID sessione vengono memorizzati come campi modulo nascosti. Quando un utente invia un modulo, l'ID sessione viene inviato insieme agli altri dati

LET'S LOOK AT THE CODE

- Live demo time!
- We will now take a look at improved version of the To-do list web app, with proper server-stored session mechanisms
- Source Code is available in the Course Materials on Teams
 - You should check the code out, and try to run (and debug) the web app



REFERENCES

- **Introduction to Web Applications Development**

By Carles Mateu

Freely available on [archive.org](https://archive.org/details/introductiontow0000mate) under the GNU Free Documentation Licence

Relevant parts: Section 3.11 (Session monitoring). You can ignore the examples with Java Servlets

- **Using HTTP cookies**

MDN web docs

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies>

