Ripasso ed esercizi

Niccolò Maltoni
niccolo.maltoni@diennea.com

Tipi di dato

Tipo	Esempio	Nota
`number`	`42`, `3.14`, `NaN`	include interi, decimali e `NaN`
`string`	`"ciao"`, `'ciao'`	con apici singoli `''`, doppi `""`, o backtick `` ``
`boolean`	`true`, `false`	
`null`	`null`	assenza intenzionale di valore
`undefined`	`undefined`	variabile dichiarata ma non ancora assegnata
`object`	`{ chiave: "valore" }`	collezione di proprietà chiave-valore
`array`	`[1, 2, 3]`	lista ordinata; è un tipo di `object`
`function`	`function f() {}`, `() => {}`	*arrow function* e *function declaration* sono equivalenti

I tipi reference come `object`, `array` e `function` vengono passati **per riferimento**: assegnare un oggetto a una nuova variabile non lo copia.

Conversioni di tipo

Verso	Esplicito	Attenzione
numero	`Number()`, `parseInt()`, `parseFloat()`	`Number("")` → `0`; `Number("a")` → `NaN`
stringa	`String()`, `.toString()`	sempre sicuro
booleano	`Boolean()`, `!!`	`false`, `0`, `""`, `null`, `undefined`, `NaN` → `false`

Meglio essere espliciti con le conversioni di tipo quando possibile.

La **coercizione implicita** può sorprendere: e.g. `"5" + 2` → `"52"`, ma `"5" - 2` → `3`.

Condizionali e fallback

Strumento	Quando usarlo	Nota
`if / else`	logica con più rami	il caso più flessibile
`switch`	confronto su più valori discreti	utile con stati o categorie
operatore ternario `? :`	scelta breve su una riga	da usare solo se resta leggibile
optional chaining `?.`	accedere a proprietà che potrebbero essere `null`/`undefined`	restituisce `undefined` invece di lanciare errore
nullish coalescing `??`	valore di fallback se `null` o `undefined`	diversamente da `||` non si attiva sui valory falsy

`0`, `-0`, `0n`, `""`, `null`, `undefined`, `NaN` sono trattati come **falsy**. Tutto il resto è **truthy**, compresi `[]` e `{}`.

Cicli

Ciclo	Quando usarlo	Nota
`for`	serve un indice o criterio personalizzato	controllo totale sull'iterazione
`while`	numero di iterazioni non noto	attenzione ai loop infiniti
`for...of`	scorrere i **valori** di array o stringhe	non funziona su oggetti plain
`for...in`	scorrere le **chiavi** di un oggetto	evitare su array
`.forEach(fn)`	eseguire un effetto su ogni elemento	metodo di array

Per trasformare o filtrare un array preferiamo i metodi funzionali (`map`, `filter`, …).

Metodi array

Metodo	Restituisce	Nota
`.filter(fn)`	nuovo array	tenere solo elementi che soddisfano la condizione
`.map(fn)`	nuovo array	trasformare ogni elemento
`.find(fn)`	elemento o `undefined`	primo elemento che soddisfa la condizione
`.reduce(fn, init)`	valore singolo	totale, media, aggregazioni
`.includes(v)`	`boolean`	verifica se un valore è presente
`.push(v)` / `.pop()`	lunghezza / elemento	aggiunge/rimuove in fondo (modifica array)

Metodi string

Metodo	Restituisce	Nota
`.split(sep)`	array	divide in array; inverso è `array.join(sep)`
`.replace(old, new)`	nuova stringa	sostituisce prima occorrenza
`.trim()`	nuova stringa	rimuove spazi all'inizio e fine
`.toLowerCase()` / `.toUpperCase()`	nuova stringa	cambio maiuscolo/minuscolo
`.includes(str)`	`boolean`	verifica se contiene la stringa
`.slice(i, j)`	nuova stringa	sottostringa senza modificare originale

Le stringhe sono **immutabili**: ogni metodo restituisce una nuova stringa.

