

Laboratorio I
a.a. 2026/2027

Funzioni & Scope

Contenuti

- Cos’è una funzione?
- Funzioni in TypeScript
  - Come si dichiarano
  - Come si usano
- Visibilità delle dichiarazioni (scope)

Funzioni in matematica

- Esempi:
  - $f : \mathbb{R} \to \mathbb{R}$, $f(x) = x^2$
  - $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, $f(x,y) = x+y$
  - $f : \mathbb{N} \to \{0,1\}$, $f(x) = 1$ se x è pari, 0 se x è dispari
  - …
- Un dominio, un codominio e *un'espressione* che definisce i valori in ogni punto del dominio
- I punti del dominio corrispondono ai *parametri* della funzione

Funzioni in matematica (2)

- Vediamo come funziona:



Dichiarare funzioni in TypeScript

- In TypeScript usiamo la keyword “`function`”

```
function ⟨identificatore⟩(⟨parametro⟩:⟨tipo⟩, … ):⟨tipo_restituito⟩
{
    ⟨comandi⟩
    return ⟨espressione⟩
}
```

- I parametri si dichiarano come le variabili, ma senza `let`
- Il valore della funzione viene restituito con l’istruzione `return`
  - Nel comando `return` ⟨*espressione*⟩, l’espressione deve avere tipo ⟨*tipo_restituito*⟩

In modo formale…

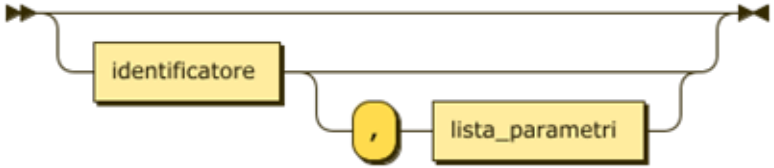
dichiarazione_funzione ::=
'function' *identificatore* '(' *lista_parametri* ')' '{' *comandi* '}'

lista_parametri ::= | *identificatore* | *identificatore* ',' *lista_parametri*

dichiarazione_funzione:

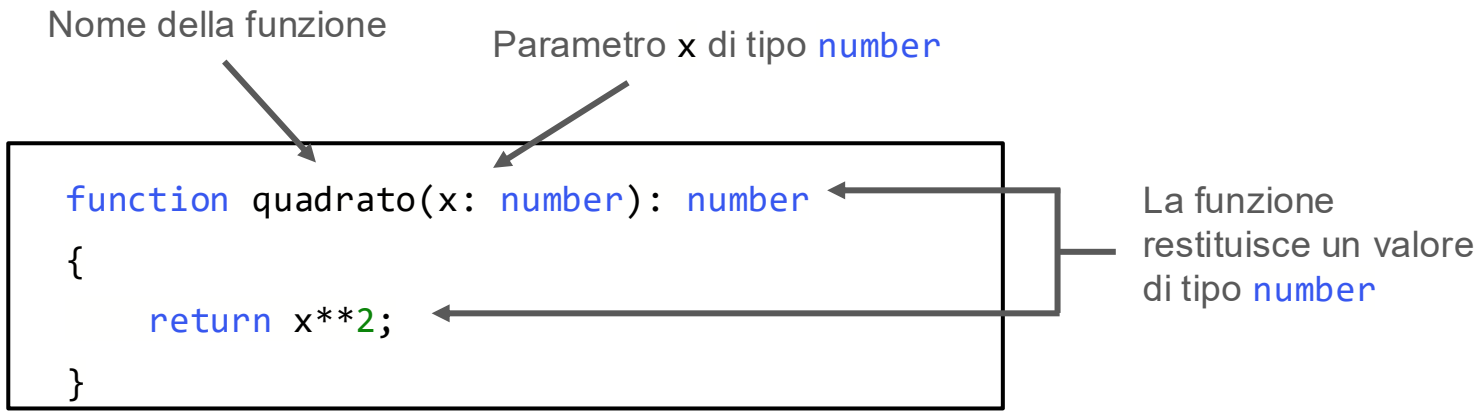


lista_parametri:



