Ripasso finale

Niccolò Maltoni
niccolo.maltoni@diennea.com

Tipi e valori

Tipo	Esempio	Nota
Tipi primitivi	`number, string, boolean, null, undefined`	passati per valore
Tipi riferimento	`object, array, function`	passati per riferimento
Falsy values	`false, 0, "", null, undefined, NaN`	tutto il resto è truthy, anche `[]` e `{}`

- I **tipi primitivi** sono passati per copia: assegnando una variabile a un'altra copiamo il valore.
- I **tipi riferimento** condividono lo stesso oggetto in memoria: se lo modifichiamo in un punto, vediamo la modifica anche altrove.
- `typeof` restituisce una stringa che indica il tipo del valore.

Operatori principali

Famiglia	Operatori	Uso tipico
Aritmetici	`+ - * / %`	calcoli numerici
Assegnazione	`= += -= *=`	aggiornare una variabile
Confronto	`> < >= <= === !==`	confrontare valori; preferire `===`
Logici	`&& || !`	combinare condizioni e gestire fallback
Accesso/fallback	`?. ??`	evitare errori e usare fallback mirati
Incremento	`++ --`	aumentare o diminuire di 1

Coercizione di tipo

Contesto	Esempio	Cosa succede
Conversione esplicita	`String(123)`	converte il numero in stringa
	`Number('42')`	converte una stringa numerica in numero
	`Boolean(0)`	`0` diventa `false`
`+` con stringhe	`"5" + 2`	concatena se c'è una stringa
Operatori numerici	`"5" - 2`	convertono in numero
Confronto debole	`"5" == 5`	`==` fa coercizione tra tipi diversi
Confronto stretto	`"5" === 5`	`===` confronta anche il tipo

- `+` concatena quando entra una stringa; `-`, `*`, `/` convertono in numero.
- `==` fa conversione automatica di tipo, `===` confronta anche il tipo.
- Nel dubbio, conviene convertire in modo esplicito con `String()`, `Number()` o `Boolean()`.
- `||` e `??` non sono equivalenti: con `??` il fallback scatta solo con `null` o `undefined`.

Metodi array

Metodo	Restituisce	Modifica originale?	Uso tipico
`.filter(fn)`	nuovo array	no	tenere solo certi elementi
`.map(fn)`	nuovo array	no	trasformare ogni elemento
`.find(fn)`	elemento o `undefined`	no	primo elemento che soddisfa
`.reduce(fn, init)`	valore singolo	no	somme, aggregazioni
`.includes(v)`	`boolean`	no	verifica presenza
`.push(v)`	nuova lunghezza	**si**	aggiunge in fondo
`.splice(i, n)`	elementi rimossi	**si**	rimuove o inserisce in posizione

> I metodi che **non modificano** l'originale si compongono bene in catena.

Funzioni: organizzare il codice

Forma	Sintassi	Quando usarla
Function declaration	`function render(items) { // ... }`	funzioni principali con un nome chiaro
Arrow function	`const somma = (a, b) => a + b;` `const somma = (a, b) => { return a + b; };`	callbacks, funzioni brevi, `.forEach` ecc.
Funzione asincrona	`async function carica() { await fetch(...) }`	funzioni che usano `await` ritornano implicitamente una `Promise`

- Ogni funzione dovrebbe avere **una responsabilità**: leggere input, aggiornare stato, o aggiornare il DOM.
- Preferire nomi descrittivi: `render`, `handleSubmit`, `caricaDati`.

Oggetti e classi

Oggetto letterale — utile per uno stato unico:

```
const libro = {
  titolo: 'Clean Code',
  anno: 2008,
  descrizione() {
    return `${this.titolo} (${this.anno})`;
  },
};

console.log(libro.descrizione());
```

Classe — utile per creare più oggetti con la stessa struttura:

```
class Libro {
  constructor(titolo, anno) {
    this.titolo = titolo;
    this.anno = anno;
  }

  descrizione() {
    return `${this.titolo} (${this.anno})`;
  }
}

const b = new Libro('Clean Code', 2008);
```

- `this` dentro un metodo si riferisce all'istanza corrente.
- `new` crea una nuova istanza e chiama `constructor`.

Quiz: catena map + filter

```
const numeri = [1, 2, 3, 4, 5];
const risultato = numeri
  .filter(n => n % 2 === 0)
  .map(n => n * 10);
```

Cosa otteniamo?

Cosa contiene `risultato`?

Risposta

`[20, 40]`

`filter` tiene solo i pari (`2`, `4`), poi `map` li moltiplica per 10. L'array originale `numeri` non viene modificato.

Quiz: falsy e `Boolean`

```
Boolean(0) === Boolean("")
```

Vero o falso?

Quale delle due affermazioni è corretta?

- A: `false`, perché `0` e `""` sono tipi diversi
- B: `true`, perché entrambi sono falsy e diventano `false`

Risposta

B — `true`.
`Boolean(0)` → `false`, `Boolean("")` → `false`, quindi `false === false` → `true`.