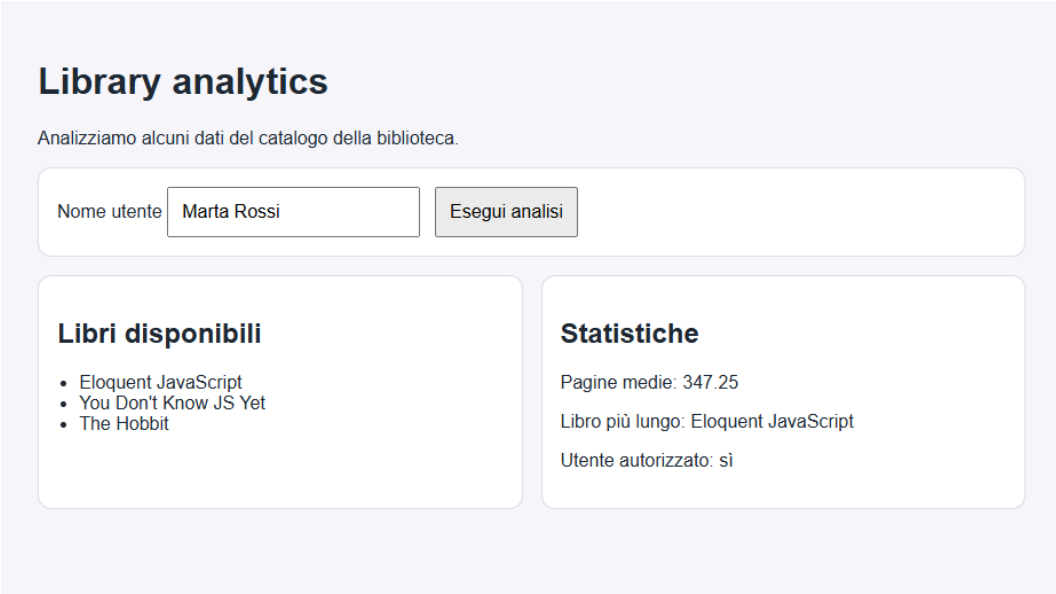
Esercizio 1

Analisi del catalogo

Osserva il codice nella cartella starter `01-library-analytics`. Si tratta di un semplice catalogo di libri, con un insieme di utenti autorizzati ai prestiti.

Nonostante la struttura fornita, il codice non è completo e non funziona correttamente. Implementa le seguenti funzionalità:

- filtrare i libri disponibili per essere presi in prestito
- calcolare il numero medio di pagine
- trovare il libro più lungo
- verificare se un utente appartiene alla whitelist dei bibliotecari



Una volta completato, verifica che la pagina funzioni correttamente.

DOM ed eventi

Operazione	Strumento principale	Nota
selezionare elementi	`querySelector`, `querySelectorAll`	selettori semplici e stabili
reagire a interazioni	`addEventListener`	separare logica e render
aggiornare la vista	funzioni di render dedicate	prima stato, poi DOM

Esempio:

```
const button = document.querySelector('.btn');                    ①
button.addEventListener('click', (e) => {
  /* ... */                                                       ②

  render();                                                       ③
});
```

① seleziona l'elemento nel DOM
② aggiorna lo stato dell'applicazione
③ aggiorna la vista in base al nuovo stato

Modificare il DOM

Operazione	Metodo	Nota
creare un elemento	`document.createElement('div')`	restituisce l'elemento, non ancora in pagina
aggiungere al DOM	`.appendChild()` / `.append()`	`append` supporta nodi e stringhe
cambiare il testo	`.textContent`	proprietà; preferire a `innerHTML`
svuotare un container	`.innerHTML = ''`	`innerHTML` è ok per modifiche strutturali
aggiungere una classe	`.classList.add('active')`	preferire `classList` a `className`
abilitare un elemento	`.disabled = false`	proprietà
rimuovere un elemento	`.remove()`	elimina l'elemento dal DOM

Esempio:

```
const container = document.querySelector('.list');

const element = document.createElement('div');
element.textContent = 'Nuovo elemento';
element.classList.add('item');

container.appendChild(element);
```

1. seleziona il contenitore nel DOM
2. crea e configura il nuovo nodo
3. inserisce il nodo nella pagina

