The Concise TypeScript Book

Simone Poggiali

The Concise TypeScript Book

The Concise TypeScript Book offre una panoramica completa e concisa delle funzionalità di TypeScript. Questo libro offre spiegazioni chiare che coprono tutti gli aspetti dell'ultima versione del linguaggio, dal suo potente sistema di tipi alle funzionalità avanzate. Che siate principianti o sviluppatori esperti, questo libro è una risorsa preziosa per migliorare la vostra comprensione e competenza in TypeScript.

Questo libro è completamente gratuito e open source.

Credo che un'istruzione tecnica di alta qualità debba essere accessibile a tutti, ed è per questo che mantengo questo libro gratuito e aperto.

Se il libro ti ha aiutato a risolvere un bug, a comprendere un concetto ostico o a progredire nella tua carriera, ti prego di considerare di sostenere il mio lavoro pagando quanto vuoi (prezzo suggerito: 15 euro) o sponsorizzando un caffè. Il tuo supporto mi aiuta a mantenere i contenuti aggiornati e ad ampliarli con nuovi esempi e spiegazioni più approfondite.

Traduzioni

Questo libro è stato tradotto in diverse lingue, tra cui:

Cinese

Italiano

Download e sito web

Puoi anche scaricare la versione Epub:

https://github.com/gibbok/typescript-book/tree/main/downloads

È disponibile una versione online su:

https://gibbok.github.io/typescript-book

Indice

Introduzione

Benvenuti a The Concise TypeScript Book! Questa guida vi fornirà le conoscenze essenziali e le competenze pratiche per uno sviluppo TypeScript efficace. Scoprite i concetti e le tecniche chiave per scrivere codice pulito e robusto. Che siate principianti o sviluppatori esperti, questo libro rappresenta sia una guida completa che un pratico riferimento per sfruttare la potenza di TypeScript nei vostri progetti.

Questo libro tratta TypeScript 5.2.

Informazioni sull'autore

Simone Poggiali è uno Staff Engineer esperto, con una passione per la scrittura di codice di livello professionale fin dagli anni '90. Nel corso della sua carriera internazionale, ha contribuito a numerosi progetti per un'ampia gamma di clienti, dalle startup alle grandi organizzazioni. Aziende di spicco come HelloFresh, Siemens, O2, Leroy Merlin e Snowplow hanno beneficiato della sua competenza e dedizione.

È possibile contattare Simone Poggiali sulle seguenti piattaforme:

- LinkedIn: https://www.linkedin.com/in/simone-poggiali
- GitHub: https://github.com/gibbok
- X.com: https://x.com/gibbok_coding

- Email: gibbok.coding gmail.com

Introduzione a TypeScript

Cos’è TypeScript?

TypeScript è un linguaggio di programmazione fortemente tipizzato basato su JavaScript. È stato originariamente progettato da Anders Hejlsberg nel 2012 ed è attualmente sviluppato e gestito da Microsoft come progetto open source.

TypeScript si compila in JavaScript e può essere eseguito in qualsiasi runtime JavaScript (ad esempio, un browser o Node.js su un server).

Supporta diversi paradigmi di programmazione, come la programmazione funzionale, generica, imperativa e orientata agli oggetti, ed è un linguaggio compilato (transpilato) che viene convertito in JavaScript prima dell’esecuzione.

Perché TypeScript?

TypeScript è un linguaggio fortemente tipizzato che aiuta a prevenire errori di programmazione comuni ed evitare determinati tipi di errori di runtime prima dell’esecuzione del programma.

Un linguaggio fortemente tipizzato consente allo sviluppatore di specificare vari vincoli e comportamenti del programma nelle definizioni dei tipi di dati, facilitando la verifica della correttezza del software e la prevenzione dei difetti. Questo è particolarmente utile nelle applicazioni su larga scala.

Alcuni dei vantaggi di TypeScript:

- Tipizzazione statica, facoltativamente fortemente tipizzata
- Inferenza di tipo
- Accesso alle funzionalità di ES6 ed ES7
- Compatibilità multipiattaforma e multibrowser
- Supporto degli strumenti con IntelliSense

TypeScript e JavaScript

TypeScript è scritto in file `.ts` o `.tsx`, mentre i file JavaScript sono scritti in file `.js` o `.jsx`.

I file con estensione `.tsx` o `.jsx` possono contenere l'estensione di sintassi JavaScript JSX, utilizzata in React per lo sviluppo dell'interfaccia utente.

TypeScript è un superset tipizzato di JavaScript (ECMAScript 2015) in termini di sintassi. Tutto il codice JavaScript è codice TypeScript valido, ma il contrario non è sempre vero.

Ad esempio, si consideri una funzione in un file JavaScript con estensione `.js`, come la seguente:

```
const sum = (a, b) => a + b;
```

La funzione può essere convertita e utilizzata in TypeScript modificando l'estensione del file in `.ts`. Tuttavia, se la stessa funzione è annotata con tipi TypeScript, non può essere eseguita in alcun runtime JavaScript senza compilazione. Il seguente codice TypeScript genererà un errore di sintassi se non compilato:

```
const sum = (a: number, b: number): number => a + b;
```

TypeScript è stato progettato per rilevare possibili eccezioni che possono verificarsi in fase di runtime durante la compilazione, consentendo allo sviluppatore di definire l'intento con annotazioni di tipo. Inoltre, TypeScript può anche rilevare problemi se non viene

fornita alcuna annotazione di tipo. Ad esempio, il seguente frammento di codice non specifica alcun tipo TypeScript:

```
const items = [{ x: 1 }, { x: 2 }];
const result = items.filter(item => item.y);
```

In questo caso, TypeScript rileva un errore e segnala:

```
La proprietà 'y' non esiste sul tipo '{ x: number; }'.
```

Il sistema di tipi di TypeScript è ampiamente influenzato dal comportamento runtime di JavaScript. Ad esempio, l'operatore di addizione (+), che in JavaScript può eseguire sia la concatenazione di stringhe che l'addizione numerica, è modellato allo stesso modo in TypeScript:

```
const result = '1' + 1; // Il risultato è di tipo stringa
```

Il team di TypeScript ha deliberatamente deciso di segnalare come errori l'utilizzo insolito di JavaScript. Ad esempio, si consideri il seguente codice JavaScript valido:

```
const result = 1 + true; // In JavaScript, il risultato è uguale a
2
```

Tuttavia, TypeScript genera un errore:

```
L'operatore '+' non può essere applicato ai tipi 'number' e
'boolean'.
```

Questo errore si verifica perché TypeScript applica rigorosamente la compatibilità di tipo e, in questo caso, identifica un'operazione non valida tra un numero e un valore booleano.

Generazione di codice TypeScript

Il compilatore TypeScript ha due responsabilità principali: il controllo degli errori di tipo e la compilazione in JavaScript. Questi

due processi sono indipendenti l'uno dall'altro. I tipi non influenzano l'esecuzione del codice in un runtime JavaScript, poiché vengono completamente cancellati durante la compilazione. TypeScript può comunque generare codice JavaScript anche in presenza di errori di tipo. Ecco un esempio di codice TypeScript con un errore di tipo:

```
const add = (a: number, b: number): number => a + b;
const result = add('x', 'y'); // L'argomento di tipo 'string' non è
assegnabile al parametro di tipo 'number'.
```

Tuttavia, può comunque produrre un output JavaScript eseguibile:

```
'use strict';
const add = (a, b) => a + b;
const result = add('x', 'y'); // xy
```

Non è possibile controllare i tipi TypeScript in fase di esecuzione. Ad esempio:

```
interface Animal {
    name: string;
}
interface Dog extends Animal {
    bark: () => void;
}
interface Cat extends Animal {
    meow: () => void;
}
const makeNoise = (animal: Animal) => {
    if (animal instanceof Dog) {
        // 'Dog' si riferisce solo a un tipo, ma qui viene
utilizzato come valore.
        // ...
    }
};
```

Poiché i tipi vengono cancellati dopo la compilazione, non è possibile eseguire questo codice in JavaScript. Per riconoscere i tipi a runtime, dobbiamo usare un altro meccanismo. TypeScript offre diverse opzioni, una delle quali è la "tagged union". Ad esempio:

```
interface Dog {
    kind: 'dog'; // Tagged union
    bark: () => void;
}
interface Cat {
    kind: 'cat'; // Tagged union
    meow: () => void;
}
type Animal = Dog | Cat;

const makeNoise = (animal: Animal) => {
    if (animal.kind === 'dog') {
        animal.bark();
    } else {
        animal.meow();
    }
};

const dog: Dog = {
    kind: 'dog',
    bark: () => console.log('bark'),
};
makeNoise(dog);
```

La proprietà “kind” è un valore che può essere utilizzato in fase di esecuzione per distinguere gli oggetti in JavaScript.

È anche possibile che un valore in fase di esecuzione abbia un tipo diverso da quello dichiarato nella dichiarazione di tipo. Ad esempio, se lo sviluppatore ha interpretato erroneamente un tipo API e lo ha annotato in modo errato.

TypeScript è un superset di JavaScript, quindi la parola chiave “class” può essere utilizzata come tipo e valore in fase di esecuzione.

```
class Animal {
    constructor(public name: string) {}
}
class Dog extends Animal {
    constructor(
        public name: string,
        public bark: () => void
```

```
    ) {
        super(name);
    }
}
class Cat extends Animal {
    constructor(
        public name: string,
        public meow: () => void
    ) {
        super(name);
    }
}
type Mammal = Dog | Cat;

const makeNoise = (mammal: Mammal) => {
    if (mammal instanceof Dog) {
        mammal.bark();
    } else {
        mammal.meow();
    }
};

const dog = new Dog('Fido', () => console.log('bark'));
makeNoise(dog);
```

In JavaScript, una “classe” ha una proprietà “prototype” e l’operatore “instanceof” può essere utilizzato per verificare se la proprietà prototype di un costruttore appare in qualsiasi punto della catena di prototipi di un oggetto.

TypeScript non ha alcun effetto sulle prestazioni di runtime, poiché tutti i tipi verranno cancellati. Tuttavia, TypeScript introduce un certo overhead in fase di compilazione.

JavaScript moderno ora (Downleveling)

TypeScript può compilare codice per qualsiasi versione rilasciata di JavaScript a partire da ECMAScript 3 (1999). Ciò significa che TypeScript può transpilare codice dalle funzionalità JavaScript più recenti a versioni precedenti, un processo noto come Downleveling.

Questo consente l'utilizzo di JavaScript moderno mantenendo la massima compatibilità con gli ambienti di runtime più vecchi.

È importante notare che durante la transpilazione a una versione precedente di JavaScript, TypeScript potrebbe generare codice che potrebbe comportare un sovraccarico di prestazioni rispetto alle implementazioni native.

Ecco alcune delle funzionalità di JavaScript moderno che possono essere utilizzate in TypeScript:

- Moduli ECMAScript al posto delle callback "define" in stile AMD o delle istruzioni "require" di CommonJS.
- Classi al posto dei prototipi.
- Dichiarazione di variabili utilizzando "let" o "const" al posto di "var".
- Ciclo "for-of" o ".forEach" al posto del tradizionale ciclo "for".
- Funzioni freccia al posto delle espressioni di funzione.
- Assegnazione destrutturata. * Nomi abbreviati di proprietà/metodi e nomi di proprietà calcolate.
- Parametri di funzione predefiniti.

Sfruttando queste moderne funzionalità di JavaScript, gli sviluppatori possono scrivere codice più espressivo e conciso in TypeScript.

Per iniziare con TypeScript

Installazione

Visual Studio Code offre un eccellente supporto per il linguaggio TypeScript, ma non include il compilatore TypeScript. Per installare il compilatore TypeScript, è possibile utilizzare un gestore di pacchetti come npm o yarn:

```
npm install typescript --save-dev
```

oppure

```
yarn add typescript --dev
```

Assicurarsi di eseguire il commit del file di lock generato per garantire che ogni membro del team utilizzi la stessa versione di TypeScript.

Per eseguire il compilatore TypeScript, è possibile utilizzare i seguenti comandi:

```
npx tsc
```

oppure

```
yarn tsc
```

Si consiglia di installare TypeScript a livello di progetto anziché globale, poiché garantisce un processo di build più prevedibile. Tuttavia, per occasioni particolari, è possibile utilizzare il seguente comando:

```
npx tsc
```

oppure installarlo globalmente:

```
npm install -g typescript
```

Se si utilizza Microsoft Visual Studio, è possibile ottenere TypeScript come pacchetto in NuGet per i progetti MSBuild. Nella console di Gestione Pacchetti di NuGet, eseguire il seguente comando:

```
Install-Package Microsoft.TypeScript.MSBuild
```

Durante l'installazione di TypeScript, vengono installati due eseguibili: "tsc" come compilatore TypeScript e "tsserver" come server autonomo TypeScript. Il server autonomo contiene il

compilatore e i servizi linguistici che possono essere utilizzati da editor e IDE per fornire il completamento intelligente del codice.

Inoltre, sono disponibili diversi transpiler compatibili con TypeScript, come Babel (tramite un plugin) o swc. Questi transpiler possono essere utilizzati per convertire il codice TypeScript in altri linguaggi o versioni di destinazione.

Configurazione

TypeScript può essere configurato utilizzando le opzioni della CLI di tsc o un file di configurazione dedicato chiamato tsconfig.json, posizionato nella radice del progetto.

Per generare un file tsconfig.json precompilato con le impostazioni consigliate, è possibile utilizzare il seguente comando:

```
tsc --init
```

Quando si esegue il comando `tsc` localmente, TypeScript compilerà il codice utilizzando la configurazione specificata nel file tsconfig.json più vicino.

Ecco alcuni esempi di comandi CLI che vengono eseguiti con le impostazioni predefinite:

```
tsc main.ts // Compila un file specifico (main.ts) in JavaScript
tsc src/*.ts // Compila tutti i file .ts nella cartella 'src' in
JavaScript
tsc app.ts util.ts --outfile index.js // Compila due file
TypeScript (app.ts e util.ts) in un singolo file JavaScript
(index.js)
```

File di configurazione TypeScript

Un file tsconfig.json viene utilizzato per configurare il compilatore TypeScript (tsc). Solitamente, viene aggiunto alla radice del progetto, insieme al file `package.json`.

Note:

- tsconfig.json accetta commenti anche se è in formato json.
- Si consiglia di utilizzare questo file di configurazione al posto delle opzioni della riga di comando.

Al seguente link potete trovare la documentazione completa e il relativo schema:

https://www.typescriptlang.org/tsconfig

https://www.typescriptlang.org/tsconfig/

Di seguito è riportato un elenco delle configurazioni più comuni e utili:

target

La proprietà “target” viene utilizzata per specificare in quale versione di JavaScript ECMAScript TypeScript deve emettere/compilare. Per i browser moderni, ES6 è una buona opzione, mentre per i browser più vecchi si consiglia ES5.

lib

La proprietà “lib” viene utilizzata per specificare quali file di libreria includere in fase di compilazione. TypeScript include automaticamente le API per le funzionalità specificate nella proprietà “target”, ma è possibile omettere o selezionare librerie specifiche per esigenze particolari. Ad esempio, se si lavora su un

progetto server, è possibile escludere la libreria “DOM”, utile solo in un ambiente browser.

strict

La proprietà “strict” offre garanzie più solide e migliora la sicurezza dei tipi. Si consiglia di includere sempre questa proprietà nel file tsconfig.json del progetto. Abilitando la proprietà “strict”, TypeScript può:

- Emettere codice utilizzando “use strict” per ogni file sorgente.
- Considerare “null” e “undefined” nel processo di controllo dei tipi.
- Disabilitare l’utilizzo del tipo “any” quando non sono presenti annotazioni di tipo.
- Generare un errore sull’utilizzo dell’espressione “this”, che altrimenti implicherebbe il tipo “any”.

module

La proprietà “module” imposta il sistema di moduli supportato dal programma compilato. Durante l’esecuzione, un caricatore di moduli viene utilizzato per individuare ed eseguire le dipendenze in base al sistema di moduli specificato.

I caricatori di moduli più comuni utilizzati in JavaScript sono Node.js CommonJS per le applicazioni lato server e RequireJS per i moduli AMD nelle applicazioni web basate su browser. TypeScript può generare codice per vari sistemi di moduli, tra cui UMD, System, ESNext, ES2015/ES6 ed ES2020.

Nota: il sistema di moduli deve essere scelto in base all’ambiente di destinazione e al meccanismo di caricamento dei moduli disponibile in tale ambiente.

moduleResolution

La proprietà “moduleResolution” specifica la strategia di risoluzione dei moduli. Utilizzare “node” per il codice TypeScript moderno, la strategia “classic” viene utilizzata solo per le vecchie versioni di TypeScript (precedenti alla 1.6).

esModuleInterop

La proprietà “esModuleInterop” consente l’importazione predefinita dai moduli CommonJS che non sono stati esportati utilizzando la proprietà “default”. Questa proprietà fornisce uno shim per garantire la compatibilità nel codice JavaScript emesso. Dopo aver abilitato questa opzione, possiamo usare `import MyLibrary from "my-library"` invece di `import * as MyLibrary from "my-library"`.

jsx

La proprietà “jsx” si applica solo ai file .tsx utilizzati in ReactJS e controlla il modo in cui i costrutti JSX vengono compilati in JavaScript. Un’opzione comune è “preserve”, che compilerà in un file .jsx mantenendo invariato il codice JSX, in modo che possa essere passato a diversi strumenti come Babel per ulteriori trasformazioni.

skipLibCheck

La proprietà “skipLibCheck” impedisce a TypeScript di controllare il tipo di tutti i pacchetti di terze parti importati. Questa proprietà riduce il tempo di compilazione di un progetto. TypeScript controllerà comunque il codice rispetto alle definizioni di tipo fornite da questi pacchetti.

files

La proprietà “files” indica al compilatore un elenco di file che devono essere sempre inclusi nel programma.

include

La proprietà “include” indica al compilatore un elenco di file che si desidera includere. Questa proprietà consente schemi di tipo glob, come “*” *per qualsiasi sottodirectory,* “” per qualsiasi nome di file e “?” per caratteri opzionali.

exclude

La proprietà “exclude” indica al compilatore un elenco di file che non devono essere inclusi nella compilazione. Questo può includere file come “node_modules” o file di test. Nota: tsconfig.json consente commenti.

importHelpers

TypeScript utilizza codice helper durante la generazione di codice per determinate funzionalità JavaScript avanzate o di livello inferiore. Per impostazione predefinita, questi helper vengono duplicati nei file che li utilizzano. L’opzione `importHelpers` importa invece questi helper dal modulo `tslib`, rendendo l’output JavaScript più efficiente.

Consigli per la migrazione a TypeScript

Per progetti di grandi dimensioni, si consiglia di adottare una transizione graduale in cui TypeScript e codice JavaScript coesisteranno inizialmente. Solo i progetti di piccole dimensioni possono essere migrati a TypeScript in un’unica soluzione.

Il primo passo di questa transizione è introdurre TypeScript nel processo di build chain. Questo può essere fatto utilizzando l'opzione del compilatore "allowJs", che consente ai file .ts e .tsx di coesistere con i file JavaScript esistenti. Poiché TypeScript tornerà al tipo "any" per una variabile quando non riesce a dedurre il tipo dai file JavaScript, si consiglia di disabilitare "noImplicitAny" nelle opzioni del compilatore all'inizio della migrazione.

