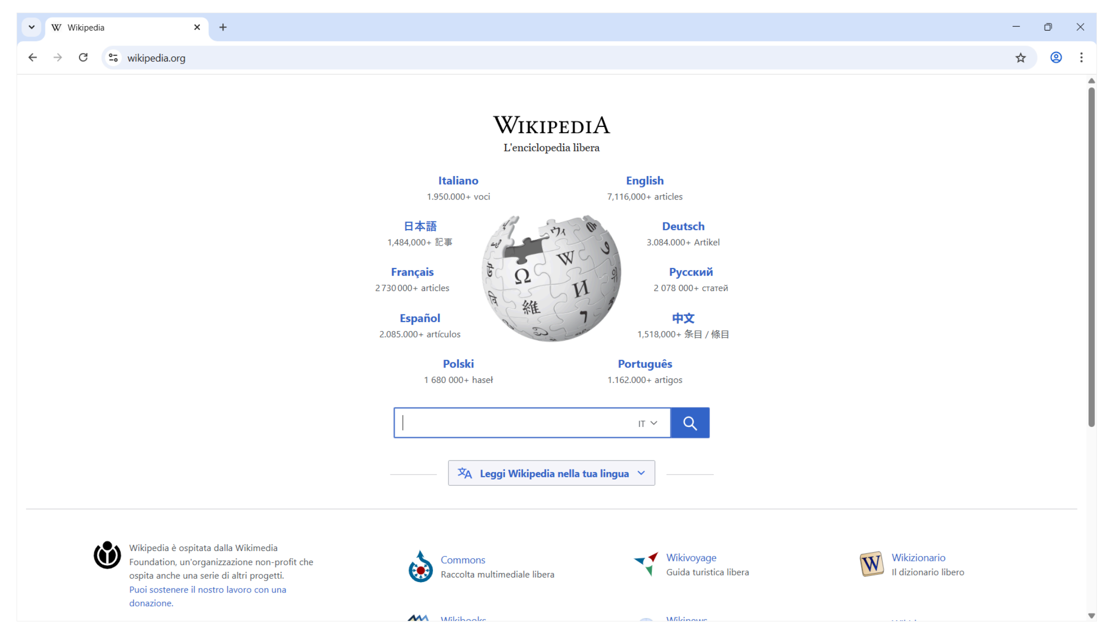
CSS, DOM ed eventi

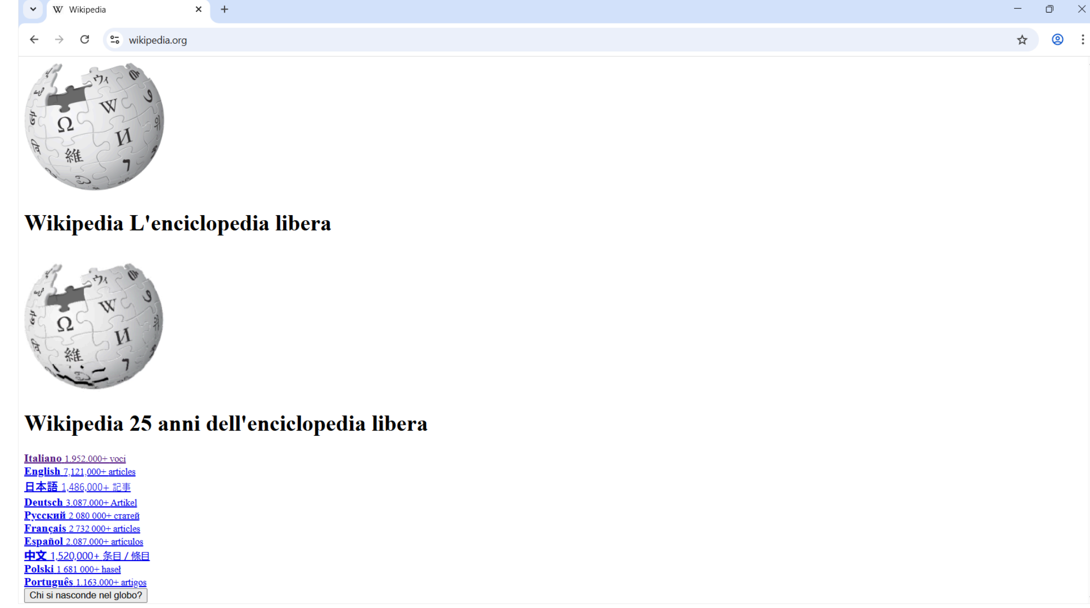
Niccolò Maltoni
niccolo.maltoni@diennea.com

A cosa serve il CSS?

HTML è obbligatorio per una pagina web, CSS è opzionale, ma ampiamente consigliato!



Con CSS



Senza CSS

La differenza tra le due immagini è evidente.

Possiamo però capire subito che una pagina senza CSS funziona comunque al 100%, ma sarà semplicemente un insieme di blocchi uno sotto l'altro.

Cosa sono i CSS?

CSS è acronimo di **Cascading Style Sheets** (*fogli di stile in cascata*).

Nel 1996, il web era caotico: ogni browser (Netscape Navigator, Internet Explorer) aggiungeva tag proprietari per distinguersi dai competitor.

Il risultato? Un sito appariva diverso (o si rompeva completamente) a seconda del browser.

