03_01 Javascript nel browser

Importare javascript nell'head con **defer.**

Nel caso in cui siano usati async o defer, lo script va inserito nell'head.

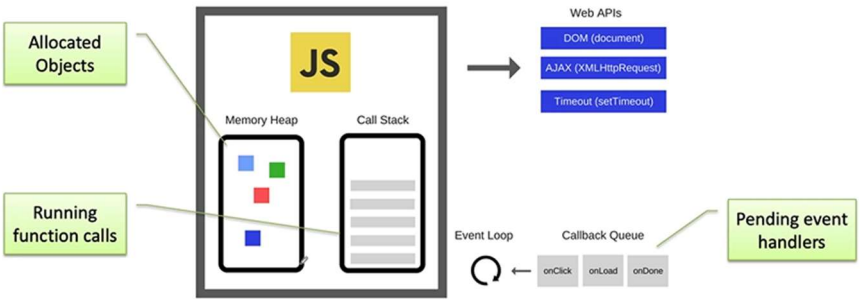
- ➔ *Script caricato in parallelo all'html ma esecuzione blocca il parsing e il rendering dell'html.* `<script async src="script.js"></script>`
- ➔ **DEFER**: Script viene caricato ad inizio pagina html ma viene eseguito dopo che la pagina html è pronta `<script defer src="script.js"></script>` (*preferred*)
    - Semplifica e migliora le cose perché dopo il caricamento del body, c'è solo l'esecuzione

Dove gira il codice js?

Il codice js gira all'interno del browser che è una **sendbox** → ambiente protetto rispetto al resto; quindi, una pagina web non può scrivere su un'altra pagina web o su disco tranne se vengono fornite determinate funzioni. Tutti gli script agiscono all'interno di una window → accedono solo a metodi e funzioni qui presenti:

- Libreria standard di javascript → tipi, async, awayt, promise…
- BOM → opzioni che il browser può mettere a disposizione alla pagina
    - Accedere ai preferiti
    - Barra di navigazione (andare avanti e indietro nella navigazione)
    - Location: url attuale
    - History: per accedere ad intera history
    - Screen: per accedere a proprietà di schermo e finestra
    - XMLHttpRequest: usare protocollo http
- DOM → funzionalità oggetto dalla pagina in cui ci troviamo

Eventi e Event loop

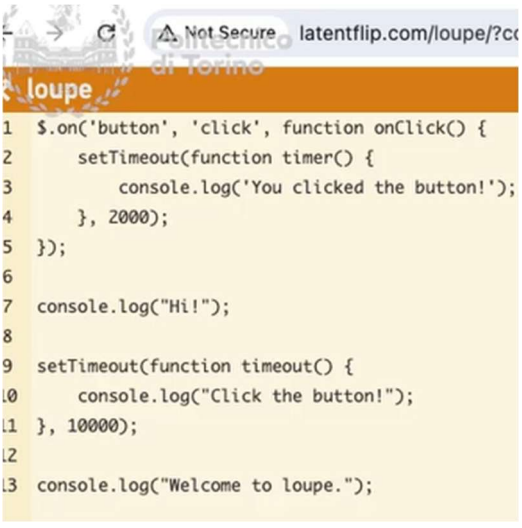


Tre tipi:

- Predefiniti
- Definiti da utente
- Ignorati

EventLoop:

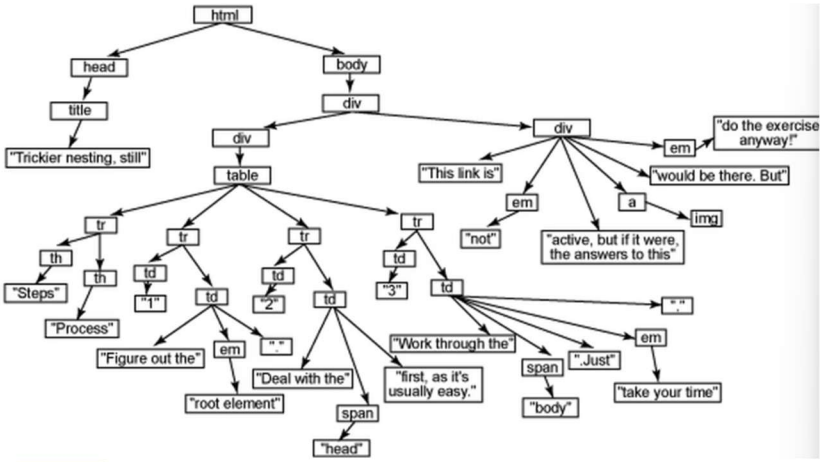
- heap di memoria: tiene oggetti
- call stack: per fare le chiamate una riga alla volta
    - inserisco in stack
    - eseguo
    - rimuovo da stack
- API offerte da execution environment, da node, altro
    - Sincrone: vanno in stack
    - Asincrone: entra in gioco l'event loop
        - Es: Timeout 5s
            - Preso in carico
            - Rimosso stack
            - Messo in callback queue
            - Callback da essere eseguita dopo 5s
            - Riga dopo dello stack viene eseguita
                - Continua ad andare avanti nello stack finché non deve eseguire qualcosa nella callback queue
            - 5secondi passati → chiama callback setTimeout che stampa a schermo
                - Messa su stack e eseguita