Il secondo passaggio consiste nell'assicurarsi che i test JavaScript funzionino insieme ai file TypeScript, in modo da poterli eseguire durante la conversione di ciascun modulo. Se si utilizza Jest, si può valutare l'utilizzo di `ts-jest`, che consente di testare i progetti TypeScript con Jest.

Il terzo passaggio consiste nell'includere le dichiarazioni di tipo per le librerie di terze parti nel progetto. Queste dichiarazioni sono disponibili in bundle o su DefinitelyTyped. È possibile cercarle utilizzando https://www.typescriptlang.org/dt/search e installarle tramite:

```
npm install --save-dev @types/package-name
```

or

```
yarn add --dev @types/package-name
```

Il quarto passaggio consiste nel migrare modulo per modulo con un approccio bottom-up, seguendo il grafo delle dipendenze partendo dalle foglie. L'idea è di iniziare a convertire i moduli che non dipendono da altri moduli. Per visualizzare i grafici delle dipendenze, è possibile utilizzare lo strumento "madge".

I moduli candidati ideali per queste conversioni iniziali sono funzioni di utilità e codice relativo ad API o specifiche esterne. È possibile generare automaticamente definizioni di tipo TypeScript da contratti Swagger, schemi GraphQL o JSON da includere nel progetto.

Quando non sono disponibili specifiche o schemi ufficiali, è possibile generare tipi da dati grezzi, come JSON restituiti da un server. Tuttavia, si consiglia di generare tipi da specifiche anziché da dati per evitare di perdere casi limite.

Durante la migrazione, evitare il refactoring del codice e concentrarsi solo sull'aggiunta di tipi ai moduli.

Il quinto passaggio consiste nell'abilitare "noImplicitAny", che garantirà che tutti i tipi siano noti e definiti, offrendo una migliore esperienza TypeScript per il progetto.

Durante la migrazione, è possibile utilizzare la direttiva `@ts-check`, che abilita il controllo dei tipi TypeScript in un file JavaScript. Questa direttiva fornisce una versione semplificata del controllo dei tipi e può essere utilizzata inizialmente per identificare problemi nei file JavaScript. Quando `@ts-check` è incluso in un file, TypeScript tenterà di dedurre le definizioni utilizzando commenti in stile JSDoc. Tuttavia, si consiglia di utilizzare le annotazioni JSDoc solo in una fase molto precoce della migrazione.

Si consiglia di mantenere il valore predefinito di `noEmitOnError` nel file tsconfig.json su false. Questo consentirà di generare codice sorgente JavaScript anche se vengono segnalati errori.

Esplorazione del sistema di tipi

Il servizio di linguaggio TypeScript

Il servizio di linguaggio TypeScript, noto anche come tsserver, offre diverse funzionalità come la segnalazione degli errori, la diagnostica, la compilazione al salvataggio, la ridenominazione, il passaggio alla definizione, gli elenchi di completamento, la guida alle firme e altro ancora. Viene utilizzato principalmente dagli ambienti di sviluppo

integrati (IDE) per fornire supporto IntelliSense. Si integra perfettamente con Visual Studio Code ed è utilizzato da strumenti come Conquer of Completion (Coc).

Gli sviluppatori possono sfruttare un'API dedicata e creare plugin di servizi linguistici personalizzati per migliorare l'esperienza di modifica di TypeScript. Questo può essere particolarmente utile per implementare funzionalità di linting speciali o abilitare il completamento automatico per un linguaggio di template personalizzato.

Un esempio di plugin personalizzato reale è "TypeScript-styled-plugin", che fornisce la segnalazione degli errori di sintassi e il supporto IntelliSense per le proprietà CSS nei componenti con stile.

Per ulteriori informazioni e guide rapide, è possibile consultare il Wiki ufficiale di TypeScript su GitHub:
https://github.com/microsoft/TypeScript/wiki/

Tipizzazione Strutturale

TypeScript si basa su un sistema di tipi strutturale. Ciò significa che la compatibilità e l'equivalenza dei tipi sono determinate dalla struttura o definizione effettiva del tipo, piuttosto che dal suo nome o dal punto di dichiarazione, come nei sistemi di tipi nominativi come C# o C.

Il sistema di tipi strutturale di TypeScript è stato progettato sulla base del funzionamento del sistema di tipizzazione dinamica di JavaScript durante l'esecuzione.

L'esempio seguente è codice TypeScript valido. Come si può osservare, "X" e "Y" hanno lo stesso membro "a", anche se hanno nomi di dichiarazione diversi. I tipi sono determinati dalle loro strutture e, in questo caso, poiché le strutture sono le stesse, sono compatibili e validi.

```
type X = {
    a: string;
};
type Y = {
    a: string;
};
const x: X = { a: 'a' };
const y: Y = x; // Valido
```

Regole fondamentali di confronto di TypeScript

Il processo di confronto di TypeScript è ricorsivo ed è eseguito su tipi annidati a qualsiasi livello.

Un tipo “X” è compatibile con “Y” se “Y” ha almeno gli stessi membri di “X”.

```
type X = {
    a: string;
};
const y = { a: 'A', b: 'B' }; // Valido, poiché ha almeno gli
stessi membri di X
const r: X = y;
```

I parametri delle funzioni vengono confrontati in base al tipo, non al nome:

```
type X = (a: number) => void;
type Y = (a: number) => void;
let x: X = (j: number) => undefined;
let y: Y = (k: number) => undefined;
y = x; // Valido
x = y; // Valido
```

I tipi restituiti dalla funzione devono essere gli stessi:

```
type X = (a: number) => undefined;
type Y = (a: number) => number;
let x: X = (a: number) => undefined;
```

```
let y: Y = (a: number) => 1;
y = x; // Non valido
x = y; // Non valido
```

Il tipo di ritorno di una funzione sorgente deve essere un sottotipo del tipo di ritorno di una funzione target:

```
let x = () => ({ a: 'A' });
let y = () => ({ a: 'A', b: 'B' });
x = y; // Valido
y = x; // Il membro non valido b è mancante
```

È consentito ignorare i parametri della funzione, come è prassi comune in JavaScript, ad esempio utilizzando "Array.prototype.map()":

```
[1, 2, 3].map((element, _index, _array) => element + 'x');
```

Pertanto, le seguenti dichiarazioni di tipo sono completamente valide:

```
type X = (a: number) => undefined;
type Y = (a: number, b: number) => undefined;
let x: X = (a: number) => undefined;

let y: Y = (a: number) => undefined; // Parametro b mancante
y = x; // Valido
```

Tutti i parametri opzionali aggiuntivi del tipo sorgente sono validi:

```
type X = (a: number, b?: number, c?: number) => undefined;
type Y = (a: number) => undefined;
let x: X = a => undefined;
let y: Y = a => undefined;
y = x; // Valido
x = y; //Valido
```

Tutti i parametri opzionali del tipo destinazione senza parametri corrispondenti nel tipo sorgente sono validi e non costituiscono un errore:

```
type X = (a: number) => undefined;
type Y = (a: number, b?: number) => undefined;
let x: X = a => undefined;
let y: Y = a => undefined;
y = x; // Valido
x = y; // Valido
```

Il parametro rest viene trattato come una serie infinita di parametri opzionali:

```
type X = (a: number, ...rest: number[]) => undefined;
let x: X = a => undefined; //valido
```

Le funzioni con overload sono valide se la firma di overload è compatibile con la firma della sua implementazione:

```
function x(a: string): void;
function x(a: string, b: number): void;
function x(a: string, b?: number): void {
    console.log(a, b);
}
x('a'); // Valido
x('a', 1); // Valido

function y(a: string): void; // Non valido, non compatibile con la
firma dell'implementazione
function y(a: string, b: number): void;
function y(a: string, b: number): void {
    console.log(a, b);
}
y('a');
y('a', 1);
```

Il confronto dei parametri della funzione ha esito positivo se i parametri sorgente e destinazione sono assegnabili a supertipi o sottotipi (bivarianza).

```
// Supertipo
class X {
    a: string;
    constructor(value: string) {
        this.a = value;
```

```
    }
}
// Sottotipo
class Y extends X {}
// Sottotipo
class Z extends X {}

type GetA = (x: X) => string;
const getA: GetA = x => x.a;

// La bivarianza accetta supertipi
console.log(getA(new X('x'))); // Valido
console.log(getA(new Y('Y'))); // Valido
console.log(getA(new Z('z'))); // Valido
```

Gli enum sono confrontabili e validi con i numeri e viceversa, ma il confronto di valori Enum di tipi Enum diversi non è valido.

```
enum X {
    A,
    B,
}
enum Y {
    A,
    B,
    C,
}
const xa: number = X.A; // Valido
const ya: Y = 0; // Valido
X.A === Y.A; // Non valido
```

Le istanze di una classe sono soggette a un controllo di compatibilità per i loro membri privati e protetti:

```
class X {
    public a: string;
    constructor(value: string) {
        this.a = value;
    }
}

class Y {
    private a: string;
```

```
    constructor(value: string) {
        this.a = value;
    }
}

let x: X = new Y('y'); // Non valido
```

Il controllo di confronto non tiene conto della diversa gerarchia di ereditarietà, ad esempio:

```
class X {
    public a: string;
    constructor(value: string) {
        this.a = value;
    }
}
class Y extends X {
    public a: string;
    constructor(value: string) {
        super(value);
        this.a = value;
    }
}
class Z {
    public a: string;
    constructor(value: string) {
        this.a = value;
    }
}
let x: X = new X('x');
let y: Y = new Y('y');
let z: Z = new Z('z');
x === y; // Valido
x === z; // Valido anche se z proviene da una gerarchia di
ereditarietà diversa
```

I generici vengono confrontati utilizzando le loro strutture in base al tipo risultante dopo l'applicazione del parametro generico; solo il risultato finale viene confrontato come tipo non generico.

```
interface X<T> {
    a: T;
}
```

```
let x: X<number> = { a: 1 };
let y: X<string> = { a: 'a' };
x === y; // Non valido poiché l'argomento tipo è utilizzato nella
struttura finale

interface X<T> {}
const x: X<number> = 1;
const y: X<string> = 'a';
x === y; // Valido poiché l'argomento tipo non è utilizzato nella
struttura finale
```

Quando i generici non hanno il loro argomento tipo specificato, tutti gli argomenti non specificati vengono trattati come tipi con “any”:

```
type X = <T>(x: T) => T;
type Y = <K>(y: K) => K;
let x: X = x => x;
let y: Y = y => y;
x = y; // Valido
```

Ricorda:

```
let a: number = 1;
let b: number = 2;
a = b; // Valido, tutto è assegnabile a se stesso

let c: any;
c = 1; // Valido, tutti i tipi sono assegnabili a qualsiasi

let d: unknown;
d = 1; // Valido, tutti i tipi sono assegnabili a sconosciuto

let e: unknown;
let e1: unknown = e; // Valido, sconosciuto è assegnabile solo a se
stesso e a qualsiasi
let e2: any = e; // Valido
let e3: number = e; // Non valido

let f: never;
f = 1; // Non valido, nulla è assegnabile a never

let g: void;
let g1: any;
```

```
g = 1; // Non valido, void non è assegnabile a o da nulla, tranne
qualsiasi
g = g1; // Valido
```

Si noti che quando “strictNullChecks” è abilitato, “null” e “undefined” vengono trattati in modo simile a “void”; in caso contrario, sono simili a “never”.

Tipi come insiemi

In TypeScript, un tipo è un insieme di possibili valori. Questo insieme è anche definito dominio del tipo. Ogni valore di un tipo può essere visto come un elemento di un insieme. Un tipo stabilisce i vincoli che ogni elemento dell’insieme deve soddisfare per essere considerato membro di quell’insieme. Il compito principale di TypeScript è controllare e verificare se un insieme è un sottoinsieme di un altro.

TypeScript supporta vari tipi di insiemi:

Termine di insieme	TypeScript	Note
Insieme vuoto	never	“never” non contiene nulla
Insieme di un singolo elemento	undefined / null / tipo letterale	
Insieme finito	boolean / union	
Insieme infinito	string / number / object	

Termine di insieme	TypeScript	Note
Insieme universale	any / unknown	Ogni elemento è un membro di “any” e ogni insieme è un suo sottoinsieme / “unknown” è una controparte di tipo sicuro di “any”

Ecco alcuni esempi:

TypeScript	Termine di insieme	Esempio
never	∅ (insieme vuoto)	const x: never = ‘x’; // Errore: il tipo ‘string’ non è assegnabile al tipo ‘never’
Tipo letterale	Insieme di elementi singoli	type X = ‘X’;

	type Y = 7;
Valore assegnabile a T	Valore ∈ T (membro di)	type XY = ‘X’	‘Y’;
	const x: XY = ‘X’;
T1 assegnabile a T2	T1 ⊆ T2 (sottoinsieme di)	type XY = ‘X’	‘Y’;
	const x: XY = ‘X’;
	const j: XY = ‘J’; // Il tipo “‘J”’ non è assegnabile al tipo ‘XY’.
T1 extends T2	T1 ⊆ T2 (sottoinsieme di)	type X = ‘X’ extends string ? true : false;
T1	T2	T1 ∪ T2 (unione)	type XY = ‘X’	‘Y’;

| | type JK = 1 | 2; |
| |
T1 & T2 | T1 ∩ T2 (intersezione) | type X = { a: string } |
	type Y = { b: string }
	type XY = X & Y
	const x: XY = { a: ‘a’, b: ‘b’ }
unknown | Insieme universale | const x: unknown = 1 |

Un’unione (T1 | T2) crea un insieme più ampio (entrambi):

```
type X = {
    a: string;
};
type Y = {
    b: string;
};
type XY = X | Y;
const r: XY = { a: 'a', b: 'x' }; // Valido
```

Un’intersezione (T1 e T2) crea un insieme più ristretto (solo condiviso):

```
type X = {
    a: string;
};
type Y = {
    a: string;
    b: string;
};
type XY = X & Y;
const r: XY = { a: 'a' }; // Non valido
const j: XY = { a: 'a', b: 'b' }; // Valido
```

La parola chiave `extends` potrebbe essere considerata un “sottoinsieme di” in questo contesto. Imposta un vincolo per un tipo. L’extends utilizzato con un generico, considera il generico come un insieme infinito e lo vincola a un tipo più specifico. Si noti che `extends non ha nulla a che fare con la gerarchia in senso OOP (questo concetto non esiste in TypeScript). TypeScript funziona con

insiemi e non ha una gerarchia rigida; infatti, come nell'esempio seguente, due tipi potrebbero sovrapporsi senza che uno dei due sia un sottotipo dell'altro (TypeScript considera la struttura e la forma degli oggetti).

```
interface X {
    a: string;
}
interface Y extends X {
    b: string;
}
interface Z extends Y {
    c: string;
}
const z: Z = { a: 'a', b: 'b', c: 'c' };
interface X1 {
    a: string;
}

interface Y1 {
    a: string;
    b: string;
}
interface Z1 {
    a: string;
    b: string;
    c: string;
}
const z1: Z1 = { a: 'a', b: 'b', c: 'c' };

const r: Z1 = z; // Valido
```

Assegnare un tipo: Dichiarazioni di tipo e asserzioni di tipo

Un tipo può essere assegnato in diversi modi in TypeScript:

Dichiarazione di tipo

Nell'esempio seguente, utilizziamo x: X (": Type") per dichiarare un tipo per la variabile x.

```
type X = {
    a: string;
};

// Dichiarazione di tipo
const x: X = {
    a: 'a',
};
```

Se la variabile non è nel formato specificato, TypeScript segnalerà un errore. Ad esempio:

```
type X = {
    a: string;
};

const x: X = {
    a: 'a',
    b: 'b', // Errore: il letterale dell'oggetto può specificare
solo proprietà note
};
```

Asserzione di tipo

È possibile aggiungere un'asserzione utilizzando la parola chiave `as`. Questo indica al compilatore che lo sviluppatore ha maggiori informazioni su un tipo e silenzia eventuali errori.

Ad esempio:

```
type X = {
    a: string;
};
const x = {
    a: 'a',
```

```
    b: 'b',
} as X;
```

Nell'esempio precedente, si asserisce che l'oggetto x abbia il tipo X utilizzando la parola chiave as. Questo informa il compilatore TypeScript che l'oggetto è conforme al tipo specificato, anche se ha una proprietà aggiuntiva b non presente nella definizione del tipo.

Le asserzioni di tipo sono utili in situazioni in cui è necessario specificare un tipo più specifico, soprattutto quando si lavora con il DOM. Ad esempio:

```
const myInput = document.getElementById('my_input') as
HTMLInputElement;
```

Qui, l'asserzione di tipo come HTMLInputElement viene utilizzata per indicare a TypeScript che il risultato di getElementById deve essere trattato come un HTMLInputElement. Le asserzioni di tipo possono anche essere utilizzate per rimappare le chiavi, come mostrato nell'esempio seguente con letterali di template:

```
type J<Type> = {
    [Property in keyof Type as `prefix_${string &
        Property}`]: () => Type[Property];
};
type X = {
    a: string;
    b: number;
};
type Y = J<X>;
```

In questo esempio, il tipo `J<Tipo>` utilizza un tipo mappato con un letterale template per rimappare le chiavi di Tipo. Crea nuove proprietà con un "prefisso_" aggiunto a ciascuna chiave e i valori corrispondenti sono funzioni che restituiscono i valori delle proprietà originali.

È importante notare che quando si utilizza un'asserzione di tipo, TypeScript non eseguirà controlli di proprietà eccessivi. Pertanto, è

generalmente preferibile utilizzare una Dichiarazione di Tipo quando la struttura dell'oggetto è nota in anticipo.

Dichiarazioni Ambientali

Le dichiarazioni Ambientali sono file che descrivono i tipi per il codice JavaScript e hanno un formato di nome file come `.d.ts.`. Di solito vengono importate e utilizzate per annotare librerie JavaScript esistenti o per aggiungere tipi a file JS esistenti nel progetto.

Molti tipi di librerie comuni sono disponibili all'indirizzo: https://github.com/DefinitelyTyped/DefinitelyTyped/

e possono essere installate tramite:

```
npm install --save-dev @types/library-name
```

Per le dichiarazioni di ambiente definite, è possibile importarle utilizzando il riferimento "tripla barra":

```
/// <reference path="./library-types.d.ts" />
```

È possibile utilizzare le dichiarazioni di ambiente anche all'interno di file JavaScript utilizzando `// @ts-check`.

La parola chiave `declare` abilita le definizioni di tipo per il codice JavaScript esistente senza importarlo, fungendo da segnaposto per i tipi da un altro file o a livello globale.

Controllo delle proprietà e controllo delle proprietà in eccesso

TypeScript si basa su un sistema di tipi strutturale, ma il controllo delle proprietà in eccesso è una proprietà di TypeScript che gli consente di verificare se un oggetto possiede esattamente le proprietà specificate nel tipo.