Il **W3C** intervenne con CSS per **separare il contenuto (HTML) dalla presentazione (CSS)**.

Adesso:

- Uno stesso sito CSS funziona uguale su Chrome, Firefox, Safari, Edge…
- Puoi modificare l’aspetto senza toccare l’HTML.
- Il CSS è un linguaggio semplice e universale.

CSS: Sintassi

Una **regola CSS** è composta da un **selettore** e un blocco di **dichiarazioni**.

```
h1 {
  color: blue;
  font-size: 12px;
}
```

- **Selettore** (`h1`): a *quale* elemento/i vogliamo applicare determinati stili.
- **Dichiarazione** (`color: blue;`): *cosa* cambiamo. Composta da **proprietà** e **valore**.
- Terminare sempre con `;`.

Selettori CSS

I selettori indicano a **quali elementi** HTML si vogliono applicare determinate regole di stile.

Selettore di elemento: tutti gli elementi con un certo tag.

```css
p {
  color: blue;
}
```

Selettore di classe: tutti gli elementi con una certa classe (attributo `class`).

```css
.my-class {
  background: yellow;
}
```

Selettore di ID: un elemento specifico con un certo ID (attributo `id`).

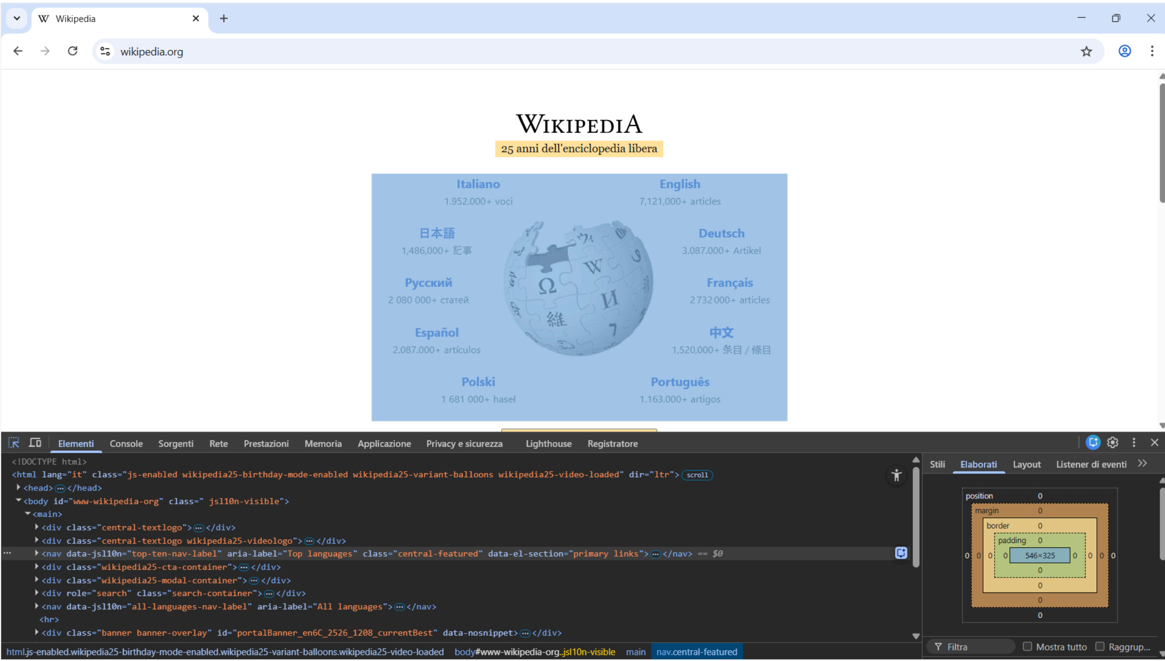
```css
#my-title {
  font-size: 2rem;
}
```

Nota: generalmente usi le **classi** per raggruppare stili, e gli **ID** solo quando serve un elemento unico.

Box Model

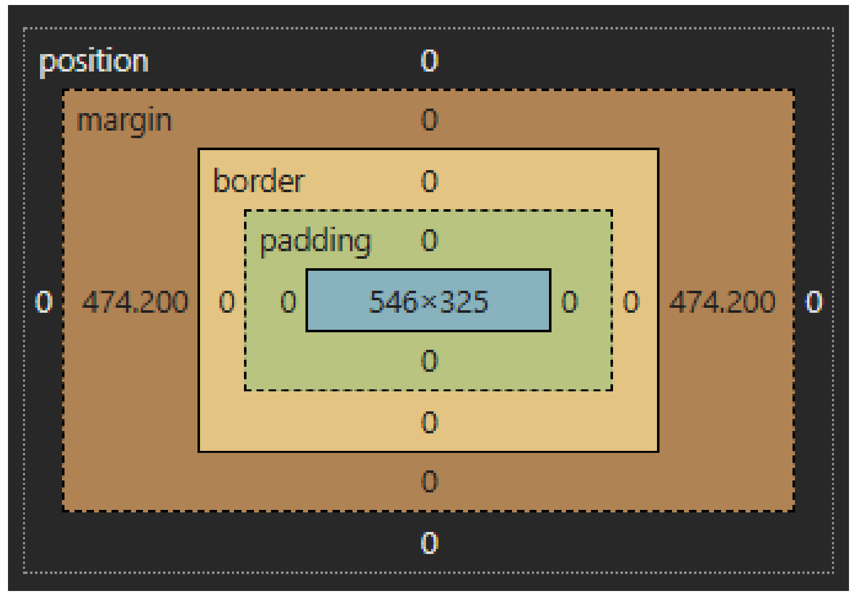
Abbiamo visto che senza CSS la pagina è un insieme di blocchi uno sotto l'altro. In particolare, i singoli elementi sono di fatto dei box rettangolari che occupano un certo spazio nello schermo.