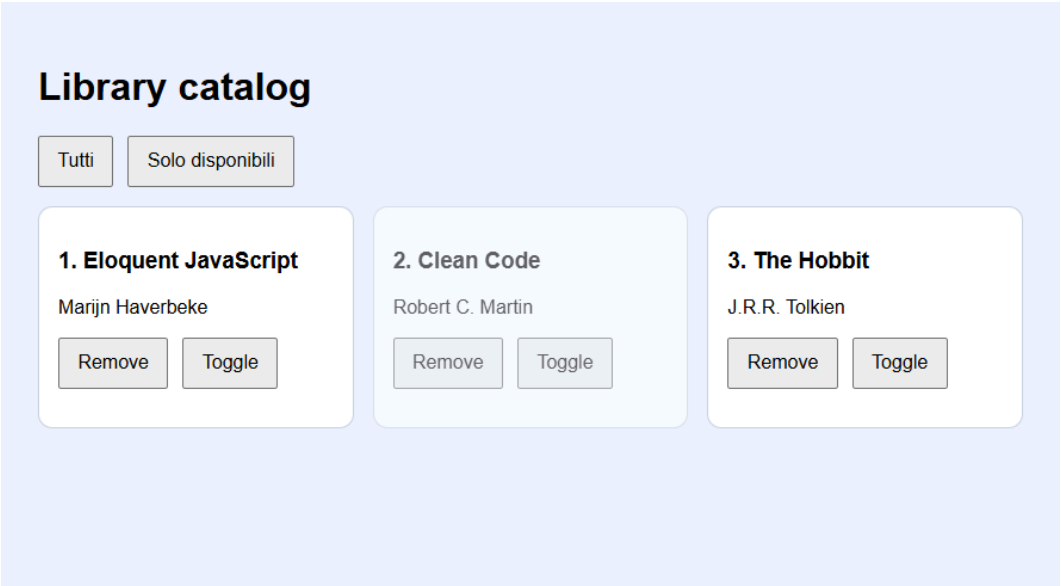
Esercizio 2

Catalogo interattivo

Osserva il codice nella cartella starter `02-library-catalog`. Vogliamo realizzare un piccolo catalogo interattivo, ma il codice non è completo e non funziona correttamente.

Implementa le seguenti funzionalità:

- definire la classe `Book` con i campi e i metodi necessari
- alla pressione dei due pulsanti, renderizzare una lista di card nel DOM, una per ogni libro
  - una classe CSS `.card` è già definita nel file `style.css` fornito
- aggiungere il pulsante per rimuovere un libro
  - come possiamo identificare univocamente un libro? Osserva il listener parzialmente implementato…
- aggiungere un pulsante per segnare un libro come disponibile o non disponibile
  - una classe CSS `.unavailable` è già definita nel file `style.css` fornito



Una volta completato, verifica che la pagina funzioni correttamente.

Form e persistenza

Fase	Strumento	Nota
gestione submit	`.preventDefault()`	evita il refresh automatico della pagina
lettura input	`new FormData()`	raccoglie i campi del form
reset form	`.reset()`	pulisce i campi dopo il salvataggio
salvataggio locale	`localStorage`	persistenza nel browser
serializzazione	`JSON.stringify` / `JSON.parse`	passaggio stringa a oggetto e viceversa

Esempio:

```
const form = document.querySelector('#data-form');

form.addEventListener('submit', (event) => {
  event.preventDefault();

  const formData = new FormData(form);
  const payload = Object.fromEntries(formData.entries());

  items.push(payload);
  localStorage.setItem('items', JSON.stringify(items));

  form.reset();
  render();
});
```