Il controllo delle proprietà in eccesso viene eseguito, ad esempio, quando si assegnano letterali di oggetto a variabili o quando li si passa come argomenti alla proprietà in eccesso di una funzione.

```
type X = {
    a: string;
};
const y = { a: 'a', b: 'b' };
const x: X = y; // Valido perché tipizzazione strutturale
const w: X = { a: 'a', b: 'b' }; // Non valido perché controllo
delle proprietà in eccesso
```

Tipi deboli

Un tipo è considerato debole quando contiene solo un insieme di proprietà completamente opzionali:

```
type X = {
    a?: string;
    b?: string;
};
```

TypeScript considera un errore assegnare qualsiasi cosa a un tipo debole quando non c'è sovrapposizione, ad esempio, il seguente codice genera un errore:

```
type Options = {
    a?: string;
    b?: string;
};

const fn = (options: Options) => undefined;

fn({ c: 'c' }); // Non valido
```

Sebbene non sia consigliato, se necessario, è possibile bypassare questo controllo utilizzando l'asserzione di tipo:

```
type Options = {
    a?: string;
```

```
    b?: string;
};
const fn = (options: Options) => undefined;
fn({ c: 'c' } as Options); // Valido
```

Oppure aggiungendo `unknown` alla firma dell'indice del tipo debole:

```
type Options = {
    [prop: string]: unknown;
    a?: string;
    b?: string;
};

const fn = (options: Options) => undefined;
fn({ c: 'c' }); // Valido
```

Controllo rigoroso dei letterali di oggetto (freschezza)

Il controllo rigoroso dei letterali di oggetto, a volte chiamato "freschezza", è una funzionalità di TypeScript che aiuta a individuare proprietà in eccesso o con errori di ortografia che altrimenti passerebbero inosservate nei normali controlli di tipo strutturale.

Quando si crea un letterale di oggetto, il compilatore TypeScript lo considera "fresco". Se il letterale di oggetto viene assegnato a una variabile o passato come parametro, TypeScript genererà un errore se il letterale di oggetto specifica proprietà che non esistono nel tipo di destinazione.

Tuttavia, la "freschezza" scompare quando un letterale di oggetto viene ampliato o viene utilizzata un'asserzione di tipo.

Ecco alcuni esempi per illustrare:

```
type X = { a: string };
type Y = { a: string; b: string };
```

```
let x: X;
x = { a: 'a', b: 'b' }; // Controllo di freschezza: Assegnazione
non valida
var y: Y;
y = { a: 'a', bx: 'bx' }; // Controllo di freschezza: Assegnazione
non valida

const fn = (x: X) => console.log(x.a);

fn(x);
fn(y); // Allargamento: Nessun errore, strutturalmente compatibile
con il tipo

fn({ a: 'a', bx: 'b' }); // Controllo di aggiornamento: argomento
non valido

let c: X = { a: 'a' };
let d: Y = { a: 'a', b: '' };
c = d; // Allargamento: nessun controllo di aggiornamento
```

Inferenza di tipo

TypeScript può inferire i tipi quando non viene fornita alcuna annotazione durante:

- Inizializzazione delle variabili.
- Inizializzazione dei membri.
- Impostazione dei valori predefiniti per i parametri.
- Tipo di ritorno della funzione.

Ad esempio:

```
let x = 'x'; // Il tipo inferito è una stringa
```

Il compilatore TypeScript analizza il valore o l'espressione e ne determina il tipo in base alle informazioni disponibili.

Inferenze più avanzate

Quando si utilizzano più espressioni nell'inferenza di tipo, TypeScript cerca i "tipi più comuni". Ad esempio:

```
let x = [1, 'x', 1, null]; // Il tipo dedotto è: (string | number | null)[]
```

Se il compilatore non riesce a trovare i tipi comuni migliori, restituisce un tipo unione. Ad esempio:

```
let x = [new RegExp('x'), new Date()]; // Il tipo inferito è: (RegExp | Date)[]
```

TypeScript utilizza la "tipizzazione contestuale" basata sulla posizione della variabile per inferire i tipi. Nell'esempio seguente, il compilatore sa che `e` è di tipo `MouseEvent` grazie al tipo di evento `click` definito nel file lib.d.ts, che contiene dichiarazioni ambientali per vari costrutti JavaScript comuni e il DOM:

```
window.addEventListener('click', function (e) {}); // Il tipo inferito di e è MouseEvent
```

Allargamento di tipo

L'allargamento di tipo è il processo in cui TypeScript assegna un tipo a una variabile inizializzata quando non è stata fornita alcuna annotazione di tipo. Consente il passaggio da tipi stretti a più ampi, ma non viceversa. Nell'esempio seguente:

```
let x = 'x'; // TypeScript inferisce come stringa, un tipo ampio
let y: 'y' | 'x' = 'y'; // Il tipo y è un'unione di tipi letterali
y = x; // Il tipo non valido 'string' non è assegnabile al tipo '"x" | "y"'.
```

TypeScript assegna `string` a `x` in base al singolo valore fornito durante l'inizializzazione (`x`); questo è un esempio di ampliamento.

TypeScript fornisce modi per controllare il processo di ampliamento, ad esempio utilizzando “const”.

Const

L’utilizzo della parola chiave `const` durante la dichiarazione di una variabile produce un’inferenza di tipo più ristretta in TypeScript.

Ad esempio:

```
const x = 'x'; // TypeScript deduce il tipo di x come 'x', un tipo più ristretto
let y: 'y' | 'x' = 'y';
y = x; // Valido: il tipo di x viene dedotto come 'x'
```

Utilizzando `const` per dichiarare la variabile x, il suo tipo viene ristretto allo specifico valore letterale ‘x’. Poiché il tipo di x viene ristretto, può essere assegnato alla variabile y senza errori. Il motivo per cui il tipo può essere dedotto è che le variabili `const` non possono essere riassegnate, quindi il loro tipo può essere ristretto a un tipo letterale specifico, in questo caso, il tipo letterale ‘x’.

Modificatore Const sui parametri di tipo

Dalla versione 5.0 di TypeScript, è possibile specificare l’attributo `const` su un parametro di tipo generico. Questo consente di dedurre il tipo più preciso possibile. Vediamo un esempio senza usare `const`:

```
function identity<T>(value: T) {
    // Nessuna costante qui
    return value;
}
const values = identity({ a: 'a', b: 'b' }); // Il tipo inferito è: { a: string; b: string; }
```

Come puoi vedere, le proprietà `a` e `b` vengono inferite con un tipo `string`.

Ora, vediamo la differenza con la versione `const`:

```
function identity<const T>(value: T) {
    // Utilizzo del modificatore const sui parametri di tipo
    return value;
}
const values = identity({ a: 'a', b: 'b' }); // Il tipo inferito è:
{ a: "a"; b: "b"; }
```

Ora possiamo vedere che le proprietà `a` e `b` vengono dedotte come `const`, quindi `a` e `b` vengono trattate come stringhe letterali anziché come semplici tipi `string`.

Asserzione Const

Questa funzionalità consente di dichiarare una variabile con un tipo letterale più preciso in base al suo valore di inizializzazione, indicando al compilatore che il valore deve essere trattato come un letterale immutabile. Ecco alcuni esempi:

Su una singola proprietà:

```
const v = {
    x: 3 as const,
};
v.x = 3;
```

Su un intero oggetto:

```
const v = {
    x: 1,
    y: 2,
} as const;
```

Questo può essere particolarmente utile quando si definisce il tipo per una tupla:

```
const x = [1, 2, 3]; // number[]
const y = [1, 2, 3] as const; // Tupla di readonly [1, 2, 3]
```

Annotazione di tipo esplicita

Possiamo essere specifici e passare un tipo, nell'esempio seguente la proprietà `x` è di tipo `number`:

```
const v = {
    x: 1, // Inferred type: number (widening)
};
v.x = 3; // Valido
```

Possiamo rendere l'annotazione di tipo più specifica utilizzando un'unione di tipi letterali:

```
const v: { x: 1 | 2 | 3 } = {
    x: 1, // x è ora un'unione di tipi letterali: 1 | 2 | 3
};
v.x = 3; // Valido
v.x = 100; // Non valido
```

Restringimento dei tipi

Il restringimento dei tipi è il processo in TypeScript in cui un tipo generico viene ridotto a un tipo più specifico. Ciò si verifica quando TypeScript analizza il codice e determina che determinate condizioni o operazioni possono perfezionare le informazioni sul tipo.

Il restringimento dei tipi può avvenire in diversi modi, tra cui:

Condizioni

Utilizzando istruzioni condizionali, come `if` o `switch`, TypeScript può restringere il tipo in base al risultato della condizione. Ad esempio:

```
let x: number | undefined = 10;

if (x !== undefined) {
```

```
    x += 100; // Il tipo è number, che è stato ristretto dalla
condizione
}
```

Generazione o restituzione

Generare un errore o restituire un'istruzione in anticipo da un branch può essere utilizzato per aiutare TypeScript a restringere un tipo. Ad esempio:

```
let x: number | undefined = 10;

if (x === undefined) {
    throw 'error';
}
x += 100;
```

Altri modi per restringere i tipi in TypeScript includono:

- Operatore `instanceof`: utilizzato per verificare se un oggetto è un'istanza di una classe specifica.
- Operatore `in`: utilizzato per verificare se una proprietà esiste in un oggetto.
- Operatore `typeof`: utilizzato per verificare il tipo di un valore in fase di esecuzione.
- Funzioni integrate come `Array.isArray()`: utilizzate per verificare se un valore è un array.

Unione Discriminata

L'utilizzo di una "Unione Discriminata" è un pattern in TypeScript in cui un "tag" esplicito viene aggiunto agli oggetti per distinguere i diversi tipi all'interno di un'unione. Questo pattern è anche definito "unione con tag". Nell'esempio seguente, il "tag" è rappresentato dalla proprietà "type":

```
type A = { type: 'type_a'; value: number };
type B = { type: 'type_b'; value: string };
```

```
const x = (input: A | B): string | number => {
    switch (input.type) {
        case 'type_a':
            return input.value + 100; // il tipo è A
        case 'type_b':
            return input.value + 'extra'; // il tipo è B
    }
};
```

Protezioni di tipo definite dall'utente

Nei casi in cui TypeScript non sia in grado di determinare un tipo, è possibile scrivere una funzione di supporto nota come "protezione di tipo definita dall'utente". Nell'esempio seguente, utilizzeremo un predicato di tipo per restringere il tipo dopo aver applicato un determinato filtro:

```
const data = ['a', null, 'c', 'd', null, 'f'];

const r1 = data.filter(x => x != null); // Il tipo è (string |
null)[], TypeScript non è riuscito a dedurre correttamente il tipo

const isValid = (item: string | null): item is string => item !==
null; // Protezione personalizzata del tipo

const r2 = data.filter(isValid); // Il tipo ora è corretto
string[], utilizzando la protezione del tipo predicato siamo
riusciti a restringere il tipo
```

Tipi primitivi

TypeScript supporta 7 tipi primitivi. Un tipo di dati primitivo si riferisce a un tipo che non è un oggetto e non ha metodi associati. In TypeScript, tutti i tipi primitivi sono immutabili, il che significa che i loro valori non possono essere modificati una volta assegnati.

string

Il tipo primitivo `string` memorizza dati testuali e il valore è sempre racchiuso tra virgolette doppie o singole.

```
const x: string = 'x';
const y: string = 'y';
```

Le stringhe possono estendersi su più righe se racchiuse dal carattere di apice inverso (`):

```
let sentence: string = `xxx,
yyy`;
```

boolean

Il tipo di dati `boolean` in TypeScript memorizza un valore binario, `true` o `false`.

```
const isReady: boolean = true;
```

number

Un tipo di dati `number` in TypeScript è rappresentato da un valore in virgola mobile a 64 bit. Un tipo di dati `number` può rappresentare numeri interi e frazioni. TypeScript supporta anche i sistemi di numerazione esadecimale, binario e ottale, ad esempio:

```
const decimal: number = 10;
const hexadecimal: number = 0xa00d; // L'esadecimale inizia con 0x
const binary: number = 0b1010; // Il binario inizia con 0b
const octal: number = 0o633; // L'ottale inizia con 0o
```

bigInt

Un `bigInt` rappresenta valori numerici molto grandi (253 – 1) e non possono essere rappresentati con un `number`.

Un `bigInt` può essere creato chiamando la funzione integrata `BigInt()` o aggiungendo `n` alla fine di qualsiasi letterale numerico intero:

```
const x: bigint = BigInt(9007199254740991);
const y: bigint = 9007199254740991n;
```

Note:

- I valori `bigInt` non possono essere combinati con `number` e non possono essere utilizzati con `Math` integrato, devono essere forzati allo stesso tipo.
- I valori `bigInt` sono disponibili solo se la configurazione di destinazione è ES2020 o superiore.

Simbolo

I simboli sono identificatori univoci che possono essere utilizzati come chiavi di proprietà negli oggetti per evitare conflitti di denominazione.

```
type Obj = {
    [sym: symbol]: number;
};

const a = Symbol('a');
const b = Symbol('b');
let obj: Obj = {};
obj[a] = 123;
obj[b] = 456;

console.log(obj[a]); // 123
console.log(obj[b]); // 456
```

null e undefined

I tipi `null` e `undefined` rappresentano entrambi nessun valore o l'assenza di qualsiasi valore.

Il tipo `undefined` indica che il valore non è assegnato o inizializzato o indica un'assenza involontaria di valore.

Il tipo `null` indica che sappiamo che il campo non ha un valore, quindi il valore non è disponibile, e indica un'assenza intenzionale di valore.

Array

Un `array` è un tipo di dati che può memorizzare più valori dello stesso tipo o meno. Può essere definito utilizzando la seguente sintassi:

```
const x: string[] = ['a', 'b'];
const y: Array<string> = ['a', 'b'];
const j: Array<string | number> = ['a', 1, 'b', 2]; // Unione
```

TypeScript supporta array di sola lettura utilizzando la seguente sintassi:

```
const x: readonly string[] = ['a', 'b']; // Modificatore di sola
lettura
const y: ReadonlyArray<string> = ['a', 'b'];
const j: ReadonlyArray<string | number> = ['a', 1, 'b', 2];
j.push('x'); // Non valido
```

TypeScript supporta tuple e tuple di sola lettura:

```
const x: [string, number] = ['a', 1];
const y: readonly [string, number] = ['a', 1];
```

any

Il tipo di dati `any rappresenta letteralmente un valore "qualsiasi", ed è il valore predefinito quando TypeScript non può dedurre il tipo o non è specificato.

Quando si utilizza `any`, il compilatore TypeScript salta il controllo del tipo, quindi non c'è sicurezza di tipo quando si utilizza `any`. In genere, non utilizzare `any` per silenziare il compilatore quando si verifica un errore, ma concentrarsi sulla correzione dell'errore, poiché utilizzando `any` è possibile interrompere i contratti e perdere i vantaggi del completamento automatico di TypeScript.

Il tipo `any` potrebbe essere utile durante una migrazione graduale da JavaScript a TypeScript, in quanto può silenziare il compilatore.

Per i nuovi progetti, utilizzare la configurazione TypeScript `noImplicitAny`, che consente a TypeScript di generare errori quando viene utilizzato o dedotto `any`.

Il tipo `any` è solitamente fonte di errori che possono mascherare problemi reali con i tipi. Evitatelo il più possibile.

Annotazioni di tipo

Sulle variabili dichiarate usando `var`, `let` e `const`, è possibile aggiungere facoltativamente un tipo:

```
const x: number = 1;
```

TypeScript esegue un buon lavoro nell'inferenza dei tipi, soprattutto quando si tratta di tipi semplici, quindi queste dichiarazioni nella maggior parte dei casi non sono necessarie.

Sulle funzioni è possibile aggiungere annotazioni di tipo ai parametri:

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

Il seguente è un esempio che utilizza una funzione anonima (la cosiddetta funzione lambda):

```
const sum = (a: number, b: number) => a + b;
```

Queste annotazioni possono essere evitate quando è presente un valore predefinito per un parametro:

```
const sum = (a = 10, b: number) => a + b;
```

Le annotazioni del tipo di ritorno possono essere aggiunte alle funzioni:

```
const sum = (a = 10, b: number): number => a + b;
```

Questo è utile soprattutto per le funzioni più complesse, poiché scrivere esplicitamente il tipo di ritorno prima di un'implementazione può aiutare a pensare meglio alla funzione.

In genere, si consiglia di annotare le firme dei tipi, ma non le variabili locali del corpo, e di aggiungere i tipi sempre ai letterali degli oggetti.

Proprietà facoltative

Un oggetto può specificare Proprietà facoltative aggiungendo un punto interrogativo `?` alla fine del nome della proprietà:

```
type X = {
    a: number;
```

```
    b?: number; // Facoltativo
};
```

È possibile specificare un valore predefinito quando una proprietà è facoltativa”

```
type X = {
    a: number;
    b?: number;
};
const x = ({ a, b = 100 }: X) => a + b;
```

Proprietà di sola lettura

È possibile impedire la scrittura su una proprietà utilizzando il modificatore `readonly`, che assicura che la proprietà non possa essere riscritta ma non fornisce alcuna garanzia di immutabilità totale:

```
interface Y {
    readonly a: number;
}

type X = {
    readonly a: number;
};

type J = Readonly<{
    a: number;
}>;

type K = {
    readonly [index: number]: string;
};
```

Firme di indice

In TypeScript possiamo usare come firma di indice `string`, `number` e `symbol`:

```
type K = {
    [name: string | number]: string;
};
const k: K = { x: 'x', 1: 'b' };
console.log(k['x']);
console.log(k[1]);
console.log(k['1']); // Stesso risultato di k[1]
```

Si noti che JavaScript converte automaticamente un indice con `number` in un indice con `string`, quindi `k[1]` o `k["1"]` restituiscono lo stesso valore.

Estensione dei tipi

È possibile estendere un'`interfaccia` (copiare membri da un altro tipo):

```
interface X {
    a: string;
}
interface Y extends X {
    b: string;
}
```

È anche possibile estendere da più tipi:

```
interface A {
    a: string;
}
interface B {
    b: string;
}
interface Y extends A, B {
    y: string;
}
```

La parola chiave `extends` funziona solo su interfacce e classi; per i tipi utilizzare un'intersezione:

```
type A = {
    a: number;
};
type B = {
    b: number;
};
type C = A & B;
```

È possibile estendere un tipo utilizzando un'inferenza, ma non viceversa:

```
type A = {
    a: string;
};
interface B extends A {
    b: string;
}
```

Tipi letterali

Un tipo letterale è un singolo insieme di elementi di un tipo collettivo; definisce un valore molto preciso che è una primitiva JavaScript.

I tipi letterali in TypeScript sono numeri, stringhe e booleani.

Esempio di letterali:

```
const a = 'a'; // Stringa tipo letterale

const b = 1; // Numeric literal type
const c = true; // Boolean literal type
```

I tipi letterali stringa, numerico e booleano vengono utilizzati nell'unione, nella protezione dei tipi e negli alias di tipo. Nell'esempio seguente, è possibile vedere un alias di tipo unione. `O` è costituito solo dai valori specificati, nessun'altra stringa è valida:

```
type O = 'a' | 'b' | 'c';
```

Inferenza letterale

L'inferenza letterale è una funzionalità di TypeScript che consente di dedurre il tipo di una variabile o di un parametro in base al suo valore.