Esempio di dichiarazione

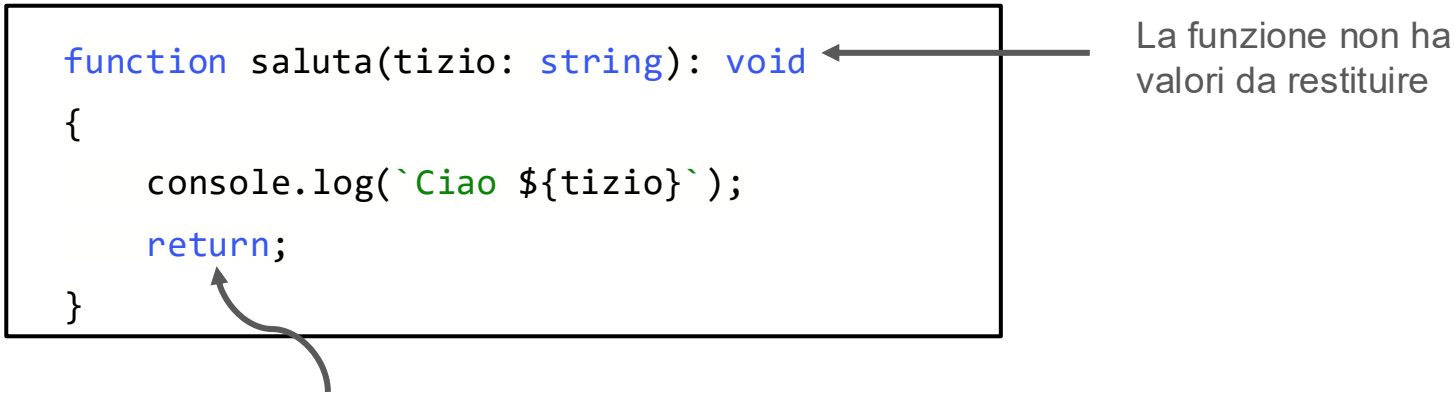
- Vediamo come dichiarare in TypeScript la funzione $f : \mathbb{R} \to \mathbb{R}$, $f(x) = x^2$



- E se non abbiamo nulla da restituire?

Tipo di ritorno `void`

- La keyword “`void`” denota che la funzione non restituisce valori



- Se è l’ultima istruzione di una funzione che restituisce tipo `void`, il `return` si può omettere
  - Non abbiamo comunque alcunché da restituire…
- Generalmente, queste funzioni producono altri effetti oltre a restituire un valore
  - Hanno cioè “side-effects”, ad esempio scrivono qualcosa a console, …

L’istruzione `return`

- Il `return` causa l’uscita dal corpo della funzione in qualsiasi punto venga usato.
- Se omesso in funzioni che restituiscono un tipo diverso da `void`, il compilatore TypeScrit dà errore.

```
function cercaValore(array: number[], valore: number): boolean
{
    for(let elemento of array) {
        if(elemento === valore) {
            return true;
        }
    }
    return false;
}
```

Restituisco subito `true` al chiamante, non ho bisogno di scorrere il resto dell’array

In caso il valore non sia nell’array, ritorno `false`

Chiamare una funzione

- La sintassi per invocare una funzione è

$$\langle nome_funzione \rangle (\langle parametro_1 \rangle, \langle parametro_2 \rangle, \dots)$$

- In un'espressione, la chiamata a funzione viene valutata col valore restituito da `return`, se il tipo restituito non è `void`

```
saluta('Peppe');
   → stampa su console "Ciao, Peppe"


let trovato: boolean = cercaValore([5, 8, 13], 8);
   → trovato = true
```

Vediamo un esempio

1. Il corpo di somma è `{ return a+b }`
2. Viene valutato in un ambiente in cui a=2, b=5
3. Si valuta `return 2+5`
4. Si valuta l'espressione del return, `2+5`→7
5. Si valuta `return 7`
6. L'esecuzione della chiamata termina con valore 7
7. Si valuta `console.log(7)`

