Funzioni, array, ed eventi

Niccolò Maltoni
niccolo.maltoni@diennea.com

Cos'è una funzione?

Una **funzione** è un blocco di codice che esegue un compito specifico. Abbiamo già visto esempi di funzioni integrate nel linguaggio, come `alert(message)` e `console.log(value)`.

Per creare una funzione dobbiamo utilizzare una **dichiarazione** di funzione:

```
// Dichiarazione di funzione
function saluta(nome) {
  console.log('Ciao ' + nome);
}
```

La parola chiave `function` va posta all'inizio; viene seguita da:

1. **nome** della funzione,
2. una lista di **parametri** (detti anche **argomenti**), racchiusi tra parentesi e separati da virgola,
3. il **codice** della funzione, chiamato anche *corpo della funzione*, racchiuso tra parentesi graffe.

```
saluta('Mario');     // Stampa: "Ciao Mario"
saluta('Luigi');     // Stampa: "Ciao Luigi"
```

Una funzione dichiarata può essere **invocata** all'interno di un'espressione attraverso il suo nome.

Restituire un valore

L'istruzione `return` **interrompe** l'esecuzione della funzione e restituisce il valore.

```
function somma(a, b) {
  return a + b;
}

const risultato = somma(5, 3);
console.log(risultato);  // Stampa: 8
```

- Ci possono essere più occorrenze di return in una singola funzione.
- Una funzione senza `return` restituisce `undefined`.

⚠ **Attenzione**

È anche possibile specificare `return` senza un valore: in questo caso la funzione restituirà `undefined`, come se non avesse un'istruzione `return`.

In JavaScript, però, è possibile terminare la riga con un punto e virgola (`;`) o andare a capo. Se si va a capo subito dopo `return`, JavaScript inserirà automaticamente un punto e virgola, causando il ritorno di `undefined`!

```
function esempio() {
  return
    42;  // Mai raggiunto, perché return termina la funzione
}
```

Esercizio: Calcolatrice semplice

Funzioni che ritornano valori

Crea 4 funzioni:

1. `somma(a, b)` che ritorna la somma di `a` e `b`
2. `sottrai(a, b)` che ritorna la sottrazione di `b` da `a`
3. `moltiplica(a, b)` che ritorna il prodotto di `a` e `b`
4. `dividi(a, b)` che ritorna la divisione di `a` per `b`

Verifica il funzionamento delle funzioni:

```
console.log(somma(10, 5));       // 15
console.log(sottrai(10, 5));     // 5
console.log(moltiplica(10, 5));  // 50
console.log(dividi(10, 5));      // 2
```

Arrow functions

Esiste un'altra sintassi, più concisa, per creare funzioni e che spesso è preferibile a `function`.

Questa notazione è chiamata **arrow function** per via della freccia `=>` (composta dai simboli uguale e maggiore) usata tra i parametri e il corpo della funzione.

```
// Dichiarazione tradizionale
const somma = function(a, b) {
  return a + b;
};

// Arrow function (forma completa)
const somma = (a, b) => {
  return a + b;
};
```

Le arrow functions possiedono altre caratteristiche, che vedremo più avanti.

Arrow functions

Se abbiamo un solo argomento, le parentesi che racchiudono gli argomenti possono essere omesse, abbreviando ulteriormente il codice.

```
// Arrow function (implicit return)
const somma = (a, b) => a + b;
```

Se non ci sono argomenti, le parentesi saranno vuote (ma devono essere presenti).

```
// Un parametro (no parentesi)
const quadrato = x => x * x;
```

Se il corpo della funzione è una singola espressione, le parentesi graffe e la parola chiave `return` possono essere omesse: il valore dell'espressione sarà restituito automaticamente (implicit return).

```
// Nessun parametro
const random = () => Math.random();
```

Variabili e scope

Lo **scope** determina dove una variabile è accessibile.

```
function test() {
  let x = 10;          // Scope locale
  const y = 20;        // Scope locale

  console.log(x, y);   // 10 20 (OK)
}

console.log(x);        // Errore! x non esiste qui
```

Le variabili dichiarate con `let` o `const` dentro una funzione **non si vedono** da fuori.