Nell'esempio seguente possiamo vedere che TypeScript considera `x` un tipo letterale in quanto il valore non può essere modificato in seguito, mentre `y` viene dedotto come stringa in quanto può essere modificato in seguito.

```
const x = 'x'; // Literal type of 'x', because this value cannot be
changed
let y = 'y'; // Type string, because we can change this value
```

Nell'esempio seguente possiamo vedere che `o.x` è stato dedotto come `string` (e non come un letterale di `a`), poiché TypeScript considera che il valore possa essere modificato in qualsiasi momento successivo.

```
type X = 'a' | 'b';

let o = {
    x: 'a', // Questa è una stringa più ampia
};

const fn = (x: X) => `${x}-foo`;

console.log(fn(o.x)); // L'argomento di tipo 'string' non è
assegnabile al parametro di tipo 'X'
```

Come puoi vedere, il codice genera un errore quando si passa `o.x` a `fn`, poiché X è un tipo più ristretto.

Possiamo risolvere questo problema utilizzando l'asserzione di tipo `const` o il tipo `X`:

```
let o = {
    x: 'a' as const,
```

```
};
```

oppure:

```
let o = {
    x: 'a' as X,
};
```

strictNullChecks

`strictNullChecks` è un'opzione del compilatore TypeScript che impone un controllo null rigoroso. Quando questa opzione è abilitata, variabili e parametri possono essere assegnati a `null` o `undefined` solo se sono stati dichiarati esplicitamente di quel tipo utilizzando l'unione `null | undefined`. Se una variabile o un parametro non viene dichiarato esplicitamente come nullable, TypeScript genererà un errore per prevenire potenziali errori di runtime.

Enumerazioni

In TypeScript, un `enum` è un insieme di valori costanti denominati.

```
enum Colore {
    Rosso = '#ff0000',
    Verde = '#00ff00',
    Blu = '#0000ff',
}
```

Gli enum possono essere definiti in diversi modi:

Enumerazioni numeriche

In TypeScript, un enum numerico è un enum in cui a ogni costante viene assegnato un valore numerico, a partire da 0 per impostazione predefinita.

```
enum Size {
    Small, // il valore inizia da 0
    Medium,
    Large,
}
```

È possibile specificare valori personalizzati assegnandoli esplicitamente:

```
enum Size {
    Small = 10,
    Medium,
    Large,
}
console.log(Size.Medium); // 11
```

Enum String

In TypeScript, un enum String è un enum in cui a ogni costante viene assegnato un valore stringa.

```
enum Language {
    English = 'EN',
    Spanish = 'ES',
}
```

Nota: TypeScript consente l'utilizzo di enum eterogenei in cui stringhe e membri numerici possono coesistere.

Enum Constant

Un enum Constant in TypeScript è un tipo speciale di enum in cui tutti i valori sono noti in fase di compilazione e vengono inlineati ovunque venga utilizzato l'enum, con conseguente maggiore efficienza del codice.

```
const enum Language {
    English = 'EN',
```

```
    Spanish = 'ES',
}
console.log(Language.English);
```

Verrà compilato in:

```
console.log('EN' /* Language.English */);
```

Note:

Gli enum costanti hanno valori hardcoded, che cancellano l'enum, il che può essere più efficiente nelle librerie autonome, ma generalmente non è auspicabile. Inoltre, gli enum costanti non possono avere membri calcolati.

Mapping inverso

In TypeScript, i mapping inversi negli enum si riferiscono alla possibilità di recuperare il nome del membro dell'enum dal suo valore. Per impostazione predefinita, i membri dell'enum hanno mapping in avanti dal nome al valore, ma i mapping inversi possono essere creati impostando esplicitamente i valori per ciascun membro. I mapping inversi sono utili quando è necessario cercare un membro dell'enum in base al suo valore o quando è necessario iterare su tutti i membri dell'enum. Si noti che solo i membri dell'enum numerico genereranno mapping inversi, mentre i membri dell'enum stringa non generano alcun mapping inverso.

Il seguente enum:

```
enum Grade {
    A = 90,
    B = 80,
    C = 70,
    F = 'fail',
}
```

Compila in:

```
"use strict";
var Grade;
(function (Grade) {
  Grade[(Grade["A"] = 90)] = "A";
  Grade[(Grade["B"] = 80)] = "B";
  Grade[(Grade["C"] = 70)] = "C";
  Grade["F"] = "fail";
})(Grade || (Grade = {}));
```

Pertanto, la mappatura dei valori alle chiavi funziona per i membri enum numerici, ma non per i membri enum stringa:

```
enum Grade {
    A = 90,
    B = 80,
    C = 70,
    F = 'fail',
}
const myGrade = Grade.A;
console.log(Grade[myGrade]); // A
console.log(Grade[90]); // A

const failGrade = Grade.F;
console.log(failGrade); // fail
console.log(Grade[failGrade]); // Element ha implicitamente un tipo
'any' perché l'espressione indice non è di tipo 'number'.
```

Enum ambientali

Un enum ambientale in TypeScript è un tipo di Enum definito in un file di dichiarazione (*.d.ts) senza un'implementazione associata. Permette di definire un set di costanti denominate che possono essere utilizzate in modo sicuro tra file diversi senza dover importare i dettagli di implementazione in ogni file.

Membri calcolati e costanti

In TypeScript, un membro calcolato è un membro di un Enum il cui valore è calcolato in fase di esecuzione, mentre un membro costante

è un membro il cui valore è impostato in fase di compilazione e non può essere modificato in fase di esecuzione. I membri calcolati sono consentiti negli Enum normali, mentre i membri costanti sono consentiti sia negli enum normali che in quelli costanti.

```
// Membri costanti
enum Color {
    Red = 1,
    Green = 5,
    Blue = Red + Green,
}
console.log(Color.Blue); // 6 generazioni in fase di compilazione

// Membri calcolati
enum Color {
    Red = 1,
    Green = Math.pow(2, 2),
    Blue = Math.floor(Math.random() * 3) + 1,
}
console.log(Color.Blue); // numero casuale generato in fase di
esecuzione
```

Gli enum sono indicati da unioni che comprendono i loro tipi di membri. I valori di ciascun membro possono essere determinati tramite espressioni costanti o non costanti, con i membri che possiedono valori costanti a cui vengono assegnati tipi letterali. Per illustrare, si consideri la dichiarazione del tipo E e dei suoi sottotipi E.A, E.B ed E.C. In questo caso, E rappresenta l'unione E.A | E.B | E.C.

```
const identity = (value: number) => value;

enum E {
    A = 2 * 5, // Letterale numerico
    B = 'bar', // Letterale stringa
    C = identity(42), // Calcolato opaco
}

console.log(E.C); //42
```

Restringimento

Il restringimento di TypeScript è il processo di perfezionamento del tipo di una variabile all'interno di un blocco condizionale. Questo è utile quando si lavora con tipi union, in cui una variabile può avere più di un tipo.

TypeScript riconosce diversi modi per restringere il tipo:

protezioni di tipo typeof

Il type guard typeof è uno specifico type guard in TypeScript che controlla il tipo di una variabile in base al suo tipo JavaScript predefinito.

```
const fn = (x: number | string) => {
    if (typeof x === 'number') {
        return x + 1; // x è un numero
    }
    return -1;
};
```

Restringimento di veridicità

Il restringimento di veridicità in TypeScript funziona verificando se una variabile è vera o falsa, per restringerne di conseguenza il tipo.

```
const toUpperCase = (name: string | null) => {
    if (name) {
        return name.toUpperCase();
    } else {
        return null;
    }
};
```

Restringimento di uguaglianza

Il restringimento di uguaglianza in TypeScript funziona verificando se una variabile è uguale o meno a un valore specifico, per restringerne di conseguenza il tipo.

Viene utilizzato insieme alle istruzioni `switch` e agli operatori di uguaglianza come `===`, `!==`, `==` e `!=` per restringere i tipi.

```
const checkStatus = (status: 'success' | 'error') => {
    switch (status) {
        case 'success':
            return true;
        case 'error':
            return null;
    }
};
```

Restringimento dell'operatore `in`

Il restringimento dell'operatore `in` in TypeScript è un modo per restringere il tipo di una variabile in base all'esistenza di una proprietà all'interno del tipo della variabile.

```
type Dog = {
    name: string;
    breed: string;
};

type Cat = {
    name: string;
    likesCream: boolean;
};

const getAnimalType = (pet: Dog | Cat) => {
    if ('breed' in pet) {
        return 'dog';
    } else {
        return 'cat';
```

```
    }
};
```

Restringimento instanceof

L'operatore di restringimento `instanceof` in TypeScript è un modo per restringere il tipo di una variabile in base alla sua funzione costruttore, verificando se un oggetto è un'istanza di una determinata classe o interfaccia.

```
class Square {
    constructor(public width: number) {}
}
class Rectangle {
    constructor(
        public width: number,
        public height: number
    ) {}
}
function area(shape: Square | Rectangle) {
    if (shape instanceof Square) {
        return shape.width * shape.width;
    } else {
        return shape.width * shape.height;
    }
}
const square = new Square(5);
const rectangle = new Rectangle(5, 10);
console.log(area(square)); // 25
console.log(area(rectangle)); // 50
```

Assegnazioni

Il restringimento TypeScript tramite assegnazioni è un modo per restringere il tipo di una variabile in base al valore assegnato. Quando a una variabile viene assegnato un valore, TypeScript ne deduce il tipo in base al valore assegnato e restringe il tipo della variabile in modo che corrisponda al tipo dedotto.

```
let value: string | number;
value = 'hello';
if (typeof value === 'string') {
    console.log(value.toUpperCase());
}
value = 42;
if (typeof value === 'number') {
    console.log(value.toFixed(2));
}
```

Analisi del flusso di controllo

L'analisi del flusso di controllo in TypeScript è un modo per analizzare staticamente il flusso di codice per dedurre i tipi di variabili, consentendo al compilatore di restringere i tipi di tali variabili secondo necessità, in base ai risultati dell'analisi.

Prima di TypeScript 4.4, l'analisi del flusso di codice veniva applicata solo al codice all'interno di un'istruzione if, ma da TypeScript 4.4 può essere applicata anche alle espressioni condizionali e agli accessi alle proprietà discriminanti referenziati indirettamente tramite variabili const.

Ad esempio:

```
const f1 = (x: unknown) => {
    const isString = typeof x === 'string';
    if (isString) {
        x.length;
    }
};

const f2 = (
    obj: { kind: 'foo'; foo: string } | { kind: 'bar'; bar: number
}
) => {
    const isFoo = obj.kind === 'foo';
    if (isFoo) {
        obj.foo;
```

```
    } else {
        obj.bar;
    }
};
```

Alcuni esempi in cui il restringimento non avviene:

```
const f1 = (x: unknown) => {
    let isString = typeof x === 'string';
    if (isString) {
        x.length; // Errore, nessun restringimento perché isString
non è costante
    }
};

const f6 = (
    obj: { kind: 'foo'; foo: string } | { kind: 'bar'; bar: number
}
) => {
    const isFoo = obj.kind === 'foo';
    obj = obj;
    if (isFoo) {
        obj.foo; // Errore, nessun restringimento perché obj è
assegnato nel corpo della funzione
    }
};
```

Note: Nelle espressioni condizionali vengono analizzati fino a cinque livelli di indirezione.

Predicati di tipo

I predicati di tipo in TypeScript sono funzioni che restituiscono un valore booleano e vengono utilizzate per restringere il tipo di una variabile a un tipo più specifico.

```
const isString = (value: unknown): value is string => typeof value
=== 'string';

const foo = (bar: unknown) => {
```

```
    if (isString(bar)) {
        console.log(bar.toUpperCase());
    } else {
        console.log('not a string');
    }
};
```

Unioni Discriminate

Le unioni Discriminate in TypeScript sono un tipo di unione che utilizza una proprietà comune, nota come discriminante, per restringere l'insieme dei tipi possibili per l'unione.

```
type Square = {
    kind: 'square'; // Discriminante
    size: number;
};

type Circle = {
    kind: 'circle'; // Discriminante
    radius: number;
};

type Shape = Square | Circle;

const area = (shape: Shape) => {
    switch (shape.kind) {
        case 'square':
            return Math.pow(shape.size, 2);
        case 'circle':
            return Math.PI * Math.pow(shape.radius, 2);
    }
};

const square: Square = { kind: 'square', size: 5 };
const circle: Circle = { kind: 'circle', radius: 2 };

console.log(area(square)); // 25
console.log(area(circle)); // 12.566370614359172
```

Il tipo never

Quando una variabile viene ristretta a un tipo che non può contenere alcun valore, il compilatore TypeScript dedurrà che la variabile deve essere del tipo `never`. Questo perché il tipo `never` rappresenta un valore che non può mai essere prodotto.

```
const printValue = (val: string | number) => {
    if (typeof val === 'string') {
        console.log(val.toUpperCase());
    } else if (typeof val === 'number') {
        console.log(val.toFixed(2));
    } else {
        // val ha il tipo never qui perché non può essere altro che
una stringa o un numero
        const neverVal: never = val;
        console.log(`Valore imprevisto: ${neverVal}`);
    }
};
```

Controllo di esaustività

Il controllo di esaustività è una funzionalità di TypeScript che garantisce che tutti i possibili casi di unione discriminata vengano gestiti in un'istruzione `switch` o in un'istruzione `if`.

```
type Direction = 'up' | 'down';

const move = (direction: Direction) => {
    switch (direction) {
        case 'up':
            console.log("Spostamento verso l'alto");
            break;
        case 'down':
            console.log('Spostamento verso il basso');
            break;
        default:
            const exhaustiveCheck: never = direction;
```

```
            console.log(exhaustiveCheck); // Questa riga non verrà
mai eseguita
    }
};
```

Il tipo `never` viene utilizzato per garantire che il caso predefinito sia esaustivo e che TypeScript generi un errore se un nuovo valore viene aggiunto al tipo Direction senza essere gestito nell'istruzione switch.

Tipi di oggetto

In TypeScript, i tipi di oggetto descrivono la forma di un oggetto. Specificano i nomi e i tipi delle proprietà dell'oggetto, nonché se tali proprietà sono obbligatorie o facoltative.

In TypeScript, è possibile definire i tipi di oggetto in due modi principali:

interface, che definisce la forma di un oggetto specificando i nomi, i tipi e l'opzionalità delle sue proprietà.

```
interface User {
    name: string;
    age: number;
    email?: string;
}
```

Un alias di tipo, simile a un'interfaccia, definisce la forma di un oggetto. Tuttavia, può anche creare un nuovo tipo personalizzato basato su un tipo esistente o su una combinazione di tipi esistenti. Ciò include la definizione di tipi unione, tipi intersezione e altri tipi complessi.

```
type Point = {
    x: number;
    y: number;
};
```

È anche possibile definire un tipo in modo anonimo:

```
const sum = (x: { a: number; b: number }) => x.a + x.b;
console.log(sum({ a: 5, b: 1 }));
```

Tipo di tupla (anonimo)

Un tipo di tupla è un tipo che rappresenta un array con un numero fisso di elementi e i relativi tipi. Un tipo di tupla impone un numero specifico di elementi e i rispettivi tipi in un ordine fisso. I tipi di tupla sono utili quando si desidera rappresentare una raccolta di valori con tipi specifici, dove la posizione di ciascun elemento nell'array ha un significato specifico.

```
type Point = [number, number];
```

Tipo di tupla denominato (etichettato)

I tipi di tupla possono includere etichette o nomi opzionali per ciascun elemento. Queste etichette servono per migliorare la leggibilità e facilitare l'utilizzo degli strumenti e non influiscono sulle operazioni che è possibile eseguire con esse.

```
type T = string;
type Tuple1 = [T, T];
type Tuple2 = [a: T, b: T];
type Tuple3 = [a: T, T]; // Tupla con nome più Tupla anonima
```

Tupla a lunghezza fissa

Una tupla a lunghezza fissa è un tipo specifico di tupla che impone un numero fisso di elementi di tipi specifici e non consente alcuna modifica alla lunghezza della tupla una volta definita.

Le tuple a lunghezza fissa sono utili quando è necessario rappresentare una raccolta di valori con un numero specifico di elementi e tipi specifici e si desidera garantire che la lunghezza e i tipi della tupla non possano essere modificati inavvertitamente.

```
const x = [10, 'hello'] as const;
x.push(2); // Errore
```

Tipo Unione

Un Tipo Unione è un tipo che rappresenta un valore che può essere di diversi tipi. I Tipi Unione sono indicati con il simbolo `|` tra ogni tipo possibile.

```
let x: string | number;
x = 'hello'; // Valido
x = 123; // Valido
```

Tipi Intersezione

Un Tipo Intersezione è un tipo che rappresenta un valore che ha tutte le proprietà di due o più tipi. I Tipi Intersezione sono indicati con il simbolo `&` tra ogni tipo.

```
type X = {
    a: string;
};

type Y = {
    b: string;
};

type J = X & Y; // Intersezione

const j: J = {
    a: 'a',
```

```
    b: 'b',
};
```

Indicizzazione dei tipi

L'indicizzazione dei tipi si riferisce alla capacità di definire tipi che possono essere indicizzati da una chiave non nota in anticipo, utilizzando una firma di indice per specificare il tipo per le proprietà che non sono dichiarate esplicitamente.

```
type Dictionary<T> = {
    [key: string]: T;
};
const myDict: Dictionary<string> = { a: 'a', b: 'b' };

console.log(myDict['a']); // Restituisce un
```

Tipo da Valore

In TypeScript, il tipo da valore si riferisce all'inferenza automatica di un tipo da un valore o da un'espressione tramite inferenza di tipo.

```
const x = 'x'; // TypeScript inferisce 'x' come una stringa
letterale con 'const' (immutabile), ma lo amplia a 'string' con
'let' (riassegnabile).
```

Tipo da Ritorno Funzione

Il tipo da Ritorno Funzione si riferisce alla possibilità di inferire automaticamente il tipo di ritorno di una funzione in base alla sua implementazione. Ciò consente a TypeScript di determinare il tipo del valore restituito dalla funzione senza annotazioni di tipo esplicite.

```
const add = (x: number, y: number) => x + y; // TypeScript può
dedurre che il tipo restituito dalla funzione sia un numero
```

Tipo da modulo

Il tipo da modulo si riferisce alla possibilità di utilizzare i valori esportati di un modulo per dedurne automaticamente il tipo. Quando un modulo esporta un valore con un tipo specifico, TypeScript può utilizzare tali informazioni per dedurre automaticamente il tipo di quel valore quando viene importato in un altro modulo.

```
// calc.ts
export const add = (x: number, y: number) => x + y;
// index.ts
import { add } from 'calc';
const r = add(1, 2); // r è un numero
```

Tipi mappati

I tipi mappati in TypeScript consentono di creare nuovi tipi basati su un tipo esistente trasformando ciascuna proprietà tramite una funzione di mappatura. Mappando i tipi esistenti, è possibile creare nuovi tipi che rappresentano le stesse informazioni in un formato diverso. Per creare un tipo mappato, si accede alle proprietà di un tipo esistente utilizzando l'operatore `keyof` e quindi le si modifica per produrre un nuovo tipo. Nell'esempio seguente:

```
type MyMappedType<T> = {
    [P in keyof T]: T[P][];
};
type MyType = {
    foo: string;
    bar: number;
};
type MyNewType = MyMappedType<MyType>;
const x: MyNewType = {
    foo: ['hello', 'world'],
    bar: [1, 2, 3],
};
```

Definiamo MyMappedType per mappare le proprietà di T, creando un nuovo tipo con ogni proprietà come array del suo tipo originale. In questo modo, creiamo MyNewType per rappresentare le stesse informazioni di MyType, ma con ogni proprietà come array.

