Typescript

Un superset di JavaScript

Tipizzazione statica

Una delle caratteristiche principali di TypeScript è la tipizzazione statica. Ciò significa che è possibile specificare il tipo di una variabile durante la dichiarazione, e questo tipo rimarrà costante durante l'esecuzione del programma. Questo aiuta a rilevare errori di tipo durante la fase di compilazione anziché a runtime, migliorando la robustezza del codice.

```
let numero: number = 42;
let testo: string = "Ciao, mondo!";
```

Orientato agli oggetti

TypeScript supporta la programmazione orientata agli oggetti, consentendo la definizione di classi, interfacce, ereditarietà e altri concetti tipici degli ambienti di sviluppo basati su classi.

```
const oggetto = {
nome: 'Mario',
età: 25,
studente: false
}
```

Il compilatore TS

Il compilatore di TypeScript è un componente essenziale per trasformare il codice sorgente scritto in TypeScript in codice JavaScript eseguibile. Questa fase di compilazione è necessaria perché la maggior parte degli ambienti di runtime, come i browser e Node.js, comprendono solo JavaScript. Ecco perché è necessario utilizzare il compilatore TypeScript per convertire il codice TypeScript in JavaScript.

Installazione del compilatore TypeScript:

Apri il terminale ed esegui il comando:

npm install -g typescript

Type Inference

La **Type Inference** (o inferenza di tipo) è una caratteristica di TypeScript che consente al compilatore di determinare automaticamente il tipo di una variabile in base al contesto in cui viene utilizzata, senza richiedere una dichiarazione esplicita del tipo da parte dello sviluppatore. In altre parole, TypeScript può dedurre il tipo delle variabili in modo implicito.

```
let numero = 42;
// TypeScript deduce che 'numero' è di tipo number

let testo = "Ciao, mondo!";
// TypeScript deduce che 'testo' è di tipo string
```

Special Type

Any

il tipo **any** è un tipo speciale che viene utilizzato per indicare una variabile di cui non si conosce o non si desidera specificare il tipo in fase di compilazione. L'utilizzo del tipo **any** rimuove gran parte dei vantaggi della tipizzazione statica offerti da TypeScript, poiché consente a una variabile di assumere qualsiasi tipo di valore durante l'esecuzione del programma.

```
let valore: any = 42;
valore = "Ciao, mondo!";
valore = true;
```

Nell'esempio sopra, la variabile **valore** è di tipo **any**, il che significa che può contenere valori di qualsiasi tipo.

Union

Il tipo **Union** in TypeScript consente di definire una variabile che può avere uno tra diversi tipi specificati. Si usa il "pipe" (OR) " | " per separare i tipi all'interno di una union. In altre parole, una variabile di tipo union può assumere valori di uno qualsiasi dei tipi elencati.

```
// La variabile 'valore' può essere di tipo number o string
let valore: number | string; valore = 42;
// Valido, poiché 'valore' può essere di tipo number
console.log(valore); // Output: 42


valore = "Hello";
// Valido, poiché 'valore' può essere di tipo string
console.log(valore); // Output: Hello
```

Tuple

Il tipo **Tuple** in TypeScript consente di dichiarare un array di elementi con tipi specifici, dove il tipo di ogni elemento è fissato e noto in anticipo. A differenza di un array normale, dove tutti gli elementi possono avere lo stesso tipo, una tupla consente di specificare tipi diversi per elementi differenti all'interno della stessa struttura di dati.

È importante notare che le tuple hanno una lunghezza e tipo fissi. Se si cerca di assegnare una tupla con un numero di elementi diverso o di tipo diverso, si otterrà un errore in fase di compilazione.

```
// Dichiarazione di una tupla con due elementi, il primo di tipo number e il secondo di tipo string
let coppia: [number, string];

// Assegnazione di valori alla tupla
coppia = [42, "Ciao"];

// Accesso ai valori
console.log(coppia[0]); // Output: 42
console.log(coppia[1]); // Output: Ciao
```

Generics

I **generics** sono una caratteristica potente in TypeScript (e in molti altri linguaggi di programmazione) che consentono di creare componenti (funzioni, classi, interfacce, ecc.) che possono lavorare con una varietà di tipi senza specificare il tipo effettivo al momento della definizione. In altre parole, i **generics** forniscono una flessibilità maggiore, consentendo di scrivere codice più riutilizzabile e generico.

```
// Funzione generica che inverte un array di elementi di qualsiasi tipo
function invertiArray<T>(array:  T[]): T[] {
    return array.reverse();
}
// Uso della funzione generica con un array di numeri
let numeriInvertiti: number[] = invertiArray([1, 2, 3, 4, 5]);
// Uso della funzione generica con un array di stringhe
let stringheInvertite :
string []= invertiArray (["uno", "due", "tre"]);
```

Types VS Interfaces

Interfaces

Le **interfacce** (interfaces) in TypeScript sono uno strumento potente che consente di definire contratti per la struttura dei dati e dei comportamenti all'interno del tuo codice. Le interfacce sono un modo per dichiarare e definire la forma di oggetti, rendendo il codice più leggibile e agevolando la manutenzione.

```
// Dichiarazione di un'interfaccia per rappresentare una persona
interface Persona {
nome: string;
cognome: string;
eta?: number; // Il "?" indica che 'eta' è opzionale
saluta(): void;
}
// Implementazione dell'interfaccia per una persona specifica
let utente: Persona = {
nome: "Mario",
cognome: "Rossi",
eta: 30,
saluta: function () {
console.log(`Ciao, sono ${this.nome} ${this.cognome}.`);
}
};
// Utilizzo del metodo definito nell'interfaccia
utente.saluta(); // Output: Ciao, sono Mario Rossi.
```

Types

Un **type** in TypeScript è un modo per assegnare un nome a un tipo esistente o per creare un nuovo tipo basato su una combinazione di tipi esistenti. Questo consente di rendere il codice più chiaro, più leggibile e più manutenibile.

```
// Definizione di un type alias per una persona
type Persona = {
nome: string;
cognome: string;
eta: number; };

// Utilizzo del type alias per dichiarare una variabile
 let utente: Persona = {
nome: "Mario",
cognome: "Rossi",
eta: 30 };
```

Resoconto

Preferire Interfacce

Se stai definendo una struttura di oggetti che verrà implementata da classi o se devi estendere o combinare interfacce, le interfacce sono spesso una scelta più appropriata.

Preferire Alias di Tipo

Se devi creare tipi basati su unioni, intersezioni, tipi letterali o tipi generici, gli alias di tipo possono essere la scelta migliore.