

Laboratorio I
a.a. 2026/2027

Oggetti, comando type, interfacce

Contenuti

- Oggetti in TypeScript
  - Denotazione
  - Esempi, uso come dizionari
- Comando `type` e interfacce
- Esercizi sugli oggetti

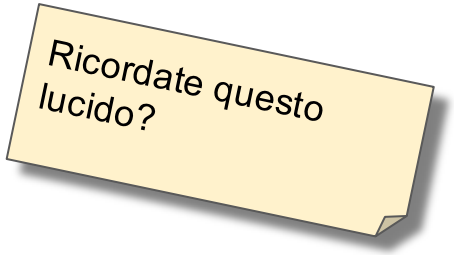
Recap: Tipi di dato primitivo e valori letterali in JS/TS

JavaScript consente di denotare **valori letterali** di determinati tipi.

I seguenti sono i corrispondenti tipi in TypeScript:

- `boolean`: `true`, `false`
- `undefined` : `undefined` (→ **non è stato definito** nessun valore)
- `null`: `null` (→ so che **non c'è** un valore)
- `number`: `1`, `5`, `-53.38`, `12.3e4` (numeri "normali", $-(2^{53}-1) \leq n \leq 2^{53}-1$)
- `bigint`: `900719925474343441698716109 92n` (interi a precisione infinita)
- `string`: `"ciao ugo"`, `"हिन्दू"`, `'hello'`, `` `x vale ${x}` `` (sequenze di caratteri)

Tipi di dato **non** primitivi



Ci sono poi tipi di dato non primitivi, che in TypeScript ricadono sotto “Object”, che vediamo oggi:

- **Object**: `{ nome:“Ugo”, età:19 }` (mappe da chiavi a valori - anche dizionari)
  - **Array**: `[ 1, 5, 8, 12, 21, 33 ]`, `[ 1, “ciao!”, [ “banana” ] ]` (liste con indici numerici)
  - **Function**: `x=>2*x`, `(a,b)=>a+b`, `x => { if (x>0) return x; else return -x }` (funzioni)

Oggetti (o *dizionari*)

Un **oggetto** (o dizionario) è una *mappa* da **chiavi** a **valori**

Qui *mappa* è usato nel senso matematico: in pratica, è una funzione matematica il cui dominio è l'insieme di tutte le chiavi, e il codominio è l'insieme di tutti i valori possibili in TypeScript.

Potremmo anche dire che un oggetto è un insieme di coppie (chiave, valore), dove ogni chiave compare una volta sola! TS segnala errore per chiavi duplicate.

Per esempio, un oggetto *pippo* potrebbe essere indicato in stile matematico con

pippo = { ("passione", "Fondamenti"), ("età", 35), ("capelli", true) }

Oggetti (o *dizionari*)

Per esempio, un oggetto *pippo* potrebbe essere indicato in stile matematico con

pippo = { ("passione", "Fondamenti"), ("età", 35), ("capelli", true) }

In TypeScript, la *denotazione* per i valori di tipo oggetto è un po' più comoda:

pippo : tipo_oggetto = **{** passione**:** "Fondamenti"**,** età**:** 35**,** capelli**:** true **}**

Gli oggetti consentono di raccogliere in un solo valore (di tipo oggetto) molti valori diversi (di qualunque tipo), assegnando a ciascuno di essi un nome (la chiave)

Alcuni esempi di letterali di tipo oggetto

{ } – oggetto vuoto

{ nome: "Pippo", età: 35 } – il caso più comune

{ x } – *espansione di variabile:* se *x* vale 5, è come { x: 5 }

{ "nome": "Pippo", "età" : 35 } – le chiavi possono essere stringhe

{ "!!": 5, "😎": true } – anche stringhe che non sono identificatori

{ 0: 6, 1: 4, 2: 12 } – le chiavi possono essere numeri

{ ["pip"+fiume]: true } – *chiave calcolata:* se *fiume* vale "po", è come { pippo: true }

Tipo di un Oggetto

Per denotare il tipo di un oggetto in TypeScript, si utilizza una sintassi simile a quella dell'oggetto stesso, indicando per ogni chiave il tipo del suo valore.

Ad esempio, il tipo dell'oggetto

```
{ passione: "Fondamenti", età: 35, capelli: true }
```

si può esprimere mediante

```
{ passione: string, età: number, capelli: boolean }
```

```
let obj : {chiave1: string, chiave2: number} = {chiave1: "abc", chiave2: 42};
```

Tipo di un Oggetto (2)

Per denotare il tipo di un oggetto in TypeScript, si utilizza una sintassi simile a quella dell'oggetto stesso, indicando per ogni chiave il tipo del suo valore.