Modificatori di tipo mappati

I modificatori di tipo mappati in TypeScript consentono la trasformazione delle proprietà all'interno di un tipo esistente:

- `readonly` o `+readonly`: questo rende una proprietà nel tipo mappato di sola lettura.
- `-readonly`: questo consente a una proprietà nel tipo mappato di essere modificabile.
- `?`: questo designa una proprietà nel tipo mappato come facoltativa.

Esempi:

```
type ReadOnly<T> = { readonly [P in keyof T]: T[P] }; // Tutte le proprietà contrassegnate come di sola lettura

type Mutable<T> = { -readonly [P in keyof T]: T[P] }; // Tutte le proprietà contrassegnate come modificabili

type MyPartial<T> = { [P in keyof T]?: T[P] }; // Tutte le proprietà contrassegnate come facoltative
```

Tipi condizionali

I tipi condizionali sono un modo per creare un tipo che dipende da una condizione, in cui il tipo da creare viene determinato in base al risultato della condizione. Sono definiti utilizzando la parola chiave `extends` e un operatore ternario per scegliere condizionatamente tra due tipi.

```ts
type IsArray<T> = T extends any[] ? true : false;

const myArray = [1, 2, 3];
const myNumber = 42;

type IsMyArrayAnArray = IsArray<typeof myArray>; // Type true
type IsMyNumberAnArray = IsArray<typeof myNumber>; // Type false
```

Tipi condizionali distributivi

I tipi condizionali distributivi sono una funzionalità che consente di distribuire un tipo su un'unione di tipi, applicando una trasformazione a ciascun membro dell'unione individualmente. Questo può essere particolarmente utile quando si lavora con tipi mappati o tipi di ordine superiore.

```ts
type Nullable<T> = T extends any ? T | null : never;
type NumberOrBool = number | boolean;
type NullableNumberOrBool = Nullable<NumberOrBool>; // number |
boolean | null
```

infer Inferenza di tipo nei tipi condizionali

La parola chiave `infer` viene utilizzata nei tipi condizionali per inferire (estrarre) il tipo di un parametro generico da un tipo che dipende da esso. Questo consente di scrivere definizioni di tipo più flessibili e riutilizzabili.

```ts
type ElementType<T> = T extends (infer U)[] ? U : never;
type Numbers = ElementType<number[]>; // number
type Strings = ElementType<string[]>; // string
```

Tipi Condizionali Predefiniti

In TypeScript, i Tipi Condizionali Predefiniti sono tipi condizionali integrati forniti dal linguaggio. Sono progettati per eseguire trasformazioni di tipo comuni in base alle caratteristiche di un dato tipo.

`Exclude<UnionType, ExcludedType>`: questo tipo rimuove da Type tutti i tipi assegnabili a ExcludedType.

`Extract<Type, Union>`: questo tipo estrae da Union tutti i tipi assegnabili a Type.

`NonNullable<Type>`: questo tipo rimuove null e undefined da Type.

`ReturnType<Type>`: questo tipo estrae il tipo di ritorno di un Type di funzione.

`Parameters<Type>`: questo tipo estrae i tipi di parametro di un Type di funzione.

`Required<Type>`: Questo tipo rende obbligatorie tutte le proprietà in Type.

`Partial<Type>`: Questo tipo rende facoltative tutte le proprietà in Type.

`Readonly<Type>`: Questo tipo rende di sola lettura tutte le proprietà in Type.

Tipi di unione di template

I tipi di unione di template possono essere utilizzati per unire e manipolare il testo all'interno del sistema di tipi, ad esempio:

```ts
type Status = 'active' | 'inactive';
type Products = 'p1' | 'p2';
type ProductId = `id-${Products}-${Status}`; // "id-p1-active" |
"id-p1-inactive" | "id-p2-active" | "id-p2-inactive"
```

Tipo Any

Il tipo `any` è un tipo speciale (supertipo universale) che può essere utilizzato per rappresentare qualsiasi tipo di valore (primitive, oggetti, array, funzioni, errori, simboli). Viene spesso utilizzato in situazioni in cui il tipo di un valore non è noto in fase di compilazione, o quando si lavora con valori provenienti da API o librerie esterne che non dispongono di tipi TypeScript.

Utilizzando il tipo `any`, si indica al compilatore TypeScript che i valori devono essere rappresentati senza alcuna limitazione. Per massimizzare la sicurezza dei tipi nel codice, tieni presente quanto segue:

- Limitare l'utilizzo di `any` a casi specifici in cui il tipo è realmente sconosciuto.
- Non restituire `any` da una funzione, poiché ciò indebolisce la sicurezza del tipo nel codice che lo utilizza.
- Invece di `any`, utilizzare `@ts-ignore` se è necessario silenziare il compilatore.

```ts
let value: any;
value = true; // Valido
value = 7; // Valido
```

Tipo Unknown

In TypeScript, il tipo `unknown` rappresenta un valore di tipo sconosciuto. A differenza del tipo `any`, che consente qualsiasi tipo di valore, `unknown` richiede un controllo o un'asserzione di tipo prima di

poter essere utilizzato in un modo specifico, quindi non sono consentite operazioni su un `unknown` senza prima aver effettuato un'asserzione o aver limitato il campo a un tipo più specifico.

Il tipo `unknown` è assegnabile solo a qualsiasi tipo e il tipo `unknown` stesso è un'alternativa type-safe ad `any`.

```
let value: unknown;

let value1: unknown = value; // Valido
let value2: any = value; // Valido
let value3: boolean = value; // Non valido
let value4: number = value; // Non valido

const add = (a: unknown, b: unknown): number | undefined =>
    typeof a === 'number' && typeof b === 'number' ? a + b :
undefined;
console.log(add(1, 2)); // 3
console.log(add('x', 2)); // non definito
```

Tipo Void

Il tipo `void` viene utilizzato per indicare che una funzione non restituisce un valore.

```
const sayHello = (): void => {
    console.log('Hello!');
};
```

Tipo Never

Il tipo `never` rappresenta valori che non si verificano mai. Viene utilizzato per indicare funzioni o espressioni che non restituiscono mai né generano errori.

Ad esempio, un ciclo infinito:

```
const infiniteLoop = (): never => {
    while (true) {
        // fai qualcosa
    }
};
```

Generazione di un errore:

```
const throwError = (message: string): never => {
    throw new Error(message);
};
```

Il tipo `never` è utile per garantire la sicurezza dei tipi e rilevare potenziali errori nel codice. Aiuta TypeScript ad analizzare e dedurre tipi più precisi se utilizzato in combinazione con altri tipi e istruzioni di controllo del flusso, ad esempio:

```
type Direction = 'up' | 'down';
const move = (direction: Direction): void => {
    switch (direction) {
        case 'up':
            // sposta verso l'alto
            break;
        case 'down':
            // sposta verso il basso
            break;
        default:
            const exhaustiveCheck: never = direction;
            throw new Error(`Unhandled direction:
${exhaustiveCheck}`);
    }
};
```

Interfaccia e tipo

Sintassi comune

In TypeScript, le interfacce definiscono la struttura degli oggetti, specificando i nomi e i tipi di proprietà o metodi che un oggetto deve

avere. La sintassi comune per definire un'interfaccia in TypeScript è la seguente:

```
interface InterfaceName {
    property1: Type1;
    // ...
    method1(arg1: ArgType1, arg2: ArgType2): ReturnType;
    // ...
}
```

Analogamente per la definizione del tipo:

```
type TypeName = {
    property1: Type1;
    // ...
    method1(arg1: ArgType1, arg2: ArgType2): ReturnType;
    // ...
};
```

`interface InterfaceName` o `type TypeName`: Definisce il nome dell'interfaccia. `property1: Type1`: Specifica le proprietà dell'interfaccia insieme ai tipi corrispondenti. È possibile definire più proprietà, ciascuna separata da un punto e virgola. `method1(arg1: ArgType1, arg2: ArgType2): ReturnType;`: Specifica i metodi dell'interfaccia. I metodi sono definiti con i loro nomi, seguiti da un elenco di parametri tra parentesi e dal tipo di ritorno. È possibile definire più metodi, ciascuno separato da un punto e virgola.

Esempio di interfaccia:

```
interface Person {
    name: string;
    age: number;
    greet(): void;
}
```

Esempio di tipo:

```
type TypeName = {
    property1: string;
```

```
    method1(arg1: string, arg2: string): string;
};
```

In TypeScript, i tipi vengono utilizzati per definire la forma dei dati e applicare il controllo dei tipi. Esistono diverse sintassi comuni per la definizione dei tipi in TypeScript, a seconda del caso d'uso specifico. Ecco alcuni esempi:

Tipi di base

```
let myNumber: number = 123; // number type
let myBoolean: boolean = true; // boolean type
let myArray: string[] = ['a', 'b']; // array di stringhe
let myTuple: [string, number] = ['a', 123]; // tupla
```

Oggetti e interfacce

```
const x: { name: string; age: number } = { name: 'Simon', age: 7 };
```

Tipi di unione e intersezione

```
type MyType = string | number; // Union type
let myUnion: MyType = 'hello'; // Can be a string
myUnion = 123; // Or a number

type TypeA = { name: string };
type TypeB = { age: number };
type CombinedType = TypeA & TypeB; // Intersection type
let myCombined: CombinedType = { name: 'John', age: 25 }; // Object
with name and age properties
```

Primitive di tipo predefinite

TypeScript dispone di diverse primitive di tipo predefinite che possono essere utilizzate per definire variabili, parametri di funzione

e tipi restituiti:

- `number`: rappresenta valori numerici, inclusi numeri interi e numeri in virgola mobile.
- `string`: rappresenta dati testuali.
- `boolean`: rappresenta valori logici, che possono essere true o false.
- `null`: rappresenta l'assenza di un valore.
- `undefined`: rappresenta un valore che non è stato assegnato o non è stato definito.
- `symbol`: rappresenta un identificatore univoco. I simboli vengono in genere utilizzati come chiavi per le proprietà degli oggetti.
- `bigint`: rappresenta numeri interi con precisione arbitraria.
- `any`: rappresenta un tipo dinamico o sconosciuto. Le variabili di tipo any possono contenere valori di qualsiasi tipo e ignorano il controllo del tipo. * `void`: rappresenta l'assenza di qualsiasi tipo. È comunemente usato come tipo di ritorno di funzioni che non restituiscono alcun valore.
- `never`: rappresenta un tipo per valori che non si verificano mai. È tipicamente usato come tipo di ritorno di funzioni che generano un errore o entrano in un ciclo infinito.

Oggetti JavaScript predefiniti comuni

TypeScript è un superset di JavaScript e include tutti gli oggetti JavaScript predefiniti comunemente usati. Un elenco completo di questi oggetti è disponibile sul sito web di documentazione di Mozilla Developer Network (MDN): https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

Ecco un elenco di alcuni oggetti JavaScript predefiniti comunemente utilizzati:

- Function
- Object
- Boolean

- Error
- Number
- BigInt
- Math
- Date
- String
- RegExp
- Array
- Map
- Set
- Promise
- Intl

Sovraccarichi

Gli overload di funzione in TypeScript consentono di definire più firme di funzione per un singolo nome di funzione, consentendo di definire funzioni che possono essere chiamate in più modi. Ecco un esempio:

```
// Sovraccarichi
function sayHi(name: string): string;
function sayHi(names: string[]): string[];

// Implementazione
function sayHi(name: unknown): unknown {
    if (typeof name === 'string') {
        return `Ciao, ${name}!`;
    } else if (Array.isArray(name)) {
        return name.map(name => `Ciao, ${name}!`);
    }
    throw new Error('Valore non valido');
}

sayHi('xx'); // Valido
sayHi(['aa', 'bb']); // Valido
```

Ecco un altro esempio di utilizzo di overload di funzione all'interno di una `class`:

```
class Greeter {
    message: string;

    constructor(message: string) {
        this.message = message;
    }

    // Sovraccarichi
    sayHi(name: string): string;
    sayHi(name: string[]): ReadonlyArray<string>;

    // Implementazione
    sayHi(name: unknown): unknown {
        if (typeof name === 'string') {
            return `${this.message}, ${name}!`;
        } else if (Array.isArray(name)) {
            return name.map(name => `${this.message}, ${name}!`);
        }
        throw new Error('value is invalid');
    }
}
console.log(new Greeter('Hello').sayHi('Simon'));
```

Unione ed estensione

Merging ed estensione si riferiscono a due concetti diversi relativi all'utilizzo di tipi e interfacce.

Merging consente di combinare più dichiarazioni con lo stesso nome in un'unica definizione, ad esempio quando si definisce un'interfaccia con lo stesso nome più volte:

```
interface X {
    a: string;
}

interface X {
```

```
    b: number;
}

const person: X = {
    a: 'a',
    b: 7,
};
```

L'estensione si riferisce alla possibilità di estendere o ereditare da tipi o interfacce esistenti per crearne di nuovi. È un meccanismo per aggiungere proprietà o metodi aggiuntivi a un tipo esistente senza modificarne la definizione originale. Esempio:

```
interface Animal {
    name: string;
    eat(): void;
}

interface Bird extends Animal {
    sing(): void;
}

const dog: Bird = {
    name: 'Bird 1',
    eat() {
        console.log('Eating');
    },
    sing() {
        console.log('Singing');
    },
};
```

Differenze tra tipo e interfaccia

Unione delle dichiarazioni (aumento):

Le interfacce supportano l'unione delle dichiarazioni, il che significa che è possibile definire più interfacce con lo stesso nome e TypeScript le unirà in un'unica interfaccia con le proprietà e i metodi

combinati. D'altra parte, i tipi non supportano l'unione delle dichiarazioni. Questo può essere utile quando si desidera aggiungere funzionalità extra o personalizzare i tipi esistenti senza modificare le definizioni originali o correggere tipi mancanti o errati.

```
interface A {
    x: string;
}
interface A {
    y: string;
}
const j: A = {
    x: 'xx',
    y: 'yy',
};
```

Estensione di altri tipi/interfacce:

Sia i tipi che le interfacce possono estendere altri tipi/interfacce, ma la sintassi è diversa. Con le interfacce, si utilizza la parola chiave `extends` per ereditare proprietà e metodi da altre interfacce. Tuttavia, un'interfaccia non può estendere un tipo complesso come un tipo unione.

```
interface A {
    x: string;
    y: number;
}
interface B extends A {
    z: string;
}
const car: B = {
    x: 'x',
    y: 123,
    z: 'z',
};
```

Per i tipi, si utilizza l'operatore & per combinare più tipi in un unico tipo (intersezione).

```
interface A {
    x: string;
    y: number;
}

type B = A & {
    j: string;
};

const c: B = {
    x: 'x',
    y: 123,
    j: 'j',
};
```

Tipi di unione e intersezione:

I tipi sono più flessibili quando si tratta di definire tipi di unione e intersezione. Con la parola chiave `type`, è possibile creare facilmente tipi di unione utilizzando l'operatore `|` e tipi di intersezione utilizzando l'operatore `&`. Sebbene le interfacce possano anche rappresentare tipi di unione indirettamente, non dispongono di supporto integrato per i tipi di intersezione.

```
type Department = 'dep-x' | 'dep-y'; // Unione

type Person = {
    name: string;
    age: number;
};

type Employee = {
    id: number;
    department: Department;
};

type EmployeeInfo = Person & Employee; // Intersezione
```

Esempio con interfacce:

```
interface A {
    x: 'x';
}
interface B {
    y: 'y';
}

type C = A | B; // Unione di interfacce
```

Classe

Sintassi comune della classe

La parola chiave `class` viene utilizzata in TypeScript per definire una classe. Di seguito è riportato un esempio:

```
class Person {
    private name: string;
    private age: number;
    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }
    public sayHi(): void {
        console.log(`Ciao, mi chiamo ${this.name} e ho ${this.age}
anni.`);
    }
}
```

La parola chiave `class` viene utilizzata per definire una classe denominata “Person”.

La classe ha due proprietà private: name di tipo `string` ed age di tipo `number`.

Il costruttore viene definito utilizzando la parola chiave `constructor`. Accetta name ed age come parametri e li assegna alle proprietà corrispondenti.

La classe ha un metodo `public` denominato sayHi che registra un messaggio di saluto.

Per creare un'istanza di una classe in TypeScript, è possibile utilizzare la parola chiave `new` seguita dal nome della classe, seguito da parentesi `()`. Ad esempio:

```
const myObject = new Person('John Doe', 25);
myObject.sayHi(); // Output: Ciao, mi chiamo John Doe e ho 25 anni.
```

Costruttore

I costruttori sono metodi speciali all'interno di una classe che vengono utilizzati per inizializzare le proprietà dell'oggetto quando viene creata un'istanza della classe.

```
class Person {
    public name: string;
    public age: number;

    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }

    sayHello() {
        console.log(`Ciao, mi chiamo ${this.name} e ho ${this.age}
anni.`);
    }
}

const john = new Person('Simon', 17);
john.sayHello();
```

È possibile sovraccaricare un costruttore utilizzando la seguente sintassi:

```
type Sex = 'm' | 'f';
```

```
class Person {
    name: string;
    age: number;
    sex: Sex;

    constructor(name: string, age: number, sex?: Sex);
    constructor(name: string, age: number, sex: Sex) {
        this.name = name;
        this.age = age;
        this.sex = sex ?? 'm';
    }
}

const p1 = new Person('Simon', 17);
const p2 = new Person('Alice', 22, 'f');
```

In TypeScript, è possibile definire più overload del costruttore, ma è possibile avere una sola implementazione che deve essere compatibile con tutti gli overload; questo si può ottenere utilizzando un parametro opzionale.

```
class Person {
    name: string;
    age: number;

    constructor();
    constructor(name: string);
    constructor(name: string, age: number);
    constructor(name?: string, age?: number) {
        this.name = name ?? 'Sconosciuto';
        this.age = age ?? 0;
    }

    displayInfo() {
        console.log(`Nome: ${this.name}, Età: ${this.age}`);
    }
}

const person1 = new Person();
person1.displayInfo(); // Nome: unknown, Età: 0

const person2 = new Person('John');
```

```
person2.displayInfo(); // Nome: John, Età: 0

const person3 = new Person('Jane', 25);
person3.displayInfo(); // Nome: Jane, Età: 25
```

Costruttori privati e protetti

In TypeScript, i costruttori possono essere contrassegnati come privati o protetti, il che ne limita l'accessibilità e l'utilizzo.

Costruttori privati: possono essere chiamati solo all'interno della classe stessa. I costruttori privati vengono spesso utilizzati in scenari in cui si desidera applicare un pattern singleton o limitare la creazione di istanze a un metodo factory all'interno della classe.

Costruttori protetti: I costruttori protetti sono utili quando si desidera creare una classe base che non deve essere istanziata direttamente, ma può essere estesa tramite sottoclassi.

```
class BaseClass {
    protected constructor() {}
}

class DerivedClass extends BaseClass {
    private value: number;

    constructor(value: number) {
        super();
        this.value = value;
    }
}

// Il tentativo di istanziare direttamente la classe base genererà
un errore.
// const baseObj = new BaseClass(); // Errore: il costruttore della
classe 'BaseClass' è protetto.

// Crea un'istanza della classe derivata
const derivedObj = new DerivedClass(10);
```

Modificatori di accesso

I modificatori di accesso `private`, `protected` e `public` vengono utilizzati per controllare la visibilità e l'accessibilità dei membri della classe, come proprietà e metodi, nelle classi TypeScript. Questi modificatori sono essenziali per applicare l'incapsulamento e stabilire limiti per l'accesso e la modifica dello stato interno di una classe.