Attenzione: `Boolean([])` → `true` e `Boolean({})` → `true`. Array e oggetti vuoti sono sempre truthy.

Quiz: cosa restituisce `find`?

```
const persone = [
  { nome: 'Anna', eta: 17 },
  { nome: 'Leo',  eta: 22 },
  { nome: 'Mia',  eta: 19 },
];

const adulto = persone.find(p => p.eta >= 18);
```

Osserva questo codice

Che valore ha `adulto`? Che tipo è?

Risposta

```
{ nome: 'Leo', eta: 22 }
```

- `find` restituisce **l'elemento** (oggetto intero), non `true` né indice.
- Si ferma al primo match.
- Se non trova nulla, restituisce `undefined`.

DOM: selezione ed eventi

Operazione	Strumento	Nota
Selezionare un elemento	`document.querySelector(sel)`	restituisce il primo, o `null`
Selezionare più elementi	`document.querySelectorAll(sel)`	restituisce `NodeList` (iterabile)
Ascoltare un evento	`el.addEventListener('click', fn)`	separare logica da render
Delegazione eventi	`parent.addEventListener('click', fn)` `// usa event.target`	un solo listener per molti figli

DOM: manipolazione e render

Operazione	Strumento	Nota
Cambiare il testo	`el.textContent = "..."`	meglio di `innerHTML` per testo
Struttura HTML dinamica	`el.innerHTML = "..."`	utile per svuotare o creare HTML
Aggiungere/rimuovere classe	`el.classList.add('x')` `el.classList.remove('x')`	meglio di `el.className`
Creare un nodo	`document.createElement('div')`	non ancora in pagina
Inserire nel DOM	`parent.appendChild(el)` `parent.append(el)`	`append` accetta anche stringhe
Rimuovere un nodo	`el.remove()`	rimuove il nodo dal DOM

Quiz: listener e `querySelectorAll`

```
const bottoni = document.querySelectorAll('.btn');

bottoni.forEach(btn => {
  btn.addEventListener('click', () => {
    console.log('cliccato');
  });
});
```

Quanti listener vengono registrati?

Se ci sono 5 bottoni in pagina, quante volte viene chiamato `addEventListener`?

Risposta

5 volte — una per ogni elemento selezionato.

Alternativa con **delegazione**: un solo listener sul contenitore padre, poi `event.target` per identificare quale bottone ha ricevuto il click. Utile quando i figli vengono creati dinamicamente.

Quiz: delegazione con `event.target`

```
const lista = document.querySelector('#lista');

lista.addEventListener('click', (event) => {
  if (!event.target.matches('button')) return;

  const id = event.target.dataset.id;
  rimuoviElemento(id);
});
```

Qual è l'elemento cliccato?

Cosa succede se clicchiamo il testo dentro al bottone? E se clicchiamo fuori dal bottone?

Risposta

- Se il click arriva su un `button`, il codice legge `dataset.id` e rimuove l'elemento.
- Se il click arriva fuori dal `button`, fa `return` e non succede nulla.
- Pattern utile: un listener sul contenitore, logica condizionale con `event.target`.

Quiz: render da stato

```
function render(items) {
  const container = document.querySelector('#lista');
  container.innerHTML = '';           // svuota

  items.forEach(item => {
    const div = document.createElement('div');
    div.textContent = item.nome;
    container.appendChild(div);
  });
}
```

Perché svuotiamo prima?

Perché svuotiamo `container` prima di ricrearne il contenuto?

Risposta

- Per evitare **duplicati** nel DOM.
- Ogni `render` deve rappresentare lo stato attuale (`items`).

Pattern applicazione:

1. evento (click, submit, …)
2. aggiorna lo stato (`array.push`, `array.splice`, …)
3. `render(stato)` → svuota container → ricrea nodi

> Lo stato è la fonte di verità. Il DOM è solo il suo riflesso.
>
> Niccolò Maltoni

Form e validazione

Fase	Strumento	Nota
Intercettare invio	`form.addEventListener('submit', (event) => { event.preventDefault(); });`	blocca il refresh automatico
Leggere i campi	`const dati = new FormData(form); Object.fromEntries(dati.entries())`	raccoglie i campi con `name`
Pulire il form	`form.reset()`	dopo il salvataggio
Mostrare errori	scrivere in `<span>` o `<p>` con classe `.error`	visibile, non solo `alert`
Persistenza locale	`localStorage.setItem(chiave, JSON.stringify(val))`	serializzare prima di salvare

Quiz: cosa succede?

```
form.addEventListener('submit', (event) => {
  const dati = new FormData(form);
  const payload = Object.fromEntries(dati.entries());
  elementi.push(payload);
  render();
});
```

Osserva questo listener

Manca qualcosa. Qual è il problema?

Risposta

Manca `event.preventDefault()`.