Questo modello è detto **box model**. Ma quali proprietà hanno questi box?



Se si aprono nuovamente i **DevTools** su una pagina e ci si sposta sulla tab **Elaborati**, si vede una rappresentazione interessante.

Box Model



Già da questa vista, si può notare che ci sono quattro componenti fondamentali:

- **Content**: è l'area occupata dal contenuto vero e proprio
- **Padding**: è lo spazio vuoto compreso tra l'area del contenuto e il bordo dell'elemento
- **Border**: è una linea che circonda la zona del padding e l'area del contenuto
- **Margin**: è lo spazio vuoto compreso tra l'elemento gli elementi adiacenti

Proprietà `display`

Ogni elemento ha un comportamento predefinito rispetto a come occupa lo spazio:

- `<h1>`, `<p>`, `<div>` sono **elementi blocco** → occupano l'intera larghezza disponibile e vanno a capo.
- `<a>`, `<span>`, `<em>` sono **elementi inline** → si allineano orizzontalmente, accanto al testo.

Con la proprietà `display`, possiamo **modificare** questo comportamento predefinito:

`block`: occupa intera larghezza, va a capo.

```
<h1>Titolo</h1>       <!-- block per default -->
<p>Paragrafo</p>      <!-- block per default -->
```

`inline`: accanto ad altri, larghezza del contenuto.

```
<a href="#">Link</a>  <!-- inline per default -->
<em>Enfasi</em>       <!-- inline per default -->
```

`inline-block`: accanto ad altri **ma** accetta `width`, `height`, margini.

```
<button>Clicca</button>  <!-- inline-block tipico -->
```

`none`: elemento invisibile, non occupa spazio.

```
<div class="hidden"></div>  <!-- Sparisce completamente -->
```

Combinatori CSS

I primi due combinatori selezionano **figli** (*children*, elementi a un livello inferiore).

Selettore di discendenti

Seleziona **qualsiasi elemento** `p` dentro `div`.

index.html

```html
<div>
  <p>Primo livello</p>
  <section>
    <p>Secondo livello</p>
  </section>
</div>
```

style.css

```css
div p {
  color: red;
}
```

Risultato: **entrambi** i paragrafi diventano rossi.

Selettore di figli (>)

Seleziona **solo i figli diretti** di `div`.

index.html

```html
<div>
  <p>Figlio diretto</p>
  <section>
    <p>Figlio di section</p>
  </section>
</div>
```

style.css

```css
div > p {
  color: blue;
}
```

Risultato: **solo il primo** paragrafo diventa blu.

Utili per evitare di aggiungere classi ovunque!

Combinatori CSS

Gli ultimi due combinatori selezionano **fratelli** (*sibling*, elementi allo stesso livello).

Fratello adiacente (+)

Seleziona il **primo fratello** che segue un elemento.

index.html

```html
<h2>Titolo</h2>
<p>Primo paragrafo</p>
<p>Secondo paragrafo</p>
```

style.css

```css
h2 + p {
  font-weight: bold;
}
```

Risultato: **solo il primo** paragrafo è in grassetto.

Fratello generico (~)

Seleziona **tutti i fratelli** che seguono un elemento.

index.html

```html
<h2>Titolo</h2>
<p>Primo paragrafo</p>
<p>Secondo paragrafo</p>
```

style.css

```css
h2 ~ p {
  color: gray;
}
```

Risultato: **entrambi** i paragrafi diventano grigi.

Molto utili con JavaScript quando usi `querySelector()`!

Come includere CSS in HTML

Ci sono tre modi per includere CSS in una pagina HTML.

1. Inline:

```html
<h1 style="background: red;">Hello World!</h1>
```

2. Nel tag `<style>` nell'`<head>`:

```html
<head>
  <style>
    h1 {
      background: red;
    }
  </style>
</head>
```

3. File CSS esterno:

```html
<head>
  <link rel="stylesheet" href="style.css"/>
</head>
```

Quali regole vincono?

Quando scriviamo CSS, spesso **più regole si applicano allo stesso elemento**. Quale regola vince?

Cascata

Le regole **più basse** nel file sovrascrivono quelle **sopra**.

```
p {
  color: red;
}
p {
  color: blue;
}
/* Vince: blu */
```

È come sovrascrivere una variabile: l'ultimo valore è quello valido.

Specificità

Alcuni selettori hanno **priorità più alta**, indipendentemente dall'ordine.

```
p { /* peso: 1 */
  color: red;
}
.highlight { /* peso: 10 → vince */
  color: blue;
}
#intro { /* peso: 100 → vince sempre */
  color: green;
}
```

Ranking: elemento (1) < classe (10) < ID (100) < inline style (1000).

Inheritance

Alcune proprietà si **ereditano** dai genitori, altre no.