Viceversa, le variabili dichiarate fuori da una funzione sono considerate **globali** e utilizzabili anche all'interno della funzione.

```
let globale = 'Posso accedervi ovunque';

function test() {
  let locale = 'Solo qui dentro';

  console.log(globale);  // OK
  console.log(locale);   // OK
}

console.log(globale);    // OK
console.log(locale);     // Errore!
```

Ricorsione

Una **funzione ricorsiva** è una funzione che **chiama se stessa**.

```
function contoAllaRovescia(n) {
  if (n === 0) {
    console.log('Partenza!');
    return;  // Caso base: ferma la ricorsione
  }
  console.log(n);
  contoAllaRovescia(n - 1);  // Chiamata ricorsiva
}

contoAllaRovescia(3);  // 3, 2, 1, Partenza!
```

La ricorsione è utile per **problemi che si ripetono** su dati più piccoli (alberi, liste, calcoli matematici).

Esempio pratico

Calcoliamo il **fattoriale** di un numero: `n! = n × (n-1) × ... × 1`

```
function fattoriale(n) {
  if (n === 1) {
    return 1;  // Caso base
  }
  return n * fattoriale(n - 1);  // n × fattoriale di (n-1)
}

fattoriale(5);  // 5 × 4 × 3 × 2 × 1 = 120
fattoriale(3);  // 3 × 2 × 1 = 6
```

Funzioni come valori

In JavaScript, le funzioni sono *first-class citizen*!

Possono essere:

- assegnate a variabili,
- passate come parametri a altre funzioni,
- restituite da altre funzioni.

```
// Funzione assegnata a una variabile
const saluta = function(nome) {
  return 'Ciao ' + nome;
};

const messaggio = saluta('Mario');  // Chiama come variabile
```

Callbacks: funzioni come parametri

Una **callback** è una funzione passata come argomento a un'altra funzione.

```
function processaNomi(nomi, callback) {
  for (let i = 0; i < nomi.length; i++) {
    callback(nomi[i]);
  }
}

// Callback di esempio
function stampa(nome) {
  console.log('Nome: ' + nome);
}

processaNomi(['Mario', 'Luigi'], stampa);
// Output: Nome: Mario
//         Nome: Luigi
```

Le callback sono fondamentali per lavorare con operazioni asincrone, che vedremo più avanti.

Esercizio: Applicare operazione

Callbacks in azione

Scrivi una funzione `applicaOperazione(numeri, operazione)` che:

1. Prende un array di numeri e una funzione callback
2. Applica la callback a ogni numero
3. Restituisce un nuovo array con i risultati

Poi crea due callback:

- `raddoppia(n)` che ritorna `n * 2`
- `quadrato(n)` che ritorna `n * n`

Testa:

```
const numeri = [1, 2, 3, 4, 5];

console.log(applicaOperazione(numeri, raddoppia));  // [2, 4, 6, 8, 10]
console.log(applicaOperazione(numeri, quadrato));   // [1, 4, 9, 16, 25]
```

Parametri opzionali

JavaScript **non** controlla il numero di parametri passati a una funzione.

```
function saluta(nome, cognome) {
  console.log('Ciao ' + nome + ' ' + cognome);
}

saluta('Mario');  // "Ciao Mario undefined" (cognome mancante)
saluta('Mario', 'Rossi', 'Extra');  // "Ciao Mario Rossi" (Extra ignorato)
```

Per evitare `undefined`, possiamo specificare i **valori di default** da utilizzare:

```
function saluta(nome = 'Ospite') {
  console.log('Ciao ' + nome);
}

saluta('Mario');    // Ciao Mario
saluta();           // Ciao Ospite
```

Anche le arrow functions supportano i default:

```
const somma = (a = 0, b = 0) => a + b;

somma(); // 0
somma(5); // 5
somma(3, 4); // 7
```

Rest parameters: `...args`

Con i **rest parameters**, possiamo accettare un **numero variabile** di argomenti come array:

```
function sommatoria(...numeri) {
  let totale = 0;
  for (let i = 0; i < numeri.length; i++) {
    totale += numeri[i];
  }
  return totale;
}

sommatoria(1, 2, 3);        // 6
sommatoria(1, 2, 3, 4, 5);  // 15
```

Possiamo anche combinare parametri normali con rest:

```
function bio(firstName, lastName, ...titles) {
  console.log(firstName + ' ' + lastName);
  console.log(titles);
}

bio('Mario', 'Rossi', 'CEO', 'Founder'); // Mario Rossi, ['CEO', 'Founder']
```

A differenza di `arguments`, i rest parameters sono veri array e supportano tutti i metodi degli array.