Esercizio 3

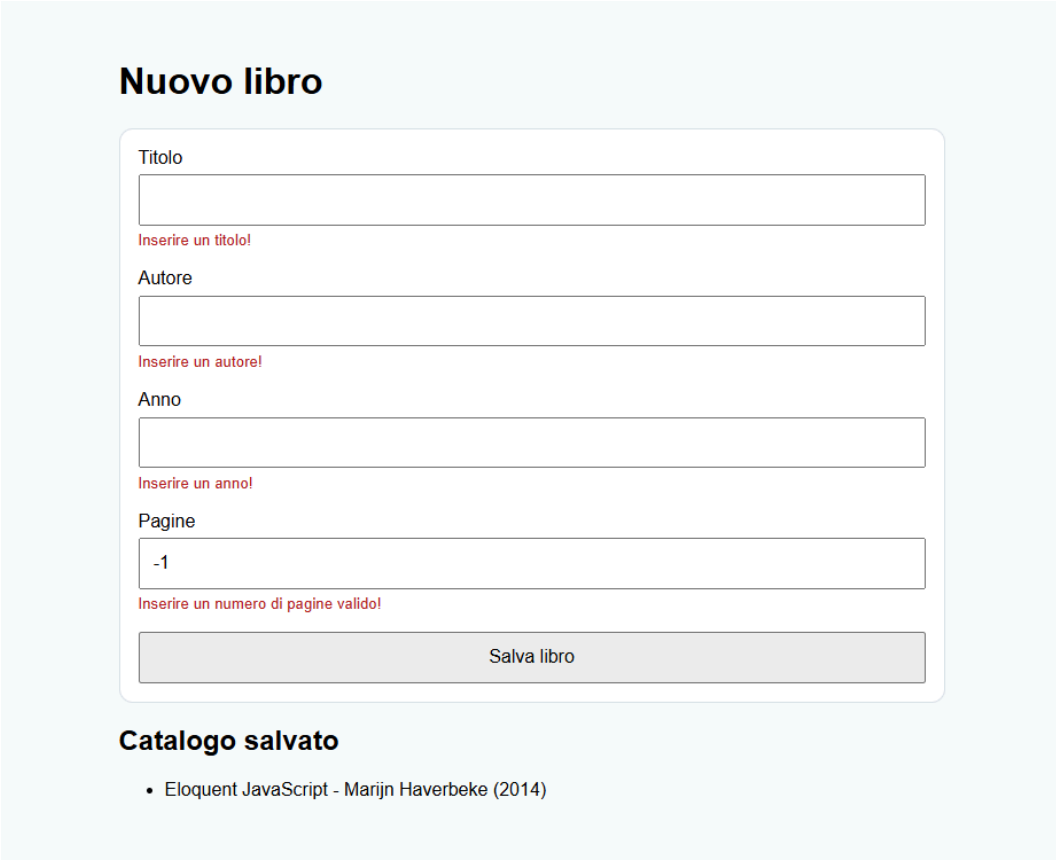
Form di inserimento

Osserva il codice nella cartella starter `03-library-form`. Vogliamo completare l'implementazione del form di registrazione di un nuovo libro.

Feature da implementare:

- leggere correttamente tutti i campi del form
- validare titolo, autore, anno e numero di pagine
- mostrare errori inline se i dati non sono validi
- salvare il catalogo in `localStorage`
- ricaricare il catalogo al refresh della pagina

Una volta completato, verifica che la pagina funzioni correttamente.



HTTP: metodi e codici principali

Metodo	Uso tipico
`GET`	leggere dati
`POST`	creare una nuova risorsa
`PUT`	sostituire una risorsa esistente
`PATCH`	aggiornare parzialmente
`DELETE`	rimuovere una risorsa

Status code	Significato
`200`	richiesta riuscita
`201`	risorsa creata
`204`	richiesta riuscita senza body
`400`	richiesta non valida
`404`	risorsa non trovata
`500`	errore lato server

XMLHttpRequest (XHR)

Passo	API	Nota
creare richiesta	`new XMLHttpRequest()`	API storica del browser
configurare	`xhr.open(method, url, true)`	`true` = asincrona
gestire risposta	`xhr.onreadystatechange`	controllare `readyState` e `status`
inviare	`xhr.send()`	avvia la richiesta

Esempio:

Asincrono (`true`):

```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'https://jsonplaceholder.typicode.com/todos/1', tru

xhr.onreadystatechange = () => {
  if (xhr.readyState === 4 && xhr.status === 200) {
    const item = JSON.parse(xhr.responseText);
    render(item);
  }
};

xhr.send();
```

Sincrono (`false`):

```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'https://jsonplaceholder.typicode.com/todos/1', fal
xhr.send();

if (xhr.status === 200) {
  const item = JSON.parse(xhr.responseText);
  render(item);
}
```

Blocca il thread principale in attesa della risposta.