Senza di esso, il browser esegue il comportamento predefinito del form: **invia una richiesta HTTP e ricarica la pagina**. Tutto il codice JS dopo `push` viene eseguito, ma immediatamente dopo la pagina si azzera.

Prima riga del listener: sempre `event.preventDefault()`.

Quiz: `JSON.stringify` e `localStorage`

```
const contatti = [{ nome: 'Anna', email: 'anna@mail.com' }];

// Caso A
localStorage.setItem('contatti', contatti);

// Caso B
localStorage.setItem('contatti', JSON.stringify(contatti));
```

Cosa viene salvato?

Qual è la differenza tra i due casi?

Risposta

`localStorage` salva solo **stringhe**.

- **Caso A**: converte l'array implicitamente → salva `"[object Object]"`. Se poi proviamo a fare `JSON.parse("[object Object]")`, otteniamo un errore.
- **Caso B**: serializza correttamente → salva `'[{"nome":"Anna","email":"anna@mail.com"}]'`. `JSON.parse` restituisce l'array originale.

Regola: `JSON.stringify` prima di `setItem`, `JSON.parse` dopo `getItem`.

Fetch e Promise

Concetto	API / sintassi	Nota
Avviare richiesta	`fetch(url)`	restituisce una `Promise` contenente `Response`
Controllare esito	`response.ok`	`fetch` non rigetta su errori HTTP 4xx/5xx
Leggere JSON	`await response.json()`	asincrono, restituisce un oggetto
Catena Promise	`.then().then().catch()`	errori propagano fino al primo `.catch`
Async/await	`async function f() { await caricaDati(); }`	sintassi lineare sulle Promise
Gestione errori	`try { await caricaDati(); } catch (err) { mostraErrore(err); }`	equivalente di `.catch()`
Richieste parallele	`Promise.all([p1, p2])`	aspetta tutte; rigetta se una fallisce

Errori con `fetch`

Caso	Comportamento
Errore di rete	`fetch` rigetta la Promise (`catch`)
Errore HTTP (es. 404, 500)	`fetch` risolve la Promise, ma `response.ok` è `false`

> Dobbiamo controllare **sempre** `response.ok` (o `response.status`) dopo `await fetch(...)`.

Quiz: `fetch` e codici HTTP

```
async function caricaDati(url) {
  const response = await fetch(url);
  const dati = await response.json();
  render(dati);
}
```

Questo codice gestisce un 404?

Se il server risponde con `404 Not Found`, cosa succede?

Risposta

`fetch` **non rigetta** su errori HTTP: la Promise si risolve con `response.ok = false`.

Quindi dobbiamo controllare `ok` prima di usare i dati.

La correzione:

```
const response = await fetch(url);
if (!response.ok) throw new Error(`HTTP ${response.status}`);
const dati = await response.json();
```

Quiz: propagazione degli errori in catena

```
fetch(url)
  .then(response => {
    if (!response.ok) throw new Error('HTTP error');
    return response.json();
  })
  .then(dati => render(dati))
  .catch(err => mostraErrore(err.message));
```

Dove finisce l'errore?

Se il primo `.then` lancia un errore, quale step viene eseguito dopo?

Risposta

L'errore **salta** il secondo `.then` e viene gestito direttamente nel `.catch` in fondo alla catena.

Un errore in qualsiasi punto della catena `.then` si propaga verso il basso fino al primo `.catch` disponibile. Il secondo `.then` (`render`) viene saltato completamente.

Quiz: `async/await` senza `try/catch`

```
async function caricaContatti() {
  const response = await fetch('/api/contatti');
  if (!response.ok) throw new Error(`HTTP ${response.status}`);
  const contatti = await response.json();
  render(contatti);
}
```

Cosa manca?

Il codice controlla `response.ok`, ma c'è ancora un caso non gestito. Quale?

Risposta

Manca la gestione degli **errori di rete**: se il dispositivo è offline o il server non risponde, `fetch` rigetta la Promise e l'errore non viene intercettato.

La funzione `async` restituisce una Promise rigettata che, se non gestita, produce un `UnhandledPromiseRejection`.

Soluzione:

```
async function caricaContatti() {
  try {
    const response = await fetch('/api/contatti');
    if (!response.ok) throw new Error(`HTTP ${response.status}`);
    const contatti = await response.json();
    render(contatti);
  } catch (err) {
    mostraErrore('Impossibile caricare i contatti');
  }
}
```

Quiz: `Promise.all` e fail-fast

```
const [utenti, prodotti] = await Promise.all([
  fetch('/api/utenti').then(r => r.json()),
  fetch('/api/prodotti').then(r => r.json()),
]);
```

Cosa succede se una fallisce?

Se la seconda richiesta restituisce un errore di rete, cosa succede all'intera `Promise.all`?

Risposta

`Promise.all` ha comportamento **fail-fast**: rigetta **immediatamente** non appena una qualsiasi Promise nella lista rigetta. La prima richiesta potrebbe essere già completata, ma il suo risultato viene scartato.