Inoltre, può essere utilizzato anche nelle arrow functions.

Spread operator: `...array`

Lo **spread operator** (operatore *espansione*) è di fatto il contrario di **rest**: espande un array in singoli elementi.

```
function bio(firstName, lastName, title) {
  return `${firstName} ${lastName}, ${title}`;
}

bio(...['Mario', 'Rossi', 'Founder']); // Mario Rossi, Founder
```

Lo spread operator è utilizzabile ogniqualvolta si ha a disposizione un oggetto iterabile come un array, non necessariamente in un contesto di chiamata di funzione.

Ad esempio, è molto utile per **concatenare** array:

```
const arr1 = [1, 2];
const arr2 = [3, 4];

const merged = [...arr1, ...arr2];  // [1, 2, 3, 4]
```

Esercizio: Unisci e copia

Spread in pratica

Mettiamo in pratica gli operatori visti.

1. **Unione di array**: crea 3 array (`primi`, `secondi`, `terzi`) e uniscili in uno solo usando spread.
2. **Copiare oggetto**: crea un oggetto `persona` con `nome` e `eta`, poi creane una copia con spread. Modifica la copia e verifica che l'originale non cambia.
3. **Aggiungere proprietà**: usa spread per creare un nuovo oggetto partendo da `persona` ma aggiungendo una proprietà `citta`.

Lexical scoping

In JavaScript, lo **scope** è **lessicale**. Questo significa che le regole che determinano la visibilità degli identificatori dipendono solo da come il codice è scritto, e non da ciò che avviene un fase di esecuzione.

Di fatto, le funzioni “vedono” le variabili del contesto in cui sono **definite**, non dove sono **chiamate**.

```
let messaggio = 'Ciao';

function saluta() {
  console.log(messaggio);  // Vede 'messaggio' perché definita qui
}

function altra() {
  let messaggio = 'Addio';  // Variabile locale
  saluta();  // Stampa 'Ciao', non 'Addio'!
}

altra();
```

Closure

Una **closure** (*chiusura*) è una funzione che ricorda le sue variabili esterne ed è in grado di accedervi. In alcuni linguaggi questo non è possibile, oppure è richiesto che la funzione venga scritta in un determinato modo. In JavaScript, tutte le funzioni sono closure di natura.

```
function creaContatore() {
  let contatore = 0;  // Variabile dello scope esterno

  return function() {
    contatore++;
    console.log(contatore);
  };
}

const incrementa = creaContatore();
incrementa();  // 1
incrementa();  // 2
incrementa();  // 3
```

La funzione **ricorda** la variabile `contatore`, anche se `creaContatore()` è finita.

Closure: uso pratico

```
function creaMultiplicatore(moltiplicatore) {
  return function(numero) {
    return numero * moltiplicatore;
  };
}

const raddoppia = creaMultiplicatore(2);
const triplica = creaMultiplicatore(3);

raddoppia(5);  // 10
triplica(5);     // 15
```

Ogni **closure** mantiene il suo proprio `moltiplicatore`.

