Typescript

Javascript è un linguaggio tipizzato che non prevedere controllo statici sui tipi di dato. **Typerscript invece effettua i controlli sui tipi.**

Typescript effettua una ***traspilazione*** di un programma in ***.ts*** in uno javascript ***.js***

> 📍 **Transpilers (traspilazione)**, o *compilatori source-to-source*, sono strumenti che leggono codice sorgente in un linguaggio di programmazione e producono codice equivalente in un altro linguaggio che ha un simile livello di astrazione

Tipi base

`let nomeVariabile : type = valoreVariabile`

- **boolean**
  - `let flag : boolean = false;`
- **number**
  - `let sum : number = 42;`
- **bigint**
  - `let big : bigint = 100n;`
- **string**
  - `let signoreDegliAnelli3 : string = “Il ritorno del re”;`
- **array** (dati di tipo **omogeneo**)
  - `let list : number[] = [1, 2, 3];`

TIPI TUPLA

I tipi **tupla** permettono di esprimere array eterogenei.Ad esempio, tupla con **string** e **number**

```
let arr: [string, nummber];

arr = ["ciao", 10];   // NON DA ERRORE
arr = [10, "ciao"];   // DA ERRORE


(Type 'number' is not assignable to type 'string'.)
(Type 'string' is not assignable to type 'number'.)
```

- quando si accede agli elementi al di fuori degli indici noti da errore
- quando si accede agli elementi tramite indice viene effettuato il controllo sui tipi

TIPO ENUM

In TypeScript, un tipo `enum` è un tipo di dato che consente di definire un insieme di costanti denominate.

```
enum Color {
  Red,
  Green,
  Blue
}
```

```
let myColor: Color = Color.Green;
```

Le costanti denominate di un enum sono numerate in modo implicito a partire da 0, ma è possibile specificare manualmente i valori numerici assegnati a ciascuna costante.

```
enum Color {
  Red = 1,
  Green = 2,
  Blue = 4
}
```

▼ Tipi speciali

TIPO ANY

Quando si dichiara una variabile in JavaScript, stiamo implicitamente affermando che **la variabile potrà contenere qualsiasi tipo di dato**

Questa assunzione implicita in JavaScript può essere resa esplicitamente in TypeScript specificando il tipo `any`

```
var myVar: any = 123;

myVar = "Una stringa";
myVar = true;
```

```
var myArray: any[] = ["stringa", 123, true];
```

> **myVar** e **myArray** possono contenere dinamicamente **qualsiasi valore** proprio come avviene nel puro JS

📍 Usando **any** si perde controllo sui tipi

TIPO RECORD

Rappresenta una mappa, ovvero un oggetto che associa chiavi a valori. Quindi dove il primo elemento di `record` è la chiave dell'oggetto che vogliamo restituire e il secondo elemento è il valore associato alla chiave

```
Record <string, number> = {}
oppure
Record <string, string> = {}
```

```
OUTPUT
record<nome, mario> = {} -> {nome: mario}
```

TIPO UNKNOWN

Si riferisce a variabili di cui non conosciamo il tipo staticamente (al tempo della scrittura) Esempio simile a **any**

```
let nonSicuro: unknown = 4;
nonSicuro = "forse una stringa";


// OK, un boolean
nonSicuro = false;
```

> A differenza di **any**, non permette di accedere proprietà che non esistono
> Simile a generics (meglio generics)

TIPO VOID

In un certo senso è l'opposto di **any.** Indica la mancanza di un tipo, o meglio l'assenza di un valore a cui poter assegnare un tipo

Si usa per indicare che una funzione non restituisce nulla

Ha poco senso usarlo su una variabile, perché potrebbe avere come valori solo **null** (se non usato --`strictNullChecks`) o `undefined`

```
let inusabile: void = undefined;
```

TIPO NEVER

Il tipo `never` rappresenta il tipo di valore di funzioni che non dovrebbero essere terminate normalmente. Ad esempio, una funzione che lancia un'eccezione o entra in un loop infinito.

```
function infiniteLoop(): never {
    while (true) {
    }
}
```

TIPO UNION

Il tipo `union` viene utilizzato per permettere ad una variabile di contenere valori di diversi tipi.

```
let a: number | string;

a = 10; // OK
a = "ciao"; // OK
a = true; // ERRORE
```

TIPO ALIAS

Il tipo alias viene utilizzato per creare un alias a un tipo esistente. con la parola chiave `type`

```
type MyNumber = number;

let a: MyNumber = 10;
```

TIPO INTERSECTION

Il tipo `intersection` viene utilizzato per combinare due o più tipi.

```
type A = { name: string };
type B = { age: number };
type C = A & B;

let c: C = { name: "Marco", age: 40 }; // OK
```

TIPO READONLY

Il tipo **readonly** viene utilizzato per dichiarare una variabile che non può essere modificata dopo la sua inizializzazione.

```
interface Person {
  readonly name: string;
  age: number;
}

let person: Person = { name: "Marco", age: 40 };
person.age = 41;
person.name = "Giuseppe"; // ERRORE
```

TIPI NULL E UNDEFINED

In TypeScript, `null` e `undefined` hanno i loro tipi come si ha in JavaScript:

```
let numero: number = undefined;
let stringa: string = null;
```

Tuttavia, ci sono alcune configurazioni che possono essere utilizzate per migliorare la gestione di questi due valori.