Fetch API

Passo	API	Nota
richiesta	`fetch(url)`	restituisce una `Promise<Response>`
controllo esito	`response.ok`	da verificare sempre
parsing	`response.json()`	asincrono
errore	`.catch()`	intercetta errori di rete o lanci manuali

Esempio:

```
fetch('https://jsonplaceholder.typicode.com/todos?_limit=5')
  .then((response) => {
    if (!response.ok) throw new Error('HTTP error');
    return response.json();
  })
  .then((items) => render(items))
  .catch(() => showError('Caricamento non riuscito'));
```

Promise

Concetto	Significato	Metodo comune
`pending`	operazione in corso	creazione richiesta
`fulfilled`	operazione completata	`.then()`
`rejected`	operazione fallita	`.catch()`

Esempio:

```
const loadItems = () =>
  fetch('https://jsonplaceholder.typicode.com/todos?_limit=10').then((response) => {
    if (!response.ok) throw new Error('HTTP error');
    return response.json();
  });

loadItems()
  .then((items) => items.filter((item) => item.completed))
  .then((activeItems) => render(activeItems))
  .catch(() => showError('Errore nel flusso Promise'));
```

Esercizio 4

Caricamento remoto

Carichiamo un catalogo remoto con `XMLHttpRequest` e `fetch`.

Da completare (cartella `04-library-api`):

- mostrare uno stato di caricamento
- recuperare i dati da un endpoint remoto
- renderizzare i primi risultati nel DOM
- gestire un errore HTTP o di rete

Open Library API

- endpoint: `https://openlibrary.org/search.json?q=javascript`
- risposta: `docs` (array di oggetti)
- campi utili: `title`, `author_name`
- prendere i primi risultati (es. `slice(0, 5)`)

Vedi documentazione: https://openlibrary.org/developers/api

Output atteso: lista libri caricata correttamente oppure messaggio di errore leggibile.

Proviamo a usare sia `XMLHttpRequest` che `fetch` per confrontare le due API.

Task A: XMLHttpRequest sincrono

- carica un singolo libro usando XHR con `open(..., false)`
- mostra il risultato nel DOM in modo sincrono

Task B: Fetch con Promise

- carica una lista di libri utilizzando `fetch()`
- gestisci l'esito della risposta con `.then()` e `.catch()`

Async/await

Invece di usare `.then()` e `.catch()`, possiamo scrivere il flusso asincrono in modo più lineare con `async` e `await`. È sempre basato su `Promise`, solo con una sintassi diversa.

Strumento	Uso	Nota
`async`	definisce funzione asincrona	la funzione ritorna una Promise
`await`	attende una `Promise`	equivalente di `.then()`, attende la risoluzione della `Promise`
`try/catch`	gestione errori	equivalente di `.catch()`, intercetta errori

Esempio:

```
async function loadItems() {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/todos?_limit=5');
    if (!response.ok) throw new Error('HTTP error');

    const items = await response.json();
    render(items);
  } catch {
    showError('Caricamento non riuscito');
  }
}

loadItems();
```

Bonus: Esercizio avanzato 4-bis

Riscrivere con async/await

Se hai completato il Task B con `.then()` e `.catch()`, prova a riscrivere la stessa funzione utilizzando `async/await` e `try/catch`.

Il comportamento esterno deve restare identico; cambia solo lo stile del codice.

Esercizio avanzato 5

Richieste parallele con fetch

Osserva il codice nella cartella starter `05-library-parallel-fetch`. Vogliamo completare il caricamento di due sorgenti dati in parallelo.

Implementa le seguenti funzionalità:

- effettuare due richieste in parallelo; recuperare ad esempio il numero di risultati
- mostrare i risultati solo quando entrambe le richieste sono concluse
- gestire il caso in cui una delle due richieste fallisca

Una volta completato, verifica che la pagina mostri i dati correttamente oppure un errore leggibile.

Esercizio avanzato 6

Ricerca e ordinamento nel catalogo

Osserva il codice nella cartella starter `06-library-extra`. Vogliamo completare una versione base ma completa del catalogo.

Implementa le seguenti funzionalità:

- aggiungere una ricerca live
- ordinare i libri per anno o numero di pagine
- combinare ricerca e ordinamento nello stesso flusso
- mostrare un messaggio quando non ci sono risultati

Output atteso:

- con ricerca `code` viene mostrato solo `Clean Code`
- con ordinamento per anno l'ordine è `1937`, `2008`, `2018`
- con una ricerca senza match viene mostrato un messaggio tipo `Nessun risultato`