Closure: uso per incapsulamento

```
function creaBanca(soldi) {
  return {
    deposita: function(importo) {
      soldi += importo;
      return soldi;
    },
    preleva: function(importo) {
      if (importo <= soldi) {
        soldi -= importo;
      }
      return soldi;
    },
    saldo: function() {
      return soldi;
    }
  };
}

const mioBancomat = creaBanca(1000);
mioBancomat.deposita(200);         // 1200
mioBancomat.preleva(500);          // 700
console.log(mioBancomat.saldo());  // 700

console.log(mioBancomat.soldi);    // undefined, non possiamo accedere direttamente a `soldi`
```

Esercizio: Contatore personalizzato

Closure per incapsulamento

Crea una funzione `creaContatore(valoreIniziale)` che restituisce un oggetto con questi metodi:

- `incrementa()` - aumenta il contatore di 1 e lo restituisce
- `decrementa()` - diminuisce il contatore di 1 e lo restituisce
- `reset()` - riporta il contatore al valore iniziale
- `getValue()` - restituisce il valore corrente senza modificarlo

La variabile del contatore deve essere **privata** (accessibile solo tramite i metodi).

Test:

```
const counter = creaContatore(10);

console.log(counter.incrementa());  // 11
console.log(counter.incrementa());  // 12
console.log(counter.decrementa());  // 11
console.log(counter.getValue());    // 11
counter.reset();
console.log(counter.getValue());    // 10
```

Lavorare con gli array

Nella **lezione 2** abbiamo visto le basi di array e oggetti.

Ora vediamo i metodi più utili per lavorare con gli array in modo funzionale.

- Metodi che **iterano** (`forEach`)
- Metodi che **trasformano** (`map`)
- Metodi che **filtrano** (`filter`)
- Metodi che **accumulano** (`reduce`)

Gli altri metodi (`find`, `some`, `every`, `sort`, `reverse`) li vedremo al bisogno.

Come sempre, MDN è un ottimo punto di partenza per esplorare tutti i metodi disponibili.

`forEach`: iterare su ogni elemento

`forEach` esegue una funzione per **ogni elemento** dell'array:

```
const nomi = ['Mario', 'Luigi', 'Anna'];

nomi.forEach(function(nome, indice) {
  console.log(indice + ': ' + nome);
});

// Equivalente con arrow function
nomi.forEach((nome, indice) => {
  console.log(indice + ': ' + nome);
});
```

Nota: `forEach` esegue solo azioni, non restituisce nulla. Per **trasformare** o **filtrare** array, usiamo `map` e `filter`.

`map` e `filter`

`map`: trasformare elementi

`map` **crea un nuovo array** trasformando ogni elemento:

```
const numeri = [1, 2, 3, 4, 5];

// Raddoppia ogni numero
const raddoppiati = numeri.map(n => n * 2);
console.log(raddoppiati);  // [2, 4, 6, 8, 10]

// Estrai nomi da oggetti
const persone = [
  { nome: 'Mario', eta: 30 },
  { nome: 'Luigi', eta: 28 },
  { nome: 'Anna', eta: 32 }
];

const nomiEstratti = persone.map(p => p.nome);
console.log(nomiEstratti);  // ['Mario', 'Luigi', 'Anna']
```

`filter`: filtrare elementi

`filter` **crea un nuovo array** con solo gli elementi che soddisfano una condizione:

```
const numeri = [1, 2, 3, 4, 5, 6];

// Solo numeri maggiori di 3
const grandi = numeri.filter(n => n > 3);
console.log(grandi);  // [4, 5, 6]

// Persone con età >= 30
const adulti = persone.filter(p => p.eta >= 30);
console.log(adulti);  // Mario e Anna
```

Accumulare l’array in un valore

`reduce` **riduce** un array a un singolo valore, usando un accumulatore:

```
const numeri = [1, 2, 3, 4, 5];

// Somma
const somma = numeri.reduce((acc, n) => acc + n, 0);
console.log(somma);  // 15
```

Come funziona?

1. Parte da `acc = 0` (valore iniziale)
2. Somma ogni elemento: `0 + 1 = 1`, poi `1 + 2 = 3`, poi `3 + 3 = 6`, ecc.
3. Ritorna il valore finale: `15`

```
// Moltiplicazione
const prodotto = numeri.reduce((acc, n) => acc * n, 1);
console.log(prodotto);  // 120

// Creare un oggetto conteggio
const colori = ['rosso', 'blu', 'rosso', 'verde', 'blu', 'rosso'];
const conteggio = colori.reduce((acc, colore) => {
  acc[colore] = (acc[colore] || 0) + 1;
  return acc;
}, {});

console.log(conteggio);  // { rosso: 3, blu: 2, verde: 1 }
```