Si eredita (proprietà di testo/font):

```
body {
  color: black;
}
```

Tutti i figli e nipoti hanno testo nero, a meno che non lo sovrascrivano.

```
<body>
  <p>Testo nero</p>
  <div>
    <span>Testo nero</span>
  </div>
</body>
```

Non si eredita (proprietà di spacing):

```
body {
  margin: 10px
}
```

I figli hanno il loro `margin` predefinito, non quello del padre.

```
<body>
  <p>Margin default (non 10px)</p>
  <div>
    <span>Margin default</span>
  </div>
</body>
```

Approfondimenti: altre risorse CSS

Abbiamo visto i concetti fondamentali di CSS necessari per lavorare con JavaScript e il DOM.

CSS è un linguaggio ricco di proprietà, selettori avanzati (pseudo-classi, pseudo-elementi), e tecniche di layout. Non è compito di questo modulo coprirle tutte.

Quando ci servirà un selettore, una proprietà, o una tecnica che non abbiamo visto, possiamo affidarci a:

MDN Web Docs W3C CSS W3Schools

Esercizio: il ricettario con CSS

Consiglio

Riprendiamo il ricettario e diamogli l'aspetto di una card semplice e ordinata.

Possiamo usare questi valori:

- circondiamo la ricetta con una card con sfondo bianco, bordi arrotondati e padding di 24px, da cui traspare uno sfondo grigio chiaro per la pagina;
- applichiamo i bordi arrotondati anche all'immagine e facciamola adattare alla card (limitiamo la dimensione della card a 720px e centriamola orizzontalmente);
- diamo una dimensione di 32px al titolo, 22px ai sottotitoli e 18px alla descrizione;
- lasciamo 12px di margine sotto il titolo e 24px di padding all'interno della card;
- lasciamo 8px di margine sotto ogni ingrediente e passaggio, e indentiamo le liste di 24px con un padding.

Il DOM: Document Object Model

Il **DOM** (*“Document Object Model”*) è l’interfaccia che permette a JavaScript di **interagire** con il codice HTML.

- Ogni tag HTML è visto come un **oggetto** `Element`.
- Il documento è rappresentato come **albero** di nodi.
- La radice è `<html>`, con `<head>` e `<body>` come figli.

Albero del DOM

```
<html>
├─ <head>
│   └─ <title>
└─ <body>
    ├─ <h1>
    ├─ <p>
    └─ <div>
        └─ <button>
```

In JavaScript, possiamo **scorrere** questo albero e **modificare** gli elementi.

Ma cos’è un oggetto?

Un **oggetto** è una collezione di dati in relazione tra loro e funzionalità correlate.

Quando sono dentro un oggetto, variabili e funzioni prendono nomi specifici:

- **Proprietà**: i dati dell’oggetto
- **Metodi**: le funzioni (azioni) dell’oggetto

Quasi tutto quello che utilizziamo in JavaScript è un oggetto!

Creare un oggetto letterale

Per creare un oggetto utilizziamo le **parentesi graffe**:

```
const nomeOggetto = {};
```

Andiamo ad inserire proprietà e metodi come **coppie chiave-valore** separate da virgola:

```
const studente = {
  nome: 'Sara',
  eta: 28,
  corso: 'JavaScript'
};
```

Nota: non usiamo `=` ma `:` per assegnare i valori, e separiamo i membri con `,` e non `;`

Scopo degli oggetti

Gli oggetti servono per descrivere dati **complessi** all'interno della nostra applicazione.

Supponiamo di dover creare una rubrica: avremo bisogno di gestire dei contatti.

```
const contatto = {
  nome: ['Bob', 'Smith'],
  eta: 32,
  email: 'bob.smith@email.it',
  citta: 'Roma'
};
```

Vantaggi:

- Trasferire dati strutturati in modo organizzato
- Individuare i singoli elementi per nome invece che per posizione

La dot notation

Accediamo alle proprietà e ai metodi dell'oggetto usando la **notazione a punto**.

```
const studente = {
  nome: 'Sara',
  eta: 28,
  corso: 'JavaScript'
};

console.log(studente.nome);   // 'Sara'
console.log(studente.eta);    // 28
console.log(studente.corso);  // 'JavaScript'
```

Il nome dell'oggetto (`studente`) funge da **spazio dei nomi**: deve essere inserito prima per accedere a qualsiasi cosa all'interno dell'oggetto.

Oggetti come proprietà

Un oggetto può avere proprietà che sono a loro volta oggetti:

```
const contatto = {
  anagrafica: {
    nome: 'Bob',
    cognome: 'Smith'
  },
  eta: 32
};
```

Possiamo accedere ai contenuti sempre tramite **dot notation**:

```
console.log(contatto.anagrafica.nome);     // 'Bob'
console.log(contatto.anagrafica.cognome);  // 'Smith'
```

Modificare oggetti

Una volta creato, possiamo **modificare** i valori delle proprietà:

```
const studente = {
  nome: 'Sara',
  eta: 28
};

studente.nome = 'Maria';  // Modifica
studente.eta = 30;        // Modifica
```

È anche possibile **aggiungere** nuove proprietà:

```
studente.corso = 'JavaScript';  // Aggiunta
studente.email = 'm.rossi@email.it';  // Aggiunta

console.log(studente);
// { nome: 'Maria', eta: 30, corso: 'JavaScript', email: 'm.rossi@email.it' }
```

Array in JavaScript

Un **array** è una lista ordinata di elementi.