Il modificatore `private` limita l'accesso al membro della classe solo all'interno della classe contenitore.

Il modificatore `protected` consente l'accesso al membro della classe all'interno della classe contenitore e delle sue classi derivate.

Il modificatore `public` fornisce accesso illimitato al membro della classe, consentendone l'accesso da qualsiasi luogo.

Get e Set

Getter e setter sono metodi speciali che consentono di definire un comportamento personalizzato di accesso e modifica per le proprietà della classe. Consentono di incapsulare lo stato interno di un oggetto e di fornire logica aggiuntiva durante l'ottenimento o l'impostazione dei valori delle proprietà. In TypeScript, getter e setter sono definiti utilizzando rispettivamente le parole chiave `get` e `set`. Ecco un esempio:

```
class MyClass {
    private _myProperty: string;

    constructor(value: string) {
        this._myProperty = value;
    }
    get myProperty(): string {
        return this._myProperty;
    }
    set myProperty(value: string) {
```

```
        this._myProperty = value;
    }
}
```

Accessori automatici nelle classi

TypeScript versione 4.9 aggiunge il supporto per auto-accessor, una funzionalità ECMAScript di prossima uscita. Assomigliano alle proprietà di classe, ma sono dichiarate con la parola chiave "accessor".

```
class Animal {
    accessor name: string;

    constructor(name: string) {
        this.name = name;
    }
}
```

Gli auto-accessor vengono "de-sugared" in accessor privati `get` e `set`, che operano su una proprietà inaccessibile.

```
class Animal {
    #__name: string;

    get name() {
        return this.#__name;
    }
    set name(value: string) {
        this.#__name = name;
    }

    constructor(name: string) {
        this.name = name;
    }
}
```

this

In TypeScript, la parola chiave `this` si riferisce all'istanza corrente di una classe all'interno dei suoi metodi o costruttori. Permette di accedere e modificare le proprietà e i metodi della classe. dall'interno del proprio ambito. Fornisce un modo per accedere e manipolare lo stato interno di un oggetto all'interno dei propri metodi.

```
class Person {
    private name: string;
    constructor(name: string) {
        this.name = name;
    }
    public introduce(): void {
        console.log(`Ciao, mi chiamo ${this.name}.`);
    }
}

const person1 = new Person('Alice');
person1.introduce(); // Ciao, mi chiamo Alice.
```

Proprietà dei parametri

Le proprietà dei parametri consentono di dichiarare e inizializzare le proprietà della classe direttamente all'interno dei parametri del costruttore, evitando codice boilerplate, ad esempio:

```
class Person {
    constructor(
        private name: string,
        public age: number
    ) {
        // Le parole chiave "private" e "public" nel costruttore
        // dichiarano e inizializzano automaticamente le proprietà
della classe corrispondenti.
    }
    public introduce(): void {
        console.log(`Ciao, mi chiamo ${this.name} e ho ${this.age}
anni.`);
    }
```

```
}
const person = new Person('Alice', 25);
person.introduce();
```

Classi astratte

Le classi astratte sono utilizzate in TypeScript principalmente per l'ereditarietà, poiché forniscono un modo per definire proprietà e metodi comuni che possono essere ereditati dalle sottoclassi. Questo è utile quando si desidera definire un comportamento comune e imporre alle sottoclassi di implementare determinati metodi. Forniscono un modo per creare una gerarchia di classi in cui la classe base astratta fornisce un'interfaccia condivisa e funzionalità comuni per le sottoclassi.

```
abstract class Animal {
    protected name: string;

    constructor(name: string) {
        this.name = name;
    }

    abstract makeSound(): void;
}

class Cat extends Animal {
    makeSound(): void {
        console.log(`${this.name} miagola.`);
    }
}

const cat = new Cat('Whiskers');
cat.makeSound(); // Output: Whiskers miagola.
```

Con i generici

Le classi con i generici consentono di definire classi riutilizzabili che possono funzionare con tipi diversi.

```
class Container<T> {
    private item: T;

    constructor(item: T) {
        this.item = item;
    }

    getItem(): T {
        return this.item;
    }

    setItem(item: T): void {
        this.item = item;
    }
}

const container1 = new Container<number>(42);
console.log(container1.getItem()); // 42

const container2 = new Container<string>('Hello');
container2.setItem('World');
console.log(container2.getItem()); // Mondo
```

Decoratori

I decoratori forniscono un meccanismo per aggiungere metadati, modificare il comportamento, convalidare o estendere la funzionalità dell'elemento di destinazione. Sono funzioni che vengono eseguite in fase di esecuzione. È possibile applicare più decoratori a una dichiarazione.

I decoratori sono funzionalità sperimentali e gli esempi seguenti sono compatibili solo con TypeScript versione 5 o successive che utilizzano ES6.

Per le versioni di TypeScript precedenti alla 5, dovrebbero essere abilitati utilizzando la proprietà `experimentalDecorators` nel file `tsconfig.json` o utilizzando `--experimentalDecorators` nella riga di comando (ma l'esempio seguente non funzionerà).

Alcuni dei casi d'uso comuni per i decoratori includono:

- Monitoraggio delle modifiche delle proprietà.
- Monitoraggio delle chiamate ai metodi.
- Aggiunta di proprietà o metodi aggiuntivi.
- Validazione in fase di esecuzione.
- Serializzazione e deserializzazione automatica.
- Registrazione.
- Autorizzazione e autenticazione.
- Protezione dagli errori.

Nota: i decoratori per la versione 5 non consentono parametri di decorazione.

Tipi di decoratori:

Decoratori di classe

I decoratori di classe sono utili per estendere una classe esistente, ad esempio aggiungendo proprietà o metodi o raccogliendo istanze di una classe. Nell'esempio seguente, aggiungiamo un metodo `toString` che converte la classe in una rappresentazione in formato stringa.

```
type Constructor<T = {}> = new (...args: any[]) => T;

function toString<Class extends Constructor>(
    Value: Class,
    context: ClassDecoratorContext<Class>
) {
    return class extends Value {
        constructor(...args: any[]) {
            super(...args);
            console.log(JSON.stringify(this));
            console.log(JSON.stringify(context));
        }
    };
}

@toString
```

```
class Person {
    name: string;

    constructor(name: string) {
        this.name = name;
    }

    greet() {
        return 'Ciao,' + this.name;
    }
}
const person = new Person('Simon');
/* Log:
{"name":"Simon"}
{"kind":"class","name":"Person"}
*/
```

Decoratore di proprietà

I decoratori di proprietà sono utili per modificare il comportamento di una proprietà, ad esempio cambiando i valori di inizializzazione. Nel codice seguente, abbiamo uno script che imposta una proprietà in modo che sia sempre in maiuscolo:

```
function upperCase<T>(
    target: undefined,
    context: ClassFieldDecoratorContext<T, string>
) {
    return function (this: T, value: string) {
        return value.toUpperCase();
    };
}

class MyClass {
    @upperCase
    prop1 = 'hello!';
}

console.log(new MyClass().prop1); // Log: HELLO!
```

Decoratore di metodo

I decoratori di metodo consentono di modificare o migliorare il comportamento dei metodi. Di seguito è riportato un esempio di un semplice logger:

```
function log<This, Args extends any[], Return>(
    target: (this: This, ...args: Args) => Return,
    context: ClassMethodDecoratorContext<
        This,
        (this: This, ...args: Args) => Return
    >
) {
    const methodName = String(context.name);

    function replacementMethod(this: This, ...args: Args): Return {
        console.log(`LOG: Accesso al metodo '${methodName}'.`);
        const result = target.call(this, ...args);
        console.log(`LOG: Uscita dal metodo '${methodName}'.`);
        return result;
    }

    return replacementMethod;
}

class MyClass {
    @log
    sayHello() {
        console.log('Ciao!');
    }
}

new MyClass().sayHello();
```

Registra:

```
LOG: Accesso al metodo 'sayHello'.
Ciao!
LOG: Uscita dal metodo 'sayHello'.
```

Decoratori Getter e Setter

I decoratori Getter e Setter consentono di modificare o migliorare il comportamento degli accessor di classe. Sono utili, ad esempio, per convalidare le assegnazioni di proprietà. Ecco un semplice esempio di decoratore getter:

```
function range<This, Return extends number>(min: number, max:
number) {
    return function (
        target: (this: This) => Return,
        context: ClassGetterDecoratorContext<This, Return>
    ) {
        return function (this: This): Return {
            const value = target.call(this);
            if (value < min || value > max) {
                throw 'Invalid';
            }
            Object.defineProperty(this, context.name, {
                value,
                enumerable: true,
            });
            return value;
        };
    };
}

class MyClass {
    private _value = 0;

    constructor(value: number) {
        this._value = value;
    }
    @range(1, 100)
    get getValue(): number {
        return this._value;
    }
}

const obj = new MyClass(10);
console.log(obj.getValue); // Valido: 10
```

```
const obj2 = new MyClass(999);
console.log(obj2.getValue); // Throw: Invalid!
```

Metadati del decoratore

I metadati del decoratore semplificano il processo per i decoratori di applicare e utilizzare i metadati in qualsiasi classe. Possono accedere a una nuova proprietà metadati sull'oggetto contesto, che può fungere da chiave sia per le primitive che per gli oggetti. Le informazioni sui metadati sono accessibili sulla classe tramite `Symbol.metadata`.

I metadati possono essere utilizzati per vari scopi, come il debug, la serializzazione o l'iniezione di dipendenze con i decoratori.

```
//@ts-ignore
Symbol.metadata ??= Symbol('Symbol.metadata'); // Simple polyfill

type Context =
    | ClassFieldDecoratorContext
    | ClassAccessorDecoratorContext
    | ClassMethodDecoratorContext; // Il contesto contiene la
proprietà metadati: DecoratorMetadata

function setMetadata(_target: any, context: Context) {
    // Imposta l'oggetto metadati con un valore primitivo
    context.metadata[context.name] = true;
}

class MyClass {
    @setMetadata
    a = 123;

    @setMetadata
    accessor b = 'b';

    @setMetadata
    fn() {}
}
```

```
const metadata = MyClass[Symbol.metadata]; // Ottieni informazioni
sui metadati

console.log(JSON.stringify(metadata)); //
{"bar":true,"baz":true,"foo":true}
```

Ereditarietà

L'ereditarietà si riferisce al meccanismo mediante il quale una classe può ereditare proprietà e metodi da un'altra classe, nota come classe base o superclasse. La classe derivata, chiamata anche classe figlia o sottoclasse, può estendere e specializzare le funzionalità della classe base aggiungendo nuove proprietà e metodi o sovrascrivendo quelli esistenti.

```
class Animal {
    name: string;

    constructor(name: string) {
        this.name = name;
    }

    speak(): void {
        console.log("L'animale emette un suono");
    }
}

class Dog extends Animal {
    breed: string;

    constructor(name: string, breed: string) {
        super(name);
        this.breed = breed;
    }

    speak(): void {
        console.log('Woof! Woof!');
    }
}
```

```
// Crea un'istanza della classe base
const animal = new Animal('Animale generico');
animal.speak(); // L'animale emette un suono

// Crea un'istanza della classe derivata
const dog = new Dog('Max', 'Labrador');
dog.speak(); // Woof! Bau!"
```

TypeScript non supporta l'ereditarietà multipla nel senso tradizionale, ma consente invece l'ereditarietà da una singola classe base. TypeScript supporta più interfacce. Un'interfaccia può definire un contratto per la struttura di un oggetto e una classe può implementare più interfacce. Questo consente a una classe di ereditare comportamento e struttura da più fonti.

```
interface Flyable {
    fly(): void;
}

interface Swimmable {
    swim(): void;
}

class FlyingFish implements Flyable, Swimmable {
    fly() {
        console.log('Flying...');
    }

    swim() {
        console.log('Swimming...');
    }
}

const flyingFish = new FlyingFish();
flyingFish.fly();
flyingFish.swim();
```

La parola chiave `class` in TypeScript, simile a JavaScript, è spesso definita "syntactic sugar". È stata introdotta in ECMAScript 2015 (ES6) offre una sintassi più familiare per la creazione e l'utilizzo di oggetti in modalità basata sulle classi. Tuttavia, è importante notare

che TypeScript, essendo un superset di JavaScript, alla fine si compila in JavaScript, che rimane fondamentalmente basato sui prototipi.

Statiche

TypeScript ha membri statici. Per accedere ai membri statici di una classe, è possibile utilizzare il nome della classe seguito da un punto, senza dover creare un oggetto.

```
class OfficeWorker {
    static memberCount: number = 0;

    constructor(private name: string) {
        OfficeWorker.memberCount++;
    }
}

const w1 = new OfficeWorker('James');
const w2 = new OfficeWorker('Simon');
const total = OfficeWorker.memberCount;
console.log(total); // 2
```

Inizializzazione delle proprietà

Esistono diversi modi per Inizializza le proprietà per una classe in TypeScript:

Inline:

Nell'esempio seguente, questi valori iniziali verranno utilizzati quando verrà creata un'istanza della classe.

```
class MyClass {
    property1: string = 'default value';
    property2: number = 42;
}
```

Nel costruttore:

```
class MyClass {
    property1: string;
    property2: number;

    constructor() {
        this.property1 = 'default value';
        this.property2 = 42;
    }
}
```

Utilizzo dei parametri del costruttore:

```
class MyClass {
    constructor(
        private property1: string = 'default value',
        public property2: number = 42
    ) {
        // Non è necessario assegnare esplicitamente i valori alle
proprietà.
    }
    log() {
        console.log(this.property2);
    }
}
const x = new MyClass();
x.log();
```

Sovraccarico dei metodi

Il sovraccarico dei metodi consente a una classe di avere più metodi con lo stesso nome ma tipi di parametri diversi o un numero diverso di parametri. Questo ci permette di chiamare un metodo in modi diversi in base agli argomenti passati.

```
class MyClass {
    add(a: number, b: number): number; // Firma di sovraccarico 1
    add(a: string, b: string): string; // Firma di sovraccarico 2

    add(a: number | string, b: number | string): number | string {
```

```
        if (typeof a === 'number' && typeof b === 'number') {
            return a + b;
        }
        if (typeof a === 'string' && typeof b === 'string') {
            return a.concat(b);
        }
        throw new Error('Argomenti non validi');
    }
}

const r = new MyClass();
console.log(r.add(10, 5)); // Log 15
```

Generici

I generici consentono di creare componenti e funzioni riutilizzabili che possono funzionare con più tipi. Con i generici, è possibile parametrizzare tipi, funzioni e interfacce, consentendo loro di operare su tipi diversi senza doverli specificare esplicitamente in anticipo.

I generici consentono di rendere il codice più flessibile e riutilizzabile.

Tipo generico

Per definire un tipo generico, si utilizzano le parentesi angolari (<>) per specificare i parametri di tipo, ad esempio:

```
function identity<T>(arg: T): T {
    return arg;
}
const a = identity('x');
const b = identity(123);

const getLen = <T,>(data: ReadonlyArray<T>) => data.length;
const len = getLen([1, 2, 3]);
```

Classi generiche

I generici possono essere applicati anche alle classi, in questo modo possono lavorare con più tipi utilizzando parametri di tipo. Questo è utile per creare definizioni di classe riutilizzabili che possono operare su diversi tipi di dati mantenendo la sicurezza dei tipi.

```
class Container<T> {
    private item: T;

    constructor(item: T) {
        this.item = item;
    }

    getItem(): T {
        return this.item;
    }
}

const numberContainer = new Container<number>(123);
console.log(numberContainer.getItem()); // 123

const stringContainer = new Container<string>('hello');
console.log(stringContainer.getItem()); // ciao
```

Vincoli generici

I parametri generici possono essere vincolati utilizzando la parola chiave `extends` seguita da un tipo o un'interfaccia che il parametro di tipo deve soddisfare.

Nell'esempio seguente, T deve contenere una `length` appropriata per essere valido:

```
const printLen = <T extends { length: number }>(value: T): void =>
{
    console.log(value.length);
};
```

```
printLen('Ciao'); // 5
printLen([1, 2, 3]); // 3
printLen({ length: 10 }); // 10
printLen(123); // Non valido
```

Una caratteristica interessante di generic introdotta nella versione 3.4 RC è l'inferenza di tipo di funzione di ordine superiore, che ha introdotto argomenti di tipo generico propagati:

```
declare function pipe<A extends any[], B, C>(
    ab: (...args: A) => B,
    bc: (b: B) => C
): (...args: A) => C;

declare function list<T>(a: T): T[];
declare function box<V>(x: V): { value: V };

const listBox = pipe(list, box); // <T>(a: T) => { value: T[] }
const boxList = pipe(box, list); // <V>(x: V) => { value: V }[]
```

Questa funzionalità consente una programmazione più semplice, sicura e senza punti, comune nella programmazione funzionale.

Restringimento contestuale generico

Il restringimento contestuale per i generici è il meccanismo di TypeScript che consente al compilatore di restringere il tipo di un parametro generico in base al contesto in cui viene utilizzato. È utile quando si lavora con tipi generici in istruzioni condizionali:

```
function process<T>(value: T): void {
    if (typeof value === 'string') {
        // Il valore viene ristretto al tipo 'string'
        console.log(value.length);
    } else if (typeof value === 'number') {
        // Il valore viene ristretto al tipo 'number'
        console.log(value.toFixed(2));
    }
}
```

```
process('hello'); // 5
process(3.14159); // 3.14
```

Tipi strutturali cancellati

In TypeScript, gli oggetti non devono necessariamente corrispondere a un tipo specifico ed esatto. Ad esempio, se creiamo un oggetto che soddisfa i requisiti di un'interfaccia, possiamo utilizzare quell'oggetto nei punti in cui l'interfaccia è richiesta, anche se non esiste una connessione esplicita tra i due. Esempio:

```
type NameProp1 = {
    prop1: string;
};

function log(x: NameProp1) {
    console.log(x.prop1);
}

const obj = {
    prop2: 123,
    prop1: 'Origin',
};

log(obj); // Valido
```

Namespace

In TypeScript, gli spazi dei nomi vengono utilizzati per organizzare il codice in contenitori logici, prevenendo collisioni di nomi e fornendo un modo per raggruppare il codice correlato. L'utilizzo delle parole chiave `export` consente l'accesso allo spazio dei nomi nei moduli "esterni".

```
export namespace MyNamespace {
    export interface MyInterface1 {
        prop1: boolean;
```

```
    }
    export interface MyInterface2 {
        prop2: string;
    }
}

const a: MyNamespace.MyInterface1 = {
    prop1: true,
};
```

Simboli

I simboli sono un tipo di dati primitivo che rappresenta un valore immutabile la cui unicità globale è garantita per tutta la durata del programma.

I simboli possono essere utilizzati come chiavi per le proprietà degli oggetti e forniscono un modo per creare proprietà non enumerabili.

```
const key1: symbol = Symbol('key1');
const key2: symbol = Symbol('key2');

const obj = {
    [key1]: 'value 1',
    [key2]: 'value 2',
};

console.log(obj[key1]); // valore 1
console.log(obj[key2]); // valore 2
```

In WeakMaps e WeakSet, i simboli sono ora consentiti come chiavi.