Il tipo `null` può essere utilizzato come valore di default quando si utilizza l'operatore `?` per rendere opzionale un parametro.

```
function funzione(parametro: string | null | undefined) {
  parametro = parametro ?? "valore di default";
}
```

Il tipo `undefined` viene utilizzato per indicare che una variabile non è stata inizializzata.

```
let a: number | undefined;

console.log(a); // undefined
a = 10;
console.log(a); // 10
```

Con `?` il tipo di una valore può essere `undefined` oltre al tipo che abbiamo già messo, ad esempio string

Con `!` si indica che il seguente valore non può essere in quel punto del codice `undefined` o `null`

Interfacce

TypeScript consente la definizione di una **interfaccia** per definire la **struttura** di un oggetto, ovvero quali proprietà e metodi deve avere.

Ad esempio, se si ha un oggetto "Persona" che ha le proprietà "nome" e "età", si può definire un'interfaccia come segue:

```
interface Persona {
  nome: string;
  eta: number;
}
```

L'interfaccia "Persona" indica che qualsiasi oggetto che implementa questa interfaccia deve avere una proprietà "nome" di tipo stringa e una proprietà "età" di tipo numero.

In questo modo, si possono definire **tipi di oggetti complessi** e riutilizzabili

Estensioni interfacce

In TypeScript, l'estensione di un'interfaccia (interface extension) permette di definire una nuova interfaccia che eredita le proprietà e i metodi di un'altra interfaccia già definita.

Ad esempio, supponiamo di avere l'interfaccia "Persona" definita come segue:

```
interface Persona {
  nome: string;
```

```
  eta: number;
}
```

Si può definire un'altra interfaccia "Studente" che estende "Persona" e ha una proprietà "corso" come segue:

```
interface Studente extends Persona {
  corso: string;
}
```

In questo modo, l'interfaccia "Studente" ha tutte le proprietà dell'interfaccia "Persona" (cioè "nome" e "età") e una nuova proprietà "corso". L'interfaccia "Studente" può essere utilizzata per definire oggetti che rappresentano studenti, ma che includono anche le proprietà dell'interfaccia "Persona".

📍 Le estensioni di interfaccia sono utili per creare gerarchie di interfacce, in cui le interfacce più specializzate estendono le interfacce più generiche e riutilizzano le proprietà e i metodi già definiti. In questo modo, si può ottenere un codice più **leggibile**, **mantenibile** e **flessibile**.

Generics

Sono un meccanismo che permette di definire funzioni e classi che possono lavorare con diversi tipi di dati, senza specificare il tipo di dati a priori. In altre parole, i generics permettono di creare codice riutilizzabile e flessibile, in grado di gestire diversi tipi di dati in modo generico.

```
function identity(arg: number): number {
 return arg;
}
```

Senza **generics**, per poter utilizzare la funzione identità con altri tipi, dovremmo scriverne una per ogni possibile tipo

> *Ad esempio quando vogliamo creare una struttura dati e non vogliamo andare a specificare di che tipo sarà dato che non lo sappiamo*

Si utilizza la sintassi `<T>`, dove `T` è un identificatore che rappresenta il tipo generico.

```
function identity<T>(arg: T): T {
  return arg;
}
```

Abbiamo usato una **type variable T**, che permette di *catturare* il tipo del dato passato dal chiamante (ad es., **number**), per poterlo poi usare dopo

Una volta definita, la funzione generica si può invocare in due modi:

```
let output = identity<string>("myString");
//       ^ = let output: string
```

```
let output = identity("myString");
//       ^ = let output: string
```

Se però invece di lavorare con tutti i tipi `<T>`, vogliamo limitarci solo a quelli che hanno la proprietà `length` disponibile

```
function loggingIdentity<T>(arg: T): T {
  console.log(arg.length); // KO: Property 'length' does not exist on type 'T'
  return arg;
}
```

Per evitare ciò, creiamo delle interfacce che riguardano le proprietà dei tipi che vogliamo usare, così da fare “distinzione”

```
interface Lengthwise {
 length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
 console.log(arg.length); // Ora <T> ha la proprietà .length
 return arg;
}

// OUTPUT
loggingIdentity({ length: 10, value: 3 });
```