In JavaScript, gli array sono oggetti speciali che:

- Hanno tutte proprietà descritte da **numeri** (indici)
- La numerazione inizia da **0**
- Possono **modificare la loro dimensione** dinamicamente

```
const ingredienti = ['Pasta', 'Uova', 'Guanciale'];
```

Gli array sono molto utili per gestire **elenchi** e **sequenze** di dati.

Accedere agli elementi di un array

Ogni elemento ha un **indice** che parte da 0:

```
const ingredienti = ['Pasta', 'Uova', 'Guanciale'];

const primo = ingredienti[0];  // 'Pasta'
const secondo = ingredienti[1];  // 'Uova'
const terzo = ingredienti[2];  // 'Guanciale'
```

Possiamo sapere quanti elementi ci sono con la proprietà `length`:

```
const quanti = ingredienti.length;  // 3
```

Possiamo anche **modificare** gli elementi:

```
ingredienti[0] = 'Spaghetti';
console.log(ingredienti);  // ['Spaghetti', 'Uova', 'Guanciale']
```

Bracket notation

Abbiamo visto che gli array usano le **parentesi quadre** con numeri per accedere agli elementi.

Allo stesso modo, possiamo accedere alle proprietà di un oggetto con le **parentesi quadre** e il nome della proprietà:

Dot notation:

```
studente.nome;
studente.eta;
```

Bracket notation:

```
studente['nome'];
studente['eta'];
```

Sono equivalenti! Gli oggetti sono anche chiamati **array associativi** perché funzionano come array con chiavi di testo invece che di numeri.

Dot notation vs bracket notation

Solitamente preferiamo la **dot notation**, ma la bracket notation permette cose altrimenti impossibili:

```
const studente = {
  nome: 'Sara',
  eta: 28
};

let propName = 'nome';
console.log(studente[propName]); // 'Sara'

propName = 'eta';
console.log(studente[propName]); // 28
```

Usiamo bracket notation quando:

- La proprietà ha un nome con spazi o caratteri speciali
- Vogliamo usare variabili per accedere dinamicamente alle proprietà

Optional chaining

Quando una proprietà intermedia può non esistere, l'accesso diretto può generare errore.

Con **optional chaining** (`?.`) interrompiamo la lettura in modo sicuro: se un passaggio è `null` o `undefined`, l'espressione restituisce `undefined`.

```
const utente = {
  profilo: {
    indirizzo: {
      citta: 'Roma'
    }
  }
};

console.log(utente.profilo?.indirizzo?.citta);  // 'Roma'
console.log(utente.profilo?.lavoro?.ruolo);      // undefined (nessun errore)
```

Usiamo `?.` quando leggiamo dati annidati da oggetti, API o DOM e non siamo sicuri che tutti i livelli esistano.

Possiamo usare l'optional chaining anche in bracket notation (`?.[]`) e per l'invocazione di metodi (`?.()`)

```
utente.profilo?.['citta']       // bracket notation sicura
utente.saluta?.()               // chiama saluta() solo se esiste
```

Nullish coalescing

L'operatore **nullish coalescing** (`??`) ci aiuta a impostare un valore di default.

- Usa il valore a destra solo se a sinistra c'è `null` o `undefined`
- Mantiene valori validi come `0`, `''`, `false`

```
const nome = '';
const punteggio = 0;
const tema = null;

console.log(nome ?? 'Anonimo');      // ''
console.log(punteggio ?? 100);       // 0
console.log(tema ?? 'chiaro');       // 'chiaro'
```

Se il dato è "assente" (`null`/`undefined`), applichiamo il default. Negli altri casi, manteniamo il valore originale.

Falsy coalescing

Optional chaining (`?.`) e nullish coalescing (`??`) sono costrutti relativamente recenti, introdotti in ES2020.

Prima di `??`, si usava spesso l'operatore logico OR (`||`) per impostare fallback:

```
const nome = '';
console.log(nome || 'Anonimo');  // 'Anonimo'
```

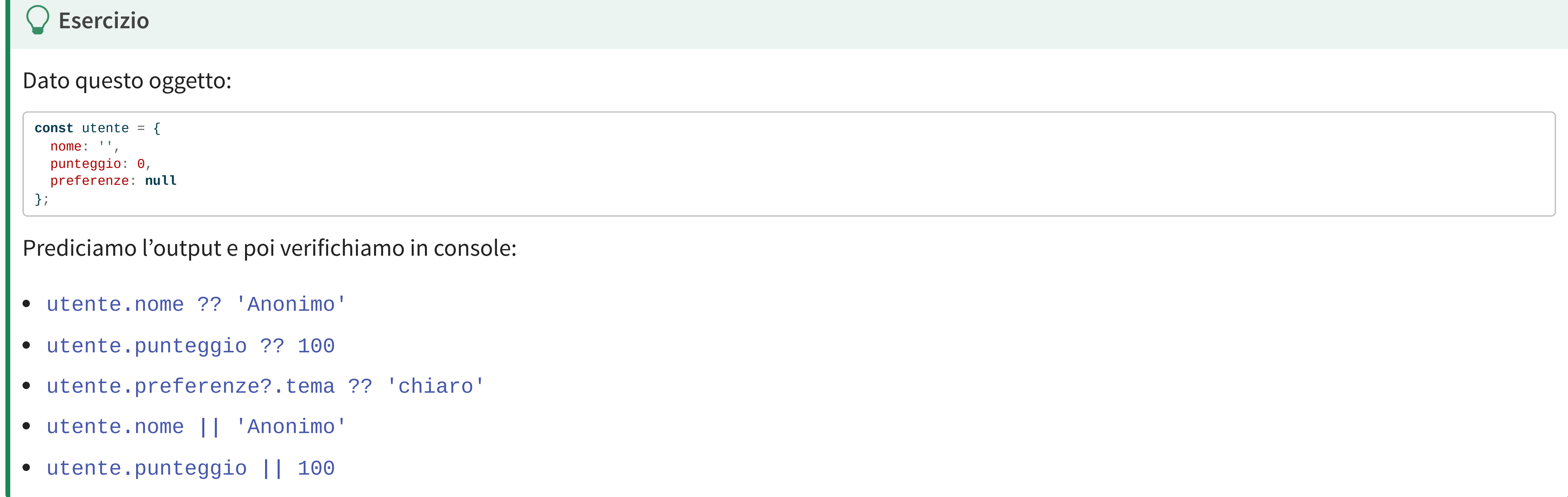
Perché `||` funziona da fallback?