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

console.log(somma(2, 5));
```



```
[LOG]: 7
```

Cosa accade durante una chiamata

Un modo per esprimere il processo di valutazione di una chiamata è il seguente:

Il corpo della funzione viene valutato in un ambiente nel quale i nomi dei parametri formali sono associati ai valori dei parametri attuali

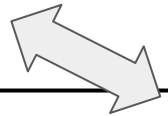
Questa definizione si applica sia al caso in cui il corpo sia una espressione, sia a quello in cui il corpo sia un blocco di comandi fra { … }

… tutti i dettagli nel corso di P&A!

Parametri di una funzione

- Per il chiamante, solo l'ordine dei parametri è rilevante
- I nomi dei parametri sono rilevanti solo nel corpo della funzione

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



```
function somma(x: number, y: number): number
{
    return x + y;
}
```

- Entrambe le definizioni sono equivalenti per il chiamante: `somma(2, 5)`

Parametri con valori di default

- È possibile definire valori di default per i parametri della funzione
  - Possibile solo per i parametri in coda alla lista dei parametri, altrimenti si perde l'ordinamento

```
function saluta(tizio: string, saluto: string = 'Ciao'): void
{
    console.log(`${saluto} ${tizio}`);
}
```

- Se omesso dal chiamante, il parametro avrà il valore di default

```
saluta('Peppe', 'Ehi');
   → stampa su console "Ehi, Peppe"
saluta('Peppe');   // equivalente a saluta('Peppe', 'Ciao')
   → stampa su console "Ciao, Peppe"
```

Scope: visibilità delle dichiarazioni

Il concetto di “scope”

- Con costrutti per il controllo di flusso (`if-else`, `for`, `while`) e funzioni abbiamo visto come costruire codice piuttosto complesso.
- In programmi grandi, devo dare nomi diversi a tutte le variabili?
  - Immaginiamo di dover controllare di non aver già usato “i” in tutto il resto del codice…
- In realtà no, grazie al concetto di *scope* o visibilità di una dichiarazione.

Visibilità delle dichiarazioni

Un nome dichiarato è **visibile** in certe parti del programma, più o meno ampie, a seconda di *dove* è stato dichiarato e di *quale tipo* di dichiarazione viene usata.

TypeScript prevede tre **scope** o **ambiti di visibilità**:

- Scope globale
- Scope di funzione
- Scope di blocco

```
let n: number = 3;

function inverti(a: number[]): number[] {
  let r: number[] = [];
  for (let i: number = 0; i < a.length; i++) {
    let n: number = a[a.length - i - 1];
    r.push(n);
  }
  return r;
}

function mul(a:number, b: number): number {
    return a * b;
}

n = mul(n, n+1);

let arr: number[] = [4,7,12,3,22,1,6];
console.log(inverti(arr));
```

Visibilità delle dichiarazioni

Le **dichiarazioni** fatte fuori da qualunque funzione avvengono nello **scope globale**.

Queste dichiarazioni sono **visibili** in tutto il programma, sia fuori che dentro le funzioni o i blocchi.

Le variabili non dichiarate dal programma, ma usate senza dichiarazione, si intendono implicitamente dichiarate nello scope globale.

- Scope **globale**
- Scope di funzione
- Scope di blocco

```
let n: number = 3;

function inverti(a: number[]): number[] {
  let r: number[] = [];
  for (let i: number = 0; i < a.length; i++) {
    let n: number = a[a.length - i - 1];
    r.push(n);
  }
  return r;
}

function mul(a:number, b: number): number {
    return a * b;
}

n = mul(n, n+1);

let arr: number[] = [4,7,12,3,22,1,6];
console.log(inverti(arr));
```

Visibilità delle dichiarazioni

Ogni funzione definisce il proprio **scope di funzione**.

I parametri formali si intendono dichiarati nello scope di funzione

Ma il nome della funzione no: è nello scope globale!

- Scope globale
- Scope di **funzione**
- Scope di blocco