Direttive con tripla barra

Le direttive con tripla barra sono commenti speciali che forniscono istruzioni al compilatore su come elaborare un file. Queste direttive iniziano con tre barre consecutive (`///`) e sono in genere posizionate

all'inizio di un file TypeScript e non hanno alcun effetto sul comportamento in fase di esecuzione.

Le direttive con tripla barra vengono utilizzate per fare riferimento a dipendenze esterne, specificare il comportamento di caricamento dei moduli, abilitare/disabilitare determinate funzionalità del compilatore e altro ancora. Alcuni esempi:

Riferimento a un file di dichiarazione:

```
/// <reference path="path/to/declaration/file.d.ts" />
```

Indicare il formato del modulo:

```
/// <amd|commonjs|system|umd|es6|es2015|none>
```

Abilitare le opzioni del compilatore, nell'esempio seguente, in modalità strict:

```
/// <strict|noImplicitAny|noUnusedLocals|noUnusedParameters>
```

Manipolazione dei tipi

Creazione di tipi da tipi

È possibile creare nuovi tipi componendo, manipolando o trasformando tipi esistenti.

Tipi di intersezione (`&`):

Consentono di combinare più tipi in un unico tipo:

```
type A = { foo: number };
type B = { bar: string };
type C = A & B; // Intersezione di A e B
const obj: C = { foo: 42, bar: 'hello' };
```

Tipi di unione (`|`):

Consente di definire un tipo che può essere di diversi tipi:

```
type Result = string | number;
const value1: Result = 'hello';
const value2: Result = 42;
```

Tipi mappati:

Consentono di trasformare le proprietà di un tipo esistente per crearne uno nuovo:

```
type Mutable<T> = {
    readonly [P in keyof T]: T[P];
};
type Person = {
    name: string;
    age: number;
};
type ImmutablePerson = Mutable<Person>; // Le proprietà diventano
di sola lettura
```

Tipi condizionali:

Consentono di creare tipi in base ad alcune condizioni:

```
type ExtractParam<T> = T extends (param: infer P) => any ? P :
never;
type MyFunction = (name: string) => number;
type ParamType = ExtractParam<MyFunction>; // string
```

Tipi di accesso indicizzati

In TypeScript è possibile accedere e manipolare i tipi di proprietà all'interno di un altro tipo utilizzando un indice, `Type[Key]`.

```
type Person = {
    name: string;
    age: number;
```

```
};

type AgeType = Person['age']; // number

type MyTuple = [string, number, boolean];
type MyType = MyTuple[2]; // boolean
```

Tipi di utilità

Diversi tipi di utilità predefiniti possono essere utilizzati per manipolare i tipi; di seguito è riportato un elenco dei più comuni:

Awaited<T>

Costruisce un tipo che esegue ricorsivamente l'unwrapping dei tipi Promise.

```
type A = Awaited<Promise<string>>; // string
```

Partial<T>

Costruisce un tipo con tutte le proprietà di T impostate su optional.

```
type Person = {
    name: string;
    age: number;
};

type A = Partial<Person>; // { name?: string | undefined;
age?:number | undefined; }
```

Required<T>

Costruisce un tipo con tutte le proprietà di T impostate su required.

```
type Person = {
    name?: string;
    age?: number;
```

```
};

type A = Required<Person>; // { name: string; age:number; }
```

Readonly<T>

Costruisce un tipo con tutte le proprietà di T impostate su readonly.

```
type Person = {
    name: string;
    age: number;
};

type A = Readonly<Person>;

const a: A = { name: 'Simon', age: 17 };
a.name = 'John'; // Non valido
```

Record<K, T>

Costruisce un tipo con un insieme di proprietà K di tipo T.

```
type Product = {
    name: string;
    price: number;
};

const products: Record<string, Product> = {
    apple: { name: 'Apple', price: 0.5 },
    banana: { name: 'Banana', price: 0.25 },
};

console.log(products.apple); // { name: 'Apple', price: 0.5 }
```

Pick<T, K>

Costruisce un tipo selezionando le proprietà specificate K da T.

```
type Product = {
    name: string;
```

```
    price: number;
};

type Price = Pick<Product, 'price'>; // { price: number; }
```

Omit<T, K>

Costruisce un tipo omettendo le proprietà specificate K da T.

```
type Product = {
    name: string;
    price: number;
};

type Name = Omit<Product, 'price'>; // { name: string; }
```

Exclude<T, U>

Costruisce un tipo escludendo tutti i valori di tipo U da T.

```
type Union = 'a' | 'b' | 'c';
type MyType = Exclude<Union, 'a' | 'c'>; // b
```

Extract<T, U>

Costruisce un tipo estraendo tutti i valori di tipo U da T.

```
type Union = 'a' | 'b' | 'c';
type MyType = Extract<Union, 'a' | 'c'>; // a | c
```

NonNullable<T>

Costruisce un tipo escludendo null e undefined da T.

```
type Union = 'a' | null | undefined | 'b';
type MyType = NonNullable<Union>; // 'a' | 'b'
```

Parameters<T>

Estrae i tipi di parametro di una funzione di tipo T.

```
type Func = (a: string, b: number) => void;
type MyType = Parameters<Func>; // [a: string, b: number]
```

ConstructorParameters<T>

Estrae i tipi di parametro di una funzione costruttore di tipo T.

```
class Person {
    constructor(
        public name: string,
        public age: number
    ) {}
}
type PersonConstructorParams = ConstructorParameters<typeof
Person>; // [name: string, age: number]
const params: PersonConstructorParams = ['John', 30];
const person = new Person(...params);
console.log(person); // Person { name: 'John', age: 30 }
```

ReturnType<T>

Estrae il tipo di ritorno di una funzione di tipo T.

```
type Func = (name: string) => number;
type MyType = ReturnType<Func>; // number
```

InstanceType<T>

Estrae il tipo di istanza di una classe di tipo T.

```
class Person {
    name: string;

    constructor(name: string) {
        this.name = name;
    }
```

```
    sayHello() {
        console.log(`Ciao, mi chiamo ${this.name}!`);
    }
}

type PersonInstance = InstanceType<typeof Person>;

const person: PersonInstance = new Person('John');

person.sayHello(); // Ciao, mi chiamo John!
```

ThisParameterType<T>

Estrae il tipo del parametro 'this' da una funzione di tipo T.

```
interface Person {
    name: string;
    greet(this: Person): void;
}
type PersonThisType = ThisParameterType<Person['greet']>; // Person
```

OmitThisParameter<T>

Rimuove il parametro 'this' da una funzione di tipo T.

```
function capitalize(this: String) {
    return this[0].toUpperCase + this.substring(1).toLowerCase();
}

type CapitalizeType = OmitThisParameter<typeof capitalize>; // ()
=> string
```

ThisType<T>

Funge da marcatore per un tipo `this` contestuale.

```
type Logger = {
    log: (error: string) => void;
};
```

```
let helperFunctions: { [name: string]: Function } &
ThisType<Logger> = {
    hello: function () {
        this.log('some error'); // Valido poiché "log" è parte di
"this".
        this.update(); // Non valido
    },
};
```

Uppercase<T>

Rendi maiuscolo il nome del tipo di input T.

```
type MyType = Uppercase<'abc'>; // "ABC"
```

Lowercase<T>

Rendi minuscolo il nome del tipo di input T.

```
type MyType = Lowercase<'ABC'>; // "abc"
```

Capitalize<T>

Inserisci in maiuscolo il nome del tipo di input T.

```
type MyType = Capitalize<'abc'>; // "Abc"
```

Uncapitalize<T>

Inserisci in maiuscolo il nome del tipo di input T.

```
type MyType = Uncapitalize<'Abc'>; // "abc"
```

NoInfer<T>

NoInfer è un tipo di utilità progettato per bloccare l'inferenza automatica dei tipi nell'ambito di una funzione generica.

Esempio:

```
// Inferenza automatica dei tipi nell'ambito di una funzione
generica.
function fn<T extends string>(x: T[], y: T) {
    return x.concat(y);
}
const r = fn(['a', 'b'], 'c'); // Il tipo qui è ("a" | "b" | "c")[]
```

Con NoInfer:

```
// Funzione di esempio che utilizza NoInfer per impedire
l'inferenza di tipo
function fn2<T extends string>(x: T[], y: NoInfer<T>) {
    return x.concat(y);
}

const r2 = fn2(['a', 'b'], 'c'); // Errore: l'argomento di tipo
'"c"' non è assegnabile al parametro di tipo '"a" | "b"'.
```

Altri

Gestione degli errori e delle eccezioni

TypeScript consente di rilevare e gestire gli errori utilizzando i meccanismi standard di gestione degli errori JavaScript:

Blocchi Try-Catch-Finally:

```
try {
    // Codice che potrebbe generare un errore
} catch (error) {
    // Gestisci l'errore
} finally {
    // Codice che viene sempre eseguito, finally è facoltativo
}
```

È anche possibile gestire diversi tipi di errore:

```
try {
    // Codice che potrebbe generare diversi tipi di errore
} catch (error) {
    if (error instanceof TypeError) {
        // Gestisci TypeError
    } else if (error instanceof RangeError) {
        // Gestisci RangeError
    } else {
        // Gestisci altri errori
    }
}
```

Tipi di errore personalizzati:

È possibile specificare errori più specifici estendendo la classe `Error`:

```
class CustomError extends Error {
    constructor(message: string) {
        super(message);
        this.name = 'CustomError';
    }
}

throw new CustomError('Questo è un errore personalizzato.');
```

Classi Mixin

Le classi Mixin consentono di combinare e comporre il comportamento di più classi in un'unica classe. Forniscono un modo per riutilizzare ed estendere le funzionalità senza la necessità di catene di ereditarietà profonde.

```
abstract class Identificabile {
    name: string = '';

    logId() {
        console.log('id:', this.name);
    }
}
```

```
abstract class Selezionabile {
    selected: boolean = false;

    select() {
        this.selected = true;
        console.log('Select');
    }

    deselect() {
        this.selected = false;
        console.log('Deselect');
    }
}

class MyClass {
    constructor() {}
}

// Estendi MyClass per includere il comportamento di Identificabile
e Selezionabile
interface MyClass extends Identificabile, Selezionabile {}

// Funzione per applicare i mixin a una classe
function applyMixins(source: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name
=> {
            const descriptor = Object.getOwnPropertyDescriptor(
                baseCtor.prototype,
                name
            );
            if (descriptor) {
                Object.defineProperty(source.prototype, name,
descriptor);
            }
        });
    });
}

// Applica i mixin a MyClass
applyMixins(MyClass, [Identificabile, Selezionabile]);

let o = new MyClass();
```

```
o.name = 'abc';
o.logId();
o.select();
```

Funzionalità del linguaggio asincrono

Essendo TypeScript un superset di JavaScript, integra funzionalità del linguaggio asincrono come:

Promise:

Le promise sono un modo per gestire le operazioni asincrone e i loro risultati utilizzando metodi come `.then()` e `.catch()` per gestire le condizioni di successo e di errore.

Per saperne di più: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Async/await:

Le parole chiave Async/await sono un modo per fornire una sintassi più sincrona per lavorare con le promise. La parola chiave `async` viene utilizzata per definire una funzione asincrona, mentre la parola chiave `await` viene utilizzata all'interno di una funzione asincrona per mettere in pausa l'esecuzione finché una Promise non viene risolta o rifiutata.

Per saperne di più: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await

Le seguenti API sono ben supportate in TypeScript:

API Fetch:

https://developer.mozilla.org/it/docs/Web/API/Fetch_API

Web Worker: https://developer.mozilla.org/it/docs/Web/API/Web_Workers_API

Condiviso Worker: https://developer.mozilla.org/en-US/docs/Web/API/SharedWorker

WebSocket: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

Iteratori e Generatori

Sia gli Iteratori che i Generatori sono ben supportati in TypeScript.

Gli Iteratori sono oggetti che implementano il protocollo Iterator, fornendo un modo per accedere agli elementi di una collezione o sequenza uno alla volta. Si tratta di una struttura che contiene un puntatore all'elemento successivo nell'iterazione. Hanno un metodo `next()` che restituisce il valore successivo nella sequenza insieme a un valore booleano che indica se la sequenza è `completata`.

```
class NumberIterator implements Iterable<number> {
    private current: number;

    constructor(
        private start: number,
        private end: number
    ) {
        this.current = start;
    }

    public next(): IteratorResult<number> {
        if (this.current <= this.end) {
            const value = this.current;
            this.current++;
            return { value, done: false };
        } else {
            return { value: undefined, done: true };
        }
    }
```

```
    [Symbol.iterator](): Iterator<number> {
        return this;
    }
}

const iterator = new NumberIterator(1, 3);

for (const num of iterator) {
    console.log(num);
}
```

I generatori sono funzioni speciali definite utilizzando la sintassi `function*` che semplifica la creazione di iteratori. Utilizzano la parola chiave `yield` per definire la sequenza di valori e mettono automaticamente in pausa e riprendono l'esecuzione quando vengono richiesti valori.

I generatori semplificano la creazione di iteratori e sono particolarmente utili per lavorare con sequenze di grandi dimensioni o infinite.

Esempio:

```
function* numberGenerator(start: number, end: number):
Generator<number> {
    for (let i = start; i <= end; i++) {
        yield i;
    }
}

const generator = numberGenerator(1, 5);

for (const num of generator) {
    console.log(num);
}
```

TypeScript supporta anche iteratori e generatori asincroni.

Per saperne di più:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Iterator

Riferimento JSDoc di TsDocs

Quando si lavora con una base di codice JavaScript, è possibile aiutare TypeScript a dedurre il tipo corretto utilizzando commenti JSDoc con annotazioni aggiuntive per fornire informazioni sul tipo.

Esempio:

```
/**
 * Calcola la potenza di un numero dato
 * @constructor
 * @param {number} base – Il valore base dell'espressione
 * @param {number} exponent – Il valore esponente dell'espressione
 */
function power(base: number, exponent: number) {
    return Math.pow(base, exponent);
}
power(10, 2); // function power(base: number, exponent: number): number
```

La documentazione completa è disponibile a questo link:

https://www.typescriptlang.org/docs/handbook/jsdoc-supported-types.html

Dalla versione 3.7 è possibile generare definizioni di tipo .d.ts dalla sintassi JavaScript JSDoc. Ulteriori informazioni sono disponibili qui: https://www.typescriptlang.org/docs/handbook/declaration-files/dts-from-js.html

@types

I pacchetti nell'organizzazione @types sono convenzioni di denominazione speciali utilizzate per fornire definizioni di tipo per librerie o moduli JavaScript esistenti. Ad esempio, usando:

```
npm install --save-dev @types/lodash
```

Installerà le definizioni di tipo di `lodash` nel tuo progetto corrente.

Per contribuire alle definizioni di tipo del pacchetto @types, invia una pull request a https://github.com/DefinitelyTyped/DefinitelyTyped.

JSX

JSX (JavaScript XML) è un'estensione della sintassi del linguaggio JavaScript che consente di scrivere codice simile a HTML all'interno dei file JavaScript o TypeScript. Viene comunemente utilizzato in React per definire la struttura HTML.

TypeScript extends le funzionalità di JSX fornendo il controllo dei tipi e l'analisi statica.

Per utilizzare JSX è necessario impostare l'opzione del compilatore `jsx` nel file `tsconfig.json`. Due opzioni di configurazione comuni:

- "preserve": emette file .jsx con il JSX invariato. Questa opzione indica a TypeScript di mantenere la sintassi JSX così com'è e di non trasformarla durante il processo di compilazione. È possibile utilizzare questa opzione se si dispone di uno strumento separato, come Babel, che gestisce la trasformazione.
- "react": abilita la trasformazione JSX integrata di TypeScript. Verrà utilizzato React.createElement.

Tutte le opzioni sono disponibili qui:
https://www.typescriptlang.org/tsconfig#jsx

Moduli ES6

TypeScript supporta ES6 (ECMAScript 2015) e molte versioni successive. Ciò significa che è possibile utilizzare la sintassi ES6, come funzioni freccia, letterali template, classi, moduli, destrutturazione e altro ancora.

Per abilitare le funzionalità ES6 nel progetto, è possibile specificare la proprietà `target` nel file tsconfig.json.

Un esempio di configurazione:

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "es6",
    "moduleResolution": "node",
    "sourceMap": true,
    "outDir": "dist"
  },
  "include": ["src"]
}
```

Operatore di elevamento a potenza ES7

L'operatore di elevamento a potenza (**) calcola il valore ottenuto elevando il primo operando alla potenza del secondo operando. Funziona in modo simile a `Math.pow()`, ma con la possibilità aggiuntiva di accettare BigInt come operandi. TypeScript supporta pienamente questo operatore, utilizzandolo come `target` nel file tsconfig.json `es2016` o versione successiva.

```
console.log(2 ** (2 ** 2)); // 16
```

L’istruzione for-await-of

Questa è una funzionalità JavaScript completamente supportata in TypeScript che consente di iterare su oggetti iterabili asincroni dalla versione target es2018.

```
async function* asyncNumbers(): AsyncIterableIterator<number> {
    yield Promise.resolve(1);
    yield Promise.resolve(2);
    yield Promise.resolve(3);
}

(async () => {
    for await (const num of asyncNumbers()) {
        console.log(num);
    }
})();
```

Nuova meta-proprietà target

In TypeScript è possibile utilizzare la meta-proprietà `new.target`, che consente di determinare se una funzione o un costruttore è stato invocato utilizzando l’operatore new. Permette inoltre di rilevare se un oggetto è stato creato a seguito di una chiamata al costruttore.

```
class Parent {
    constructor() {
        console.log(new.target); // Registra la funzione
costruttore utilizzata per creare un'istanza
    }
}

class Child extends Parent {
    constructor() {
        super();
    }
}

const parentX = new Parent(); // [Function: Parent]
const child = new Child(); // [Function: Child]
```

Espressioni di importazione dinamica

È possibile caricare i moduli in modo condizionale o caricarli in modo differito su richiesta utilizzando la proposta ECMAScript per l'importazione dinamica, supportata in TypeScript.

La sintassi per le espressioni di importazione dinamica in TypeScript è la seguente:

```
async function renderWidget() {
    const container = document.getElementById('widget');
    if (container !== null) {
        const widget = await import('./widget'); // Importazione
dinamica
        widget.render(container);
    }
}

renderWidget();
```

"tsc –watch"

Questo comando avvia un compilatore TypeScript con il parametro –watch, con la possibilità di ricompilare automaticamente i file TypeScript ogni volta che vengono modificati.

```
tsc --watch
```

A partire dalla versione 4.9 di TypeScript, il monitoraggio dei file si basa principalmente sugli eventi del file system, ricorrendo automaticamente al polling se non è possibile stabilire un watcher basato sugli eventi.

Operatore di asserzione non nullo

L'operatore di asserzione non nullo (Postfix !), noto anche come asserzione di assegnazione definita, è una funzionalità di TypeScript che consente di affermare che una variabile o una proprietà non è nulla o indefinita, anche se l'analisi statica dei tipi di TypeScript suggerisce che potrebbe esserlo. Con questa funzionalità è possibile rimuovere qualsiasi controllo esplicito.