- **Short-circuit:** `a || b` restituisce `a` se `a` è truthy, altrimenti passa a `b`
- Quindi con `||` i valori falsy (`0`, `''`, `false`) attivano il fallback
- `??` invece attiva il fallback solo con `null` o `undefined`

```
const valore = 0;

console.log(valore || 10);  // 10
console.log(valore ?? 10);  // 0
```

Esercizio: Accesso sicuro e fallback



Esercizio

Dato questo oggetto:

```
const utente = {
  nome: '',
  punteggio: 0,
  preferenze: null
};
```

Prediciamo l’output e poi verifichiamo in console:

- `utente.nome ?? 'Anonimo'`
- `utente.punteggio ?? 100`
- `utente.preferenze?.tema ?? 'chiaro'`
- `utente.nome || 'Anonimo'`
- `utente.punteggio || 100`

Oggetti e array insieme

Gli elementi di un array possono essere di qualsiasi tipo, inclusi **oggetti**. Spesso combiniamo **array di oggetti** per rappresentare dati strutturati:

```
const ricette = [
  { nome: 'Carbonara', tempo: 20, difficolta: 'Media' },
  { nome: 'Amatriciana', tempo: 25, difficolta: 'Media' },
  { nome: 'Cacio e pepe', tempo: 15, difficolta: 'Facile' }
];

// Accesso agli elementi
console.log(ricette[0].nome);  // 'Carbonara'
console.log(ricette[1].tempo); // 25

// Iterazione con for
for (let i = 0; i < ricette.length; i++) {
  console.log(ricette[i].nome + ' - ' + ricette[i].tempo + ' min');
}
```

Oggetti con array come proprietà

Possiamo anche avere il contrario: **oggetti che contengono array**:

```
const ricetta = {
  nome: 'Carbonara',
  tempo: 20,
  ingredienti: ['Pasta', 'Uova', 'Guanciale', 'Pecorino'],
  passaggi: [
    'Cuoci la pasta',
    'Taglia il guanciale a cubetti',
    'Mescola le uova con il pecorino',
    'Combina tutto insieme'
  ]
};

// Accesso ai dati
console.log(ricetta.nome);                    // 'Carbonara'
console.log(ricetta.ingredienti[0]);          // 'Pasta'
console.log(ricetta.passaggi.length);         // 4

// Iterazione sugli ingredienti
for (let i = 0; i < ricetta.ingredienti.length; i++) {
  console.log('- ' + ricetta.ingredienti[i]);
}
```

Verificare se è un array: `Array.isArray()`

Gli array sono oggetti! `typeof` ritorna `'object'` per entrambi:

```
const lista = [1, 2, 3];
const oggetto = { a: 1, b: 2 };

console.log(typeof lista);     // 'object'
console.log(typeof oggetto);   // 'object'
```

Per **distinguere** un array da un oggetto, usa **`Array.isArray()`**:

```
console.log(Array.isArray(lista));     // true
console.log(Array.isArray(oggetto));   // false
console.log(Array.isArray('testo'));   // false
console.log(Array.isArray(123));       // false
```

Quando usarlo

Utile quando ricevi dati da API o funzioni e vuoi verificare il tipo prima di usare metodi come `map()` o `filter()`.

Iterare oggetti e array

Sono presenti due costrutti “for each” per iterare dati in modo più leggibile: `for...of` e `for...in`.

`for...of`

```
const ingredienti = [
  'Uova',
  'Guanciale'
];

for (const ingrediente of ingredienti) {
  console.log(ingrediente);
}
```

`for...of` itera i **valori** di un oggetto **iterabile**.

`for...in`

```
const ricetta = {
  nome: 'Carbonara',
  tempo: 20
};

for (const chiave in ricetta) {
  console.log(chiave, ricetta[chiave]);
}
```

`for...in` itera le **chiavi** di un oggetto.

Tuttavia, all’atto pratico, `for...in` è generalmente **sconsigliato** per iterare oggetti, perché itera anche le proprietà ereditate dalla catena di prototipi (capiremo più avanti cosa significa).

Di conseguenza, per iterare le chiavi di un oggetto è preferibile usare `Object.keys()` o `Object.entries()` insieme a `for...of`.