Esempi di errore di dichiarazione di oggetti segnalati da TS

```
let p: { nome: string, età: number } = { nome: "Ugo" };
// errore: manca la proprietà 'età'
let q: { nome: string, età: number } = { nome: "Ugo", età: 19, altezza: 180 };
// errore: 'altezza' non esiste nel tipo
```

Tipo di un Oggetto (3)

Per definire il tipo di un oggetto omogeneo (con tutti i valori dello stesso tipo) del quale **non** sono noti a priori i nomi delle chiavi, si può usare il tipo:

```
{[chiave: string]: boolean}
```

Tipo di un Oggetto (3)

Per definire il tipo di un oggetto omogeneo (con tutti i valori dello stesso tipo) del quale **non** sono noti a priori i nomi delle chiavi, si può usare il tipo:

```
{[chiave: string]: boolean}
```

ATTENZIONE: `[chiave: string]: boolean` indica che **tutte le chiavi hanno valori boolean**. Con questa notazione non è quindi possibile mescolare chiavi con valori diversi, ma è possibile specificare il nome di altre chiavi con lo stesso tipo:

```
{[chiave: string]: boolean, capelli: boolean} // OK!
{[chiave: string]: boolean, eta: number}  // ERRORE!
```

Tipo di un Oggetto: disclaimer

In TypeScript, esiste anche il tipo generico **`object`**, che denota qualunque valore non primitivo (array, funzioni, oggetti):

```
pippo:object = {passione:"Fondamenti", età:35, capelli:true}
```

Dato che uno degli scopi del corso è quello di imparare una **corretta e accurata tipizzazione, non ammetteremo** l'utilizzo di **`object`** come tipo.

Tra l'altro, avendo **`pippo:object`** non si può nemmeno leggere, ad esempio, **`pippo.passione`**, perché il tipo `object` non dichiara alcuna proprietà specifica, e pippo.passione verrebbe bloccato subito in fase di compilazione.

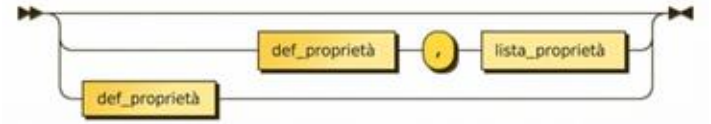
Costanti letterali di tipo oggetto - Sintassi

letterale_oggetto ::= '**{**' *lista_proprietà* '**}**'
lista_proprietà ::= | *def_proprietà* | *def_proprietà* '**,**' *lista_proprietà*
def_proprietà ::= identificatore | *chiave* '**:**' *espressione*
chiave ::= identificatore
| *letterale_stringa*
| *letterale_numero*
| '**[**' *espressione* '**]**'

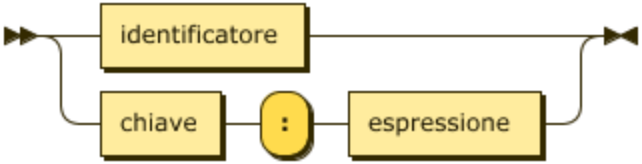
letterale_oggetto:



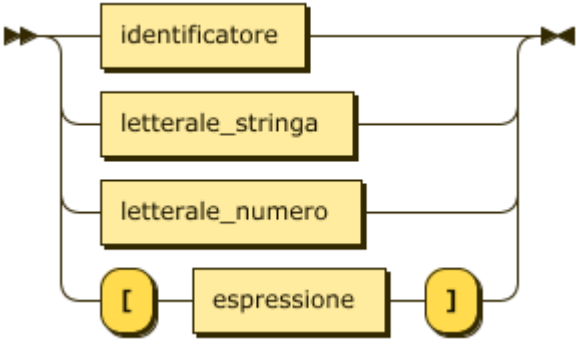
lista_proprietà:



def_proprietà:



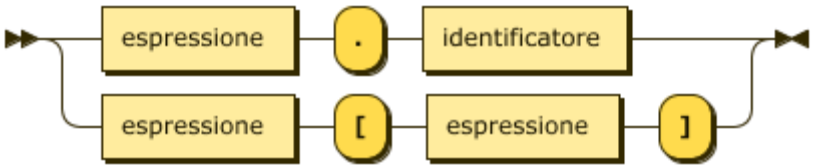
chiave:



Operazioni sugli oggetti

L'operazione principale sugli oggetti è l'**accesso** a un membro (chiave)

espressione_accesso:



espressione_accesso ::= espressione '.' *identificatore*
| *espressione* '[' *espressione* ']'

```
let pippo: {nome:string, età:number, capelli:boolean} = {nome:"Filippo", età:35, capelli:true}

pippo.nome → "Filippo"        pippo["nome"] → "Filippo"              pippo.età → 35

if (pippo.capelli) {
     pippo.età++
     pippo.nome="Guendalino"}

pippo → { capelli: true, età: 36, nome: "Guendalino" }

pippo.occhi → Property 'occhi' does not exist on type '{ nome: string;
                                                         età: number; capelli: boolean;}'
```