Esercizio: Array methods

Esercizio

Dato questo array:

```
const numeri = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

1. filtra solo i numeri pari.
2. poi, eleva al quadrato i numeri filtrati.
3. infine, raccogli e somma il risultato.

Cos’è un evento?

Come abbiamo visto, JavaScript è un linguaggio multi-paradigma orientato agli eventi (*event-driven*). Ma cosa sono esattamente gli eventi?

Un **evento** è un segnale che sta ad indicare che è avvenuto *qualcosa*:

- **Interazione utente**: click, doppio click, spostamento del mouse, pressione di tasti.
- **Ciclo di vita della pagina**: caricamento della pagina, scroll, ridimensionamento finestra.
- **Attività nel form**: submit, cambio valore, focus, blur.

Tutti i nodi DOM generano questi segnali; inoltre, *gli eventi non sono limitati al DOM*.

Eventi utili

- **Eventi del mouse**:
  - `click`: quando si clicca col mouse su un elemento (i dispositivi touch lo generano tramite il tocco).
  - `contextmenu`: quando si clicca col tasto destro su un elemento.
  - `mouseover` e `mouseout`: quando il cursore passa sopra o abbandona un elemento.
  - `mousedown` e `mouseup`: quando viene premuto o rilasciato il pulsante del mouse su un elemento.
  - `mousemove`: quando si sposta il mouse.
- **Eventi da tastiera**:
  - `keydown` e `keyup`: quando viene premuto e rilasciato un tasto.
- **Eventi del `Document`**:
  - `DOMContentLoaded`: quando l'HTML viene caricato e la costruzione del DOM è stata completata.
- **Eventi dei CSS**: `transitionend`: quando termina un'animazione CSS.

Come sempre, Mozilla e W3Schools offrono ottime risorse per esplorare tutti gli eventi disponibili.

Reagire agli eventi

Gli eventi da soli non fanno nulla, dobbiamo **collegare** una funzione che reagisca quando l'evento accade. Questa funzione è chiamata **event handler** (*gestore* dell'evento) o **event listener**.

Possiamo collegare un handler in **tre modi**:

1. Un gestore può essere impostato con un **attributo HTML** chiamato `on<evento>`, ad esempio:

```
<script>
  function saluta() {
    alert('Ciao!');
  }
</script>
<button onclick="saluta()">Clicca</button>
```

2. Oppure, possiamo assegnare una proprietà dell'elemento DOM chiamata `on<evento>`, ad esempio:

```
<button id="ciao">Clicca</button>
<script>
  const bottone = document.querySelector('#ciao');
  bottone.onclick = function() {
    alert('Ciao!');
  };
</script>
```

3. Infine, possiamo utilizzare il costrutto moderno **`addEventListener`**.

`addEventListener`: ascoltare gli eventi

Il problema dei primi due metodi è che **possono essere usati solo una volta** per ogni evento su un elemento. Se ne assegniamo un secondo, sovrascrive il primo.

`addEventListener` permette di aggiungere **più gestori** allo stesso evento senza sovrascrivere:

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

bottone.addEventListener('click', function() {
  console.log('Bottone cliccato!');
});

bottone.addEventListener('click', function() {
  alert('Bottone cliccato!');
});
```

Rimuovere un listener

Per rimuovere un listener serve **la stessa funzione**:

```
function onClick() {
  console.log('Click');
}

bottone.addEventListener('click', onClick);
bottone.removeEventListener('click', onClick);
```

Attenzione alla registrazione dei listener

La nostra funzione handler deve essere **passata come riferimento**, non invocata:

```
function saluta() {
  alert('Ciao!');
}

const bottone = document.querySelector('button');

// Corretto: passiamo la funzione senza parentesi
bottone.addEventListener('click', saluta);

// Errato: stiamo invocando la funzione subito, invece di passarla come riferimento
bottone.addEventListener('click', saluta());
```

L’oggetto Event

Per gestire correttamente un evento, vorremmo saperne di più su cosa è avvenuto.