L’oggetto `document`

L’oggetto globale `document` è il punto di accesso a tutto il DOM.

```
// Accedere ai principali elementi
const htmlRoot = document.documentElement;  // <html>
const page = document.body;                 // <body>
const head = document.head;                 // <head>
```

- Permette di **leggere e modificare** la pagina.
- Fornisce proprietà utili per **navigare** l’albero.
- Ogni tag HTML diventa un **oggetto** `Element` che possiamo manipolare.

L'oggetto `Element`

Ogni elemento HTML nel DOM è rappresentato come un **oggetto `Element`** con proprietà e metodi.

Proprietà comuni:

- `textContent` - testo puro (sicuro)
- `innerHTML` - contenuto HTML completo
- `className` - valore attributo `class`
- `id` - valore attributo `id`
- `tagName` - nome del tag (`'DIV'`, `'P'`)

Metodi comuni:

- `querySelector()` - cerca elemento figlio
- `classList.add/remove/toggle()` - gestisce classi
- `getAttribute()` / `setAttribute()` - legge/modifica attributi

```
const titolo = document.querySelector('h1');
console.log(titolo.textContent);  // Legge il testo
console.log(titolo.tagName);      // 'H1'
console.log(titolo.id);           // valore dell'attributo id
```

DOM traversing: come navigare l’albero

Ottenuto un elemento, possiamo **navigare l’albero** usando proprietà che collegano elementi tra loro:

```
const elemento = document.querySelector('.box');

// Accedere ai figli (SOLO elementi)
const figli = elemento.children;              // HTMLCollection (solo elementi)
const primoElemento = elemento.firstElementChild;   // Primo elemento figlio
const ultimoElemento = elemento.lastElementChild;   // Ultimo elemento figlio

// Accedere ai figli (TUTTI i nodi, incluso testo)
const tuttiNodi = elemento.childNodes;        // NodeList (elementi + testo)
const primoNodo = elemento.firstChild;        // Primo nodo (anche testo)
const ultimoNodo = elemento.lastChild;        // Ultimo nodo (anche testo)

// Accedere al genitore
const genitore = elemento.parentElement;

// Accedere ai fratelli
const fratelloSuccessivo = elemento.nextElementSibling;
const fratelloPrecedente = elemento.previousElementSibling;
```

Differenza chiave: `children` contiene **solo elementi HTML**, `childNodes` include anche **nodi di testo** (spazi, a capo).

Metodi di ricerca: `getElementsBy...`

```
// Cercare PER ID (un solo risultato)
const form = document.getElementById('main-form');

// Cercare PER TAG (più risultati)
const paragrafi = document.getElementsByTagName('p');

// Cercare PER CLASSE (più risultati)
const box = document.getElementsByClassName('box');
```

Importante: questi metodi ritornano **collezioni live** (HTMLCollection) che si aggiornano automaticamente quando il DOM cambia.

```
const items = document.getElementsByClassName('item');
console.log(items.length);  // Es. 3

// Aggiungo un nuovo elemento con classe 'item'
document.body.innerHTML += '<div class="item">Nuovo</div>';

console.log(items.length);  // 4 (si è aggiornato automaticamente!)
```

Metodi di ricerca: `querySelector`

Si tratta di alternative più moderne e flessibili rispetto ai metodi `getElementsBy...`.

```
// Cercare UN elemento (primo che corrisponde)
const firstItem = document.querySelector('.item');

// Cercare TUTTI gli elementi che corrispondono
const allItems = document.querySelectorAll('.item');
```

- Usano i **selettori CSS** completi (classi, ID, attributi, combinatori...).
- `querySelector` ritorna il **primo** match o `null`.
- `querySelectorAll` ritorna una **collezione statica** (NodeList, snapshot).

Quale metodo usare?

Metodo	Risultato	Live?	Selettore
`getElementById('id')`	`Element`	No	Solo ID
`getElementsByClassName('class')`	`HTMLCollection`	**Sì**	Solo classe
`getElementsByTagName('tag')`	`HTMLCollection`	**Sì**	Solo tag
`querySelector('selector')`	`Element`	No	CSS completo
`querySelectorAll('selector')`	`NodeList`	No	CSS completo

Generalmente, i metodi `querySelector*` sono preferibili, tranne quando serve una collezione live.

In tal caso, invece, si usano i metodi `getElement*`.

Metodo: `matches()`

Per verificare se un elemento **corrisponde a un selettore**:

```
const elemento = document.querySelector('button');

if (elemento.matches('.primary')) {
  console.log('Questo bottone ha classe "primary"');
}

if (elemento.matches('button[type="submit"]')) {
  console.log('È un bottone di submit');
}
```

Utile per **validare** o **filtrare** elementi dinamicamente.

NodeList vs HTMLCollection vs Array

I metodi di ricerca ritornano collezioni **simili ad array**, ma **non sono array veri**!

```
const paragrafi = document.querySelectorAll('p');  // NodeList
const items = document.getElementsByClassName('item');  // HTMLCollection
```

Come convertire in array vero:

```
const arrayParagrafi = Array.from(paragrafi);

for (let i = 0; i < arrayParagrafi.length; i++) {
  const p = arrayParagrafi[i];
  console.log(p.textContent);
}
```

Nota: per sicurezza, converti sempre in array quando vuoi scorrere con un ciclo.