Interfacce con i generics

```
interface Nodo<T> {
 value: T;
 next: Nodo<T> | null;
 prev: Nodo<T> | null;
}
```

```
let n1: Nodo<number> = {value: 1, next: null, prev: null};
```

Una classe generica ha la stessa forma di una interfaccia generica

```
interface Pizza {
  name: string;
  toppings: string[];
}
```

```
class Pizza {
  name: string;
  toppings: string[];
}
```

La differenza è che l'Interfaccia usata in TS è solo per scopi legati al type check (dopo il *transpiling*, non esiste più), invece da una classe si possono creare oggetti, e la struttura che definite continua ad esistere in JS (dopo il *transpiling*)

Visibilità

TS introduce public, protected e private

- `public`: visibilità di default
- `protected`: visibile alla classe o alle sottoclassi
- `private`: visibile solo alla classe

```
class Greeter {
 private a: number = 0;
 public greet() {
   console.log("Hello, " + this.getName());
 }
 protected getName() {
   return "hi";
 }
}
```

Tipo di una funzione

Il "tipo di una funzione" si riferisce al tipo di dato che descrive il formato della funzione, ovvero il tipo di input che accetta e il tipo di output che produce.

Per definire il tipo di una funzione in TypeScript, si utilizza la seguente sintassi:

```
function nomeFunzione(parametro1: tipo1, parametro2: tipo2): tipoOutput {
  // corpo della funzione
};
```

```
let myAdd = function (x: number, y: number): number {
 return x + y;
```

```
};
```

Segnatura di una funzione

Si riferisce al formato della dichiarazione della funzione, che descrive i suoi parametri di input e il tipo di dato di ritorno. La segnatura di una funzione include il nome della funzione, il tipo di ogni parametro di input e il tipo di dato di ritorno.

```
let myAdd2: (baseValue: number, increment: number) => number = function (x, y) {
 return x + y;
};

// baseValue e x sono la stessa cosa ma baseValue viene ridefinito per avere un
// codice più leggibile
```

Parametri opzionali di una funzione

In TS tutti i parametri attesi da una funzione sono richiesti al momento della chiamata

```
function buildName(firstName: string, lastName?: string) {
 if (lastName) return firstName + " " + lastName;
 else return firstName;
}

// OUTPUT
let result1 = buildName("Bob"); // OK
let result2 = buildName("Bob", "Adams", "Sr."); // KO, troppi parametri
let result3 = buildName("Bob", "Adams"); // OK
```

Nel primo output questo è corretto perché abbiamo messo `lastname` opzionale, ovvero con `?` infatti questo carattere chiave indica che `lastname` può essere sia `string` che `undefined`, ovvero diventa opzionale.

Inoltre è possibile fornire valori di inizializzazione ai parametri, usati se il parametro non viene fornito o se **`undefined`**

> Quelli con *inizializzazione* che vengono dopo tutti quelli richiesti sono trattati come *opzionali* (e quindi possono essere omessi in chiamata)
>
> In questo caso, la funzione ha lo stesso tipo di quella con parametro opzionale

```
function buildName(firstName: string, lastName = “Woman”) {
 if (lastName) return firstName + " " + lastName;
 else return firstName;
}

// OUTUPUT
let result1 = buildName("Wonder"); // “Wonder Woman”
let result2 = buildName("Wonder", undefined); // “Wonder Woman”
let result3 = buildName("Bob", "Adams", "Sr."); // KO, troppi parametri
let result4 = buildName("Bob", "Adams"); // “Bob Adams”
```

Nel secondo output si può vedere proprio come venga restituito il valore messo di default da noi se in caso quel parametro della funzione non viene restituito.

Quindi, se uno di questi parametri precede uno richiesto, deve sempre essere passato (eventualmente come undefined)

```
function buildName(firstName = "Will", lastName: string) {
 return firstName + " " + lastName;
}

// OUTPUT
let result1 = buildName("Bob"); // KO, pochi parametri
let result2 = buildName("Bob", "Adams", "Sr."); // KO, troppi parametri
let result3 = buildName("Bob", "Adams"); // OK -> "Bob Adams"
let result4 = buildName(undefined, "Adams"); // OK -> "Will Adams"
```

```
Tipo della funzione:
```

```
(firstName: string | undefined, lastName: string): string
```

Inferenza di tipo

Typescript applica una strategia di **type inference** quando non viene utilizzata alcuna annotazione esplicita sul tipo utilizzato

```
let x = 3;
//  ^ = let x: number
```

```
let x = [0, 1, null];
//  ^ = let x: (number | null)[]
```

L'algoritmo di inferenza dei tipi considera tutti i possibili "tipi candidati" (in questo caso **number** e **null**), e sceglie quello **migliore** (in questo caso il *tipo unione*)

📍 L'inferenza, se necessaria, avviene ad esempio quando si *inizializzano variabili* o quando si determinano *i valori di ritorno di una funzione*