A questo scopo, quando si verifica un evento, il browser crea un **oggetto evento** (*event object*) inserisce i dettagli al suo interno e lo passa come argomento alla funzione handler.

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

lista.addEventListener('click', function(event) {
  console.log(event.type);          // 'click'
  console.log(event.target);        // elemento cliccato
  console.log(event.currentTarget); // elemento con il listener
});
```

Alcune proprietà dell’oggetto event sono:

- `event.type`: tipo di evento
- `event.target`: l’elemento *più interno* su cui è *avvenuto* l’evento
- `event.currentTarget`: l’elemento che ha *gestito* l’evento (ovvero su cui è registrato il listener)
- `event.clientX` e `event.clientY`: coordinate del cursore relative alla finestra
- `event.timeStamp`: istante (in millisecondi) in cui l’evento è avvenuto

Prevenire il comportamento di default

Molti eventi vengono gestiti direttamente dal browser (es. `<form>` ricarica la pagina, `<a>` naviga); è possibile bloccare queste gestioni di default e sostituirle con altri comportamenti.

Il modo migliore è usando il metodo `event.preventDefault()`:

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

form.addEventListener('submit', function(event) {
  event.preventDefault();  // Blocca il ricaricamento
  console.log('Form inviato via JavaScript!');
});
```

Un'alternativa, se l'handler viene assegnato tramite `on<event>`, è restituire `false`:

```
<a href="/" onclick="return false">Clicca qui</a>
```

Quest'ultima modalità non funziona con `addEventListener` ed è generalmente sconsigliata.

Event bubbling e propagazione

Quando viene innescato un evento su un elemento:

1. come prima cosa vengono eseguiti gli handler ad esso assegnati,
2. poi ai nodi genitori,
3. ed infine risale fino agli altri nodi antenati.

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

parent.addEventListener('click', () => console.log('Parent cliccato'));
button.addEventListener('click', () => console.log('Button cliccato'));

// Click button → stampa: "Button cliccato", poi "Parent cliccato"
```

Questa dinamica è chiamata **event bubbling** (propagazione dell'evento) ed è utile per gestire eventi su più elementi senza dover assegnare un listener a ciascuno.

Se necessario, possiamo interrompere il bubbling con `event.stopPropagation()`, ma è una pratica da usare con cautela.

Event delegation

Invece di assegnare un listener a **ogni elemento**, possiamo usarne uno solo sul contenitore:

index.html

```html
<div id="menu">
  <button data-action="save">Save</button>
  <button data-action="load">Load</button>
  <button data-action="search">Search</button>
</div>
```

script.js

```js
const actions = {
  save() {
    alert('saving');
  },
  load() {
    alert('loading');
  },
  search() {
    alert('searching');
  },
};

const menu = document.querySelector('#menu');

menu.addEventListener('click', function(event) {
  const action = event.target.dataset.action;
  if (action) {
    actions[action]();
  }
});
```

Esercizio: Il ricettario interattivo

Evoluzione del ricettario

Nella lezione precedente abbiamo costruito il ricettario usando HTML, CSS e JavaScript per iniettare il contenuto nel DOM.

Adesso aggiungiamo **interattività** usando gli eventi, i dataset e gli array.

Partendo dal ricettario della lezione 2, aggiungiamo le seguenti funzionalità:

1. **Evidenziazione di ingredienti**: quando l'utente clicca su un ingrediente (elemento `<li>`), aggiungiamo una classe CSS (es. `.used`) che evidenzia la riga selezionata (usando `classList.toggle`).
2. **Contatore di ingredienti usati**: mostriamo quanti ingredienti sono stati marcati come "usati" (incrementando/decrementando un contatore al click).
3. **Porzioni dinamiche**: aggiungiamo due bottoni (`+` e `-`) per modificare il numero di porzioni (default: 4).
    - Ogni ingrediente deve avere un attributo `data-base-qty` con la quantità per singola porzione.
    - Quando le porzioni cambiano, iteriamo sugli ingredienti e aggiorniamo il valore visualizzato in proporzione.
    - *Suggerimento:* usa uno `<span>` con una classe specifica per racchiudere solo il numero della quantità nell'HTML.