Esercizio: Selezionare elementi

Esercizio

Dato il seguente codice HTML:

```html
<div id="sample">
  <h2 id="sample-title">Titolo di prova</h2>
  <ul>
    <li class="item">Elemento 1</li>
    <li class="item">Elemento 2</li>
    <li class="item">Elemento 3</li>
  </ul>
</div>
```

1. Proviamo a selezionare un elemento con `id` e stampiamo `textContent`.
2. Proviamo a selezionare tutti gli elementi con classe `item`.
3. Proviamo a cambiare il testo del primo elemento trovato con `textContent`.

Creare elementi: `createElement`

```
const paragrafo = document.createElement('p');
paragrafo.textContent = 'Nuovo paragrafo!';
paragrafo.className = 'importante';
```

Proprietà principali:

- `textContent`: testo puro.
- `innerHTML`: interpreta HTML.
- `className`: classe CSS.

`textContent`

Inserisce **solo testo puro**, senza interpretare HTML.

```
const paragrafo = document.querySelector('p');

paragrafo.textContent = 'Testo <strong>normale</strong>';
// Risultato visibile:
// "Testo <strong>normale</strong>"
```

`innerHTML`

Interpreta il contenuto come **HTML completo**.

```
const paragrafo = document.querySelector('p');

paragrafo.innerHTML = 'Testo <strong>normale</strong>';
// Risultato visibile:
// "Testo normale" (in grassetto)
```

`textContent` è da preferire quando possibile. Usa `innerHTML` solo quando controlli il contenuto.

Aggiungere e rimuovere elementi

`append` e `prepend`

```
const container = document.querySelector('#container');
const paragrafo = document.createElement('p');
paragrafo.textContent = 'Sono nuovo!';

// Aggiungi in fondo
container.append(paragrafo);

// Aggiungi all'inizio
container.prepend(paragrafo);
```

`remove`

```
const elemento = document.querySelector('.da-eliminare');
elemento.remove();  // Lo toglie dal DOM
```

Classi CSS: `classList`

```
const box = document.querySelector('.box');

box.classList.add('highlight');      // Aggiungi classe
box.classList.remove('hidden');      // Togli classe
box.classList.toggle('visible');     // Attiva/Disattiva

if (box.classList.contains('active')) {
  console.log('Box è attivo');
}
```

Modificare stili: `element.style`

Possiamo modificare gli stili CSS **direttamente** con JavaScript usando `element.style`:

```
const box = document.querySelector('.box');

// Modifica stile inline (come style="...")
box.style.backgroundColor = 'red';  // Nota: camelCase!
box.style.width = '200px';
box.style.fontSize = '16px';
box.style.display = 'none';  // Nasconde
```

Nota: le proprietà CSS con trattino (`background-color`) diventano camelCase (`backgroundColor`) in JavaScript.

Quando usare `classList` vs `style`:

- `classList` → preferibile per toggle show/hide, stati (attivo/inattivo), temi. Più **manutenibile**.
- `style` → utile per valori **dinamici** calcolati (es. posizione, dimensione da variabili).

```
// Meglio con classList (riusabile, manutenibile)
box.classList.toggle('hidden');

// Meglio con style (valore dinamico)
box.style.left = mouseX + 'px';
box.style.top = mouseY + 'px';
```

Attributi: `getAttribute` e `setAttribute`

Differenza tra proprietà DOM e attributi HTML:

```
const link = document.querySelector('a');

// Proprietà DOM (accesso diretto)
link.href = 'https://example.com';  // Modifica proprietà
console.log(link.href);  // Legge (URL assoluto normalizzato)

// Attributi HTML (via metodi)
link.setAttribute('href', '/nuova-url');  // Modifica attributo
const href = link.getAttribute('href');   // '/nuova-url' (valore esatto)
```

Attributi `data-*` personalizzati:

Gli attributi `data-*` sono utili per memorizzare dati custom su elementi:

```
// HTML: <button data-user-id="123" data-action="delete">Elimina</button>
const button = document.querySelector('button');

// Accesso via getAttribute
const userId = button.getAttribute('data-user-id');  // '123'

// Accesso via dataset (più comodo!)
console.log(button.dataset.userId);    // '123' (camelCase)
console.log(button.dataset.action);    // 'delete'

// Modifica
button.dataset.userId = '456';
button.dataset.status = 'active';  // Crea data-status="active"
```

Esercizio: Il ricettario via JavaScript

Ricostruiamo la pagina da zero

Riprendiamo il ricettario della lezione 1, ma stavolta **l'HTML deve essere vuoto**: costruiamo tutta la pagina con JavaScript.

1. Nell'`index.html` lascia solo un contenitore vuoto: `<div id="app"></div>`.
2. In `script.js` crea un oggetto `ricetta` con `titolo`, `descrizione`, `immagine`, `ingredienti`, `passaggi`
3. Usa `document.createElement()` per creare `h1`, `img`, `p`, `h2`, `ul`, `ol`
4. Popola `ul` e `ol` con un ciclo, creando un `li` per ogni elemento.
5. Usa `textContent` per il testo e `setAttribute` per `src` e `alt` dell'immagine.
6. Aggiungi classi con `classList.add()` per poter applicare gli stili CSS.

La pagina finale deve avere **la stessa struttura** del ricettario originale.