```
let n: number = 3;

function inverti (a: number[]): number[] {
  let r: number[] = [];
  for (let i: number = 0; i < a.length; i++) {
    let n: number = a[a.length - i - 1];
    r.push(n);
  }
  return r;
}

function mul(a:number, b: number): number {
    return a * b;
}

n = mul(n, n+1);

let arr: number[] = [4,7,12,3,22,1,6];
console.log(inverti(arr));
```

Visibilità delle dichiarazioni

Ogni blocco (0 o più comandi racchiusi fra {...}) definisce il proprio **scope di blocco**.
In particolare, il corpo di una funzione è un blocco.

Un blocco può essere contenuto in un altro blocco. Le dichiarazioni all'interno di un **`for`** (ma anche di **`if`** e **`while`**) si considerano fatte nel suo blocco.

Le **dichiarazioni** fatte all'interno di un blocco con **`let`** e **`const`** sono **visibili** nello scope di quel blocco, ma non fuori.

- Scope globale
- Scope di funzione
- Scope di **blocco**

```
let n: number = 3;

function inverti(a: number[]): number[] {
  let r: number[] = [];
  for (let i: number = 0; i < a.length; i++) {
    let n: number = a[a.length - i - 1];
    r.push(n);
  }
  return r;
}

function mul(a:number, b: number): number {
    return a * b;
}

n = mul(n, n+1);

let arr: number[] = [4,7,12,3,22,1,6];
console.log(inverti(arr));
```

Annidamento e *shadowing*

Notate che gli scope sono contenuti uno nell'altro:

- Lo scope globale contiene 0 o più scope di funzione e 0 o più scope di blocco
- Ogni scope di funzione contiene 0 o più scope di funzione, e 1 o più scope di blocco
- Ogni scope di blocco contiene 0 o più scope di funzione e 0 o più scope di blocco

Quando TypeScript incontra una dichiarazione di un identificatore, definisce una nuova variabile con quel nome e inserisce il nome nel blocco corrispondente.

Quando TypeScript incontra un riferimento a un identificatore, lo interpreta come un riferimento alla dichiarazione più interna con lo stesso nome, partendo dalla posizione nella quale incontra il riferimento.

Annidamento e *shadowing*

```
let x: number = 1;

function f(): void {
  console.log(x);
  for (let i=0; i<10; i++) {
    let x: number = i*2;
    g(x+1);
  }
}

function g(n: number): void {
  let f = x * n;
  console.log(f);
}

console.log(x);
f();
```

La regola del “dall’interno all’esterno” fa sì che sia possibile ri-usare lo stesso nome all’interno di uno scope senza troppi problemi.

La variabile esterna non è più visibile (quella interna con lo stesso nome la “copre”, *shadowing*)

I nomi dichiarati dentro uno scope sono ***locali*** o ***privati*** allo scope.

“Da dentro si vede fuori, ma da fuori non si vede dentro”

Q & A

Esercizi

1. Definire una funzione `replace(arr, target, replacement)` che, dato un array `arr`, ritorni un nuovo array dove tutte le istanze di `target` sono sostituite da `replacement`. Non modificare l'array originale.
2. Definire una funzione `contamaggioredi(arr, threshold)` che ritorni il numero di elementi di `arr` maggiori strettamente di `threshold`.
3. Dati due array `x` e `y` ad n componenti, definire `prodotto_scalare(x, y)` che ritorna la somma di `x[i]*y[i]`. Se le lunghezze differiscono, ritorna `NaN`. Formula: `x · y = somma_{i=1..n} x_i * y_i`
4. Definire `clip(arr, threshold, replacement)` che ritorni un nuovo array dove ogni elemento > `threshold` viene sostituito con `replacement`. Usare come valore di default per il parametro `replacement` la soglia `threshold`.

Esercizio Extra

1. (Statistiche con funzioni) Leggere da tastiera una sequenza di numeri positivi, interrompendosi quando l'utente inserisce 0, e memorizzarla in un array. Definire le seguenti funzioni e utilizzare l'ultima per visualizzare le statistiche sulla sequenza
   a. `somma(a: number[]): number`
   b. `minMax(a: number[]): [number, number] //(restituisce minimo e massimo)`
   c. `media(a: number[], cifre: number = 2): number //(media arrotondata al numero di cifre decimali, 2 di default)`
   d. `stampaRiepilogo(a: number[]): void //(stampa numeri e statistiche)`