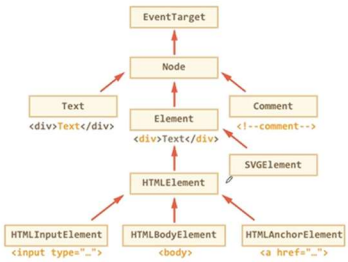
DOM:

Albero che rappresenta una **struttura** di una pagina WEB.

Attraverso javascript si può chiedere all'albero di prendere un determinato figlio, navigando sull'albero.



- **Document**: the document Node, the root of the tree
- **Element**: an HTML tag
- **Attr**: an attribute of a tag
- **Text**: the text content of an Element or Attr Node
- **Comment**: an HTML comment
- **DocumentType**: the Doctype declaration



NodeList: lista di nodi simile ad array

- `.length` property
- `.item(i)` , equivalent to `list[i]`
- `.entries()`, `.keys()`, `.values()` iterators
- `.forEach()` functional iteration
- `for…of` classical iteration

Manipolazione del DOM: attraverso document:

css:

- *.row*
- *.row > table*

- `document.getElementById(value)`
  - Returns the Node with the attribute id=value
- `document.getElementsByTagName(value)`
  - Returns the NodeList of all elements with the specified tag name (e.g., 'div')
- `document.getElementsByClassName(value)`
  - Returns the NodeList of all elements with attribute class=value (e.g., 'col-8')
- `document.querySelector(css)`
  - Returns the first Node element that matches the CSS selector syntax
- `document.querySelectorAll(css)`
  - Returns the NodeList of all elements that match the CSS selector syntax

I metodi per manipolazione, si possono **concatenare**:

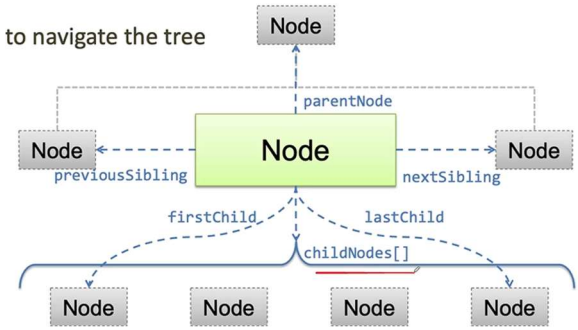
```
let main = document.getElementById('main');
let articleText = main.getElementsByTagName('p');
```

esempi:

```
<!DOCTYPE html>
<html>
<head></head>
<body>
<div id="foo"></div>
<div class="bold"></div>
<div class="bold color"></div>
<script>
 document.getElementById('foo');
 document.querySelector('#foo');
 document.querySelectorAll('.bold');
 document.querySelectorAll('.color');
 document.querySelectorAll('.bold, .color');
</script>
</body>
</html>
```

```
<div id="foo"></div>
<div id="foo"></div>
▶ NodeList(2) [div.bold, div.bold.color]
▶ NodeList [div.bold.color]
▶ NodeList(2) [div.bold, div.bold.color]
>
```

Navigare l'albero:



Quando si ha un nodo, gli attributi di un elemento diventano proprietà, body, dell'oggetto. Se si ha un body id → proprietà di body id.

```
– <body id="page">
– DOM object: document.body.id="page"
– Also: document["body"]["id"]

– <input id="input" type="checkbox" checked />
– DOM object: input.checked  // boolean
```

Gestione degli attributi:

- `elem.hasAttribute(name)`
  - check the existence of the attribute
- `elem.getAttribute(name)`
  - check the value, like `elem[name]`
- `elem.setAttribute(name, value)`
  - set the value of the attribute
- `elem.removeAttribute(name)`
  - delete the attribute
- `elem.attributes`
  - collection of all attributes
- `elem.matches(css)`
  - Check whether the element matches the CSS selector