Operazioni sugli oggetti

Altre operazioni utili: **accedere a chiavi e valori**

- Controllo se una chiave esiste: *chiave* **in** *oggetto*
- Array contenente tutte le chiavi: *Object.keys(oggetto)*
- Array contenente tutte le coppie chiave-valore: *Object.entries(oggetto)*
- Scorrere le chiavi: **for** (*k* **in** *oggetto*) …

```
let pippo:{nome:string, età:number, capelli:boolean} = {nome:"Filippo", età:35, capelli:true}

"età" in pippo → true                    "occhi" in pippo → false

for (let k in pippo){
     console.log(k) → "nome","età","capelli"
     console.log(pippo[k]) → ERRORE!
}

Object.keys(pippo) → ["nome","età","capelli"]
Object.entries(pippo) → [["nome", "Filippo"], ["età", 35], ["capelli", true]]

pippo.zampe = 4 → ERRORE! Non è omogeneo..
```

Operazioni sugli oggetti (omogenei)

Altre operazioni utili **per oggetti omogenei** (oltre alle precedenti):

- Scorrere le chiavi e i valori: **for** (*k* **in** *oggetto*) …
- Aggiungere una chiave:
  - *oggetto.chiave=valore*, oppure *oggetto[“chiave”] = valore*

```
let pluto:{[chiave:string]:number} = {età:12}

"età" in pluto→ true                              "occhi" in pluto→ false

for (let k in pluto){
     console.log(k) → “età”
     console.log(pluto[k]) → 12
}

pluto.zampe = 4 → pluto = {età:12, zampe:4}

Object.keys(pluto) → [“età”,"zampe"]
Object.entries(pluto) → [["età", 12], ["zampe", 4]]
```

Definire nuovi tipi con `type`

Abbiamo visto come la tipizzazione di un oggetto ne rispecchia la sintassi; se vogliamo però definire tanti oggetti dello stesso tipo, potrebbe essere scomodo ripetere la tipizzazione dell'oggetto!

In TypeScript, si possono in generale **definire nuovi tipi** mediante il comando `type`, che possono successivamente essere utilizzati con la sintassi usuale:

```
type NumArray = number[];
type MyObject = {nome: string, età: number};

let a: NumArray = [12, 31, 1];
let pippo: MyObject = {nome: "Pippo", età: 35};
```

Definire strutture degli oggetti mediante interfacce

Mentre `type` permette di dare un nome a un tipo qualsiasi, TypeScript possiede un costrutto specifico per definire la **struttura** di un oggetto: le **interfacce.**

```
interface MyObj {
 nome: string,
 età: number
}

let pippo: MyObj = {nome: "Pippo", età: 35};
```

Un'interfaccia rappresenta un "contratto": TypeScript controlla che gli oggetti di quel tipo rispettino la struttura dichiarata. Per descrivere semplici oggetti, è equivalente a usare type; vedremo ulteriori funzionalità delle interfacce quando parleremo di classi.

Q & A

Esercizio 2 degli array - stavolta con gli oggetti!

(**Inventario**) Si scriva un programma per gestire un inventario, formato da un insieme di coppie prodotto-quantità. Tali coppie vengono lette da tastiera, fino a quando l'utente inserisce "stop" come nome del prodotto. Stampare la lista dei prodotti inventariati, con le rispettive quantità, su una sola riga.

Esempio

Input: latte, 2, uova, 6, pane, 7, latte, 3, uova, 6, stop

Output: (latte, 5) (uova, 12) (pane, 7)

Esercizio 3 degli array - stavolta con gli oggetti!

(Media dei voti) Si scriva un programma che, dato un insieme di studenti con i rispettivi voti di 3 esami, calcoli e visualizzi la media aritmetica dei voti di ciascuno. L'utente fornirà il nome di ogni studente, seguito dai suoi 3 voti. L'acquisizione termina quando l'utente inserisce "stop" come nome di uno studente.

Esempio

Input: Luca, 28, 30, 27, Marta, 30, 29, 30, Marco, 18, 22, 25, stop

Output: Luca: 28.33
Marta: 29.67
Marco: 21.67

Esercizio Extra

Un negozio registra i suoi prodotti utilizzando un oggetto TypeScript: ogni chiave è il nome di un prodotto, che ha come valore associato un altro oggetto con chiavi:

- **prezzo**, associato al prezzo unitario del prodotto, e
- **vendite**, associato ad un array contenente le quantità vendute nei diversi giorni.

Scrivere una funzione **analizzaVendite** che, dato un oggetto non vuoto con questa struttura, restituisce una tupla contenente 1) l'incasso complessivo e 2) il nome del prodotto con il maggior numero totale di pezzi venduti, in questo ordine. In caso di parità, restituire il prodotto alfabeticamente più piccolo.

Esempio per l’esercizio extra

Input: { “quaderno”: { prezzo: 3, vendite: [2, 4] }, “penna”: { prezzo: 2, vendite: [5, 3, 1] }, “gomma”: { prezzo: 1, vendite: [1, 2, 0] } }

Output: [39, "penna"] Incasso totale: 3*(2+4) + 2*(5+3+1) + 1*(1+2+0) = 39

Input: {"matita": { prezzo: 1, vendite: [2, 4] }, "penna": { prezzo: 2, vendite: [5, 1]}}))

Output: [18, "matita"] Parità di pezzi venduti, ma “matita” < “penna”