Creare elementi:

- document.createElement(tag) → crea elemento del tipo tag
- document.createTextNode(text) → nodo testuale

```
let div = document.createElement('div');
div.className = "alert alert-success";
div.innerText = "Hi there! You've read an important message.";
```

```
<div class="alert alert-success">
Hi there! You've read an important message.
</div>
```

quando si crea un elemento, **bisogna inserirlo nel DOM**:

- document.body.appendChild(div) → aggiunge in coda al body

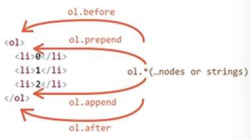
metodi per elementi html:

- aggiungere un figlio
- aggiungere prima di un nodo
- rimpiazzare un nodo con uno nuovo.

- `parentElem.appendChild(node)`
- `parentElem.insertBefore(node, nextSibling)`
- `parentElem.replaceChild(node, oldChild)`

Metodi applicabili su elementi ma anche su **nodi**:

- `node.append(…nodes or strings)`
- `node.prepend(…nodes or strings)`
- `node.before(…nodes or strings)`
- `node.after(…nodes or strings)`
- `node.replaceWith(…nodes or strings)`



Recuperare contenuto presente in html in maniera testuale → **.innerHTML**:

```
<div class="alert alert-success">
    <strong>Hi there!</strong> You've read an important message.
</div>
```

```
div.innerHTML  // "<strong>Hi there!</strong> You've read an important message."
```

permette di impostare o recuperare l'html di un nodo o di un elemento in forma testuale.

Permette di aggiungere elementi o testo.

Inserting New Content

- `elem.innerHTML = "html fragment"`
- `elem.insertAdjacentHTML(where, HTML)`
  - where = "beforebegin" | "afterbegin" | "beforeend" | "afterend"
  - HTML = HTML fragment with the nodes to insert
- `elem.insertAdjacentText(where, text)`
- `elem.insertAdjacentElement(where, elem)`

Permette di clonare: utile per replicare e poi modificare

- `elem.cloneNode(true)`
  - Recursive (deep) copy of the element, including its attributes, sub-elements, ...
- `elem.cloneNode(false)`
  - Shallow copy (will not contain the children)
- Useful to "replicate" some part of the document

Si può cambiare lo stile attraverso attirbuto classList

- To add/remove a single class use `classList`
  - `elem.classList.add("col-3")` add a class
  - `elem.classList.remove("col-3")` remove a class
  - `elem.classList.toggle("col-3")` if the class exists, it removes it, otherwise it adds it
  - `elem.classList.contains("col-3")` returns true/false checking if the element contains the class

Esiste anche il metodo elem.style:

- `elem.style` contains all CSS properties
  - Example: hide element
    `elem.style.display="none"`
    (equivalent to CSS declaration `display:none`)
- `getComputedStyle(element[,pseudo])`
  - `element`: selects the element of which we want to read the value
  - `pseudo`: a pseudo element, if necessary
- For properties that use more words the camelCase is used (`backgroundColor`, `zIndex`... instead of `background-color` ...)

EVENT HANDLING

La modalità di programmazione è **event-driven** → ogni cosa è causata da un evento.

Gli eventi sono determinati da:

- **target**: l'elemento DOM che genera l'evento
- **type**: il tipo di evento generato

addEventListener

```
window.addEventListener('load', (event) => {
  //window loaded
})
```

```
const link = document.getElementById('my-link')
link.addEventListener('mousedown', event => {
  // mouse button pressed
  console.log(event.button) //0=left, 2=right
})
```

Esistono varie categorie di eventi:

- User Interface events (load, resize, scroll, etc.)
- Focus/blur events
- Mouse events (click, dblclick, mouseover, drag, etc.)
- Keyboard events (keyup, etc.)
- Form events (submit, change, input)
- Mutation events (DOMContentLoaded, etc.)
- HTML5 events (invalid, loadeddata, etc.)
- CSS events (animations etc.)

Molti eventi hanno un comportamento di default (click su link → vai ad url). Si può modifcare il comportamento di default rimpiazzandolo con uno proprio: `event.preventDefault()`

Eventi relativi al ciclo di vita della pagina HTML:

- DOMContentLoaded (definito su documento)
  - Il browser carica tutto l'html, e l'albero del DOM è pronto
  - Risorse esterne non sono caricate
- Load
  - Il browser è finito caricando tutto le risorse esterne
- beforeunload/unload → da usare a proprio rischio e pericola perché non sempre funzionano