```
type Person = {
    name: string;
};

const printName = (person?: Person) => {
    console.log(`Name is ${person!.name}`);
};
```

Dichiarazioni predefinite

Le dichiarazioni predefinite vengono utilizzate quando a una variabile o a un parametro viene assegnato un valore predefinito. Ciò significa che se non viene fornito alcun valore per quella variabile o parametro, verrà utilizzato il valore predefinito.

```
function greet(name: string = 'Anonymous'): void {
    console.log(`Ciao, ${name}!`);
}
greet(); // Ciao, Anonymous!
greet('John'); // Ciao, John!
```

Concatenamento opzionale

L'operatore di concatenamento opzionale `?.` funziona come il normale operatore punto (`.`) per accedere a proprietà o metodi. Tuttavia, gestisce in modo elegante i valori nulli o indefiniti

terminando l'espressione e restituendo `undefined`, invece di generare un errore.

```
type Person = {
    name: string;
    age?: number;
    address?: {
        street?: string;
        city?: string;
    };
};

const person: Person = {
    name: 'John',
};

console.log(person.address?.city); // indefinito
```

Operatore di coalescenza nullo

L'operatore di coalescenza nullo `??` restituisce il valore del lato destro se il lato sinistro è `null` o `undefined`; in caso contrario, restituisce il valore del lato sinistro.

```
const foo = null ?? 'foo';
console.log(foo); // foo

const baz = 1 ?? 'baz';
const baz2 = 0 ?? 'baz';
console.log(baz); // 1
console.log(baz2); // 0
```

Tipi letterali di template

I tipi letterali modello consentono di manipolare i valori stringa a livello di tipo e di generare nuovi tipi stringa basati su quelli esistenti. Sono utili per creare tipi più espressivi e precisi da operazioni basate su stringhe.

```ts
type Department = 'engineering' | 'hr';
type Language = 'english' | 'spanish';
type Id = `${Department}-${Language}-id`; // "engineering-english-id" | "engineering-spanish-id" | "hr-english-id" | "hr-spanish-id"
```

Sovraccarico di funzioni

Il sovraccarico di funzioni consente di definire più firme di funzione per lo stesso nome di funzione, ciascuna con tipi di parametro e tipo di ritorno diversi. Quando si chiama una funzione sovraccaricata, TypeScript utilizza gli argomenti forniti per determinare la firma di funzione corretta:

```ts
function makeGreeting(name: string): string;
function makeGreeting(names: string[]): string[];

function makeGreeting(person: unknown): unknown {
    if (typeof person === 'string') {
        return `Ciao ${person}!`;
    } else if (Array.isArray(person)) {
        return person.map(name => `Ciao, ${name}!`);
    }
    throw new Error('Impossibile salutare');
}

makeGreeting('Simon');
makeGreeting(['Simone', 'John']);
```

Tipi ricorsivi

Un tipo ricorsivo è un tipo che può fare riferimento a se stesso. Questo è utile per definire strutture dati che hanno una struttura gerarchica o ricorsiva (annidamento potenzialmente infinito), come liste concatenate, alberi e grafi.

```ts
type ListNode<T> = {
    data: T;
```

```
    next: ListNode<T> | undefined;
};
```

Tipi condizionali ricorsivi

È possibile definire relazioni di tipo complesse utilizzando la logica e la ricorsione in TypeScript. Analizziamole in termini semplici:

Tipi condizionali: consente di definire tipi in base a condizioni booleane:

```
type CheckNumber<T> = T extends number ? 'Number' : 'Not a number';
type A = CheckNumber<123>; // 'Number'
type B = CheckNumber<'abc'>; // 'Not a number'
```

Ricorsione: indica una definizione di tipo che fa riferimento a se stessa all'interno della propria definizione:

```
type Json = string | number | boolean | null | Json[] | { [key:
string]: Json };

const data: Json = {
    prop1: true,
    prop2: 'prop2',
    prop3: {
        prop4: [],
    },
};
```

I tipi condizionali ricorsivi combinano sia la logica condizionale che la ricorsione. Ciò significa che una definizione di tipo può dipendere da se stessa tramite la logica condizionale, creando relazioni di tipo complesse e flessibili.

```
type Flatten<T> = T extends Array<infer U> ? Flatten<U> : T;

type NestedArray = [1, [2, [3, 4], 5], 6];
type FlattenedArray = Flatten<NestedArray>; // 2 | 3 | 4 | 5 | 1 |
6
```

Supporto per i moduli ECMAScript in Node

Node.js ha aggiunto il supporto per i moduli ECMAScript a partire dalla versione 15.3.0, mentre TypeScript supporta i moduli ECMAScript per Node.js dalla versione 4.7. Questo supporto può essere abilitato utilizzando la proprietà `module` con il valore `nodenext` nel file tsconfig.json. Ecco un esempio:

```
{
  "compilerOptions": {
    "module": "nodenext",
    "outDir": "./lib",
    "declaration": true
  }
}
```

Node.js supporta due estensioni di file per i moduli: `.mjs` per i moduli ES e `.cjs` per i moduli CommonJS. Le estensioni di file equivalenti in TypeScript sono `.mts` per i moduli ES e `.cts` per i moduli CommonJS. Quando il compilatore TypeScript trascrive questi file in JavaScript, creerà i file `.mjs` e `.cjs`.

Se desideri utilizzare moduli ES nel tuo progetto, puoi impostare la proprietà `type` su “module” nel file package.json. Questo indica a Node.js di trattare il progetto come un progetto di modulo ES.

Inoltre, TypeScript supporta anche le dichiarazioni di tipo nei file .d.ts. Questi file di dichiarazione forniscono informazioni sul tipo per librerie o moduli scritti in TypeScript, consentendo ad altri sviluppatori di utilizzarli con le funzionalità di controllo del tipo e di completamento automatico di TypeScript.

Funzioni di asserzione

In TypeScript, le funzioni di asserzione sono funzioni che indicano la verifica di una condizione specifica in base al loro valore di ritorno. Nella loro forma più semplice, una funzione di asserzione esamina un predicato fornito e genera un errore quando il predicato restituisce false.

```
function isNumber(value: unknown): asserts value is number {
    if (typeof value !== 'number') {
        throw new Error('Not a number');
    }
}
```

Oppure può essere dichiarato come espressione di funzione:

```
type AssertIsNumber = (value: unknown) => asserts value is number;
const isNumber: AssertIsNumber = value => {
    if (typeof value !== 'number') {
        throw new Error('Not a number');
    }
};
```

Le funzioni di asserzione condividono alcune somiglianze con le type guard. Le type guard sono state inizialmente introdotte per eseguire controlli in fase di esecuzione e garantire il tipo di un valore all'interno di un ambito specifico. Nello specifico, una type guard è una funzione che valuta un predicato di tipo e restituisce un valore booleano che indica se il predicato è vero o falso. Questo differisce leggermente dalle funzioni di asserzione, in cui l'intenzione è quella di generare un errore anziché restituire false quando il predicato non è soddisfatto.

Esempio di type guard:

```
const isNumber = (value: unknown): value is number => typeof value
=== 'number';
```

Tipi di tupla variadici

I tipi di tupla variadici sono una funzionalità introdotta nella versione 4.0 di TypeScript. Iniziamo a conoscerli ripassando cos'è una tupla:

Un tipo di tupla è un array di lunghezza definita, di cui è noto il tipo di ogni elemento:

```
type Student = [string, number];
const [name, age]: Student = ['Simone', 20];
```

Il termine "variadico" significa indefinito (accetta un numero variabile di argomenti).

Una tupla variadica è un tipo di tupla che ha tutte le proprietà di prima, ma la forma esatta non è ancora definita:

```
type Bar<T extends unknown[]> = [boolean, ...T, number];

type A = Bar<[boolean]>; // [booleano, booleano, numero]
type B = Bar<['a', 'b']>; // [boolean, 'a', 'b', number]
type C = Bar<[]>; // [boolean, number]
```

Nel codice precedente possiamo vedere che la forma della tupla è definita dal generico `T` passato.

Le tuple variadiche possono accettare più generici, il che le rende molto flessibili:

```
type Bar<T extends unknown[], G extends unknown[]> = [...T,
boolean, ...G];

type A = Bar<[number], [string]>; // [number, boolean, string]
type B = Bar<['a', 'b'], [boolean]>; // ["a", "b", boolean,
boolean]
```

Con le nuove tuple variadiche possiamo usare:

- Gli spread nella sintassi dei tipi di tupla ora possono essere generici, quindi possiamo rappresentare operazioni di ordine superiore su tuple e array anche quando non conosciamo i tipi effettivi su cui stiamo operando.
- Gli elementi rimanenti possono trovarsi ovunque in una tupla.

Esempio:

```
type Items = readonly unknown[];

function concat<T extends Items, U extends Items>(
    arr1: T,
    arr2: U
): [...T, ...U] {
    return [...arr1, ...arr2];
}

concat([1, 2, 3], ['4', '5', '6']); // [1, 2, 3, "4", "5", "6"]
```

Tipi boxed

I tipi boxed si riferiscono agli oggetti wrapper utilizzati per rappresentare i tipi primitivi come oggetti. Questi oggetti wrapper forniscono funzionalità e metodi aggiuntivi che non sono disponibili direttamente sui valori primitivi.

Quando si accede a un metodo come `charAt` o `normalize` su una primitiva `string`, JavaScript lo racchiude in un oggetto `String`, chiama il metodo e quindi elimina l'oggetto.

Dimostrazione:

```
const originalNormalize = String.prototype.normalize;
String.prototype.normalize = function () {
    console.log(this, typeof this);
    return originalNormalize.call(this);
};
console.log('\u0041'.normalize());
```

TypeScript rappresenta questa differenziazione fornendo tipi separati per le primitive e i corrispondenti wrapper di oggetti:

- string => String
- number => Number
- boolean => Boolean
- symbol => Symbol
- bigint => BigInt

I tipi boxed di solito non sono necessari. Evitare di utilizzare tipi boxed e utilizzare invece type per le primitive, ad esempio `string` invece di `String`.

Covarianza e Controvarianza in TypeScript

Covarianza e Controvarianza vengono utilizzate per descrivere il funzionamento delle relazioni quando si ha a che fare con l'ereditarietà o l'assegnazione di tipi.

Covarianza significa che una relazione di tipo preserva la direzione dell'ereditarietà o dell'assegnazione, quindi se un tipo A è un sottotipo del tipo B, anche un array di tipo A è considerato un sottotipo di un array di tipo B. La cosa importante da notare qui è che la relazione di sottotipo viene mantenuta, il che significa che Covarianza accetta il sottotipo ma non il supertipo.

La controvarianza significa che una relazione di tipo inverte la direzione dell'ereditarietà o dell'assegnazione, quindi se un tipo A è un sottotipo del tipo B, allora un array di tipo B è considerato un sottotipo di un array di tipo A. La relazione di sottotipo è invertita, il che significa che la controvarianza accetta il supertipo ma non il sottotipo.

Note: La bivarianza significa accettare sia il supertipo che il sottotipo.

Esempio: supponiamo di avere uno spazio per tutti gli animali e uno spazio separato solo per i cani.

In covarianza, puoi inserire tutti i cani nello spazio degli animali perché i cani sono un tipo di animale. Ma non puoi inserire tutti gli animali nello spazio dei cani perché potrebbero esserci altri animali mescolati.

In controvarianza, non puoi inserire tutti gli animali nello spazio dei cani perché lo spazio degli animali potrebbe contenere anche altri animali. Tuttavia, puoi inserire tutti i cani nello spazio degli animali perché tutti i cani sono anche animali.

```
// Esempio di covarianza
class Animal {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
}

class Dog extends Animal {
    breed: string;
    constructor(name: string, breed: string) {
        super(name);
        this.breed = breed;
    }
}

let animals: Animal[] = [];
let dogs: Dog[] = [];

// La covarianza consente di assegnare l'array del sottotipo (Dog)
all'array del supertipo (Animal)
animals = dogs;
dogs = animals; // Non valido: il tipo 'Animal[]' non è assegnabile
al tipo 'Dog[]'
```

```
// Esempio di controvarianza
type Feed<in T> = (animal: T) => void;

let feedAnimal: Feed<Animal> = (animal: Animal) => {
    console.log(`Nome animale: ${animal.name}`);
};

let feedDog: Feed<Dog> = (dog: Dog) => {
    console.log(`Nome del cane: ${dog.name}, Razza: ${dog.breed}`);
};

// La controvarianza consente di assegnare la callback del
supertipo (Animal) alla callback del sottotipo (Dog)
feedDog = feedAnimal;
feedAnimal = feedDog; // Non valido: il tipo 'Feed<Dog>' non è
assegnabile al tipo 'Feed<Animal>'.
```

In TypeScript, le relazioni di tipo per gli array sono covarianti, mentre le relazioni di tipo per i parametri di funzione sono controvarianti. Ciò significa che TypeScript presenta sia covarianza che controvarianza, a seconda del contesto.

Annotazioni di varianza opzionali per i parametri di tipo

A partire da TypeScript 4.7.0, possiamo usare le parole chiave `out` e `in` per specificare l'annotazione di varianza.

Per la covarianza, usare la parola chiave `out`:

```
type AnimalCallback<out T> = () => T; // T è covariante in questo
caso
```

E per la controvarianza, usare la parola chiave `in`:

```
type AnimalCallback<in T> = (value: T) => void; // T è
controvariante in questo caso
```

Firme di indice con pattern di stringhe modello

Le firme di indice con pattern di stringhe modello ci consentono di definire firme di indice flessibili utilizzando pattern di stringhe modello. Questa funzionalità ci consente di creare oggetti che possono essere indicizzati con pattern specifici di chiavi stringa, offrendo maggiore controllo e specificità durante l'accesso e la manipolazione delle proprietà.

TypeScript dalla versione 4.4 consente firme di indice per simboli e pattern di stringhe modello.

```
const uniqueSymbol = Symbol('description');

type MyKeys = `key-${string}`;

type MyObject = {
    [uniqueSymbol]: string;
    [key: MyKeys]: number;
};

const obj: MyObject = {
    [uniqueSymbol]: 'Chiave simbolo univoca',
    'key-a': 123,
    'key-b': 456,
};

console.log(obj[uniqueSymbol]); // Chiave simbolo univoca
console.log(obj['key-a']); // 123
console.log(obj['key-b']); // 456
```

Operatore `satisfies`

L'operatore `satisfies` consente di verificare se un dato tipo soddisfa una specifica interfaccia o condizione. In altre parole, garantisce che un tipo abbia tutte le proprietà e i metodi richiesti da una specifica

interfaccia. È un modo per garantire che una variabile rientri nella definizione di un tipo. Ecco un esempio:

```
type Columns = 'name' | 'nickName' | 'attributes';

type User = Record<Columns, string | string[] | undefined>;

// Annotazione del tipo tramite `User`
const user: User = {
    name: 'Simone',
    nickName: undefined,
    attributes: ['dev', 'admin'],
};

// Nelle righe seguenti, TypeScript non sarà in grado di dedurre
correttamente
user.attributes?.map(console.log); // La proprietà 'map' non esiste
sul tipo 'string | string[]'. La proprietà 'map' non esiste sul
tipo 'string'.
user.nickName; // string | string[] | undefined

// Asserzione di tipo tramite `as`
const user2 = {
    name: 'Simon',
    nickName: undefined,
    attributes: ['dev', 'admin'],
} as User;

// Anche in questo caso, TypeScript non sarà in grado di dedurre
correttamente
user2.attributes?.map(console.log); // La proprietà 'map' non
esiste sul tipo 'string | string[]'. La proprietà 'map' non esiste
sul tipo 'string'.
user2.nickName; // string | string[] | undefined

// Utilizzando gli operatori `satisfies` ora possiamo dedurre
correttamente i tipi
const user3 = {
    name: 'Simon',
    nickName: undefined,
    attributes: ['dev', 'admin'],
} satisfies User;
```

```
user3.attributes?.map(console.log); // TypeScript deduce
correttamente: string[]
user3.nickName; // TypeScript deduce correttamente: undefined
```

Importazioni ed esportazioni solo per tipo

Le importazioni ed esportazioni solo per tipo consentono di importare o esportare tipi senza importare o esportare i valori o le funzioni associati a tali tipi. Questo può essere utile per ridurre le dimensioni del bundle.

Per utilizzare le importazioni solo per tipo, è possibile utilizzare la parola chiave `import type`.

TypeScript consente l'utilizzo di estensioni di file sia di dichiarazione che di implementazione (.ts, .mts, .cts e .tsx) nelle importazioni solo tipo, indipendentemente dalle impostazioni `allowImportingTsExtensions`.

Ad esempio:

```
import type { House } from './house.ts';
```

Sono supportati i seguenti formati:

```
import type T from './mod';
import type { A, B } from './mod';
import type * as Types from './mod';
export type { T };
export type { T } from './mod';
```

Dichiarazione using e Gestione Risorse Esplicita

Una dichiarazione `using` è un binding immutabile con ambito a blocco, simile a `const`, utilizzato per la gestione delle risorse usa e getta. Quando inizializzato con un valore, il metodo `Symbol.dispose` di quel valore viene registrato e successivamente eseguito all'uscita dall'ambito del blocco che lo racchiude.

Questo si basa sulla funzionalità di Gestione Risorse di ECMAScript, utile per eseguire attività di pulizia essenziali dopo la creazione di oggetti, come la chiusura di connessioni, l'eliminazione di file e il rilascio di memoria.

Note:

- A causa della sua recente introduzione nella versione 5.2 di TypeScript, la maggior parte dei runtime non dispone di supporto nativo. Sono necessari polyfill per: `Symbol.dispose`, `Symbol.asyncDispose`, `DisposableStack`, `AsyncDisposableStack`, `SuppressedError`. * Inoltre, dovrai configurare il tuo file tsconfig.json come segue:

```
{
  "compilerOptions": {
    "target": "es2022",
    "lib": ["es2022", "esnext.disposable", "dom"]
  }
}
```

Esempio:

```
//@ts-ignore
Symbol.dispose ??= Symbol('Symbol.dispose'); // Simple polyfill

const doWork = (): Disposable => {
    return {
        [Symbol.dispose]: () => {
```

```
            console.log('disposed');
        },
    };
};

console.log(1);

{
    using work = doWork(); // La risorsa è dichiarata
    console.log(2);
} // La risorsa viene eliminata (ad esempio, viene valutata
`work[Symbol.dispose]()`)

console.log(3);
```

Il codice registrerà:

```
1
2
disposed
3
```

Una risorsa idonea per l'eliminazione deve rispettare l'interfaccia `Disposable`:

```
// lib.esnext.disposable.d.ts
interface Disposable {
    [Symbol.dispose](): void;
}
```

Le dichiarazioni `using` registrano le operazioni di eliminazione delle risorse in uno stack, assicurandosi che vengano eliminate nell'ordine inverso rispetto alla dichiarazione:

```
{
    using j = getA(),
        y = getB();
    using k = getC();
} // elimina `C`, poi `B`, poi `A`.
```

È garantito che le risorse vengano eliminate, anche se si verificano codice o eccezioni successive. Questo potrebbe portare alla

generazione di un'eccezione durante l'eliminazione, con la possibile soppressione di un'altra. Per conservare le informazioni sugli errori soppressi, è stata introdotta una nuova eccezione nativa, `SuppressedError`.

dichiarazione await using

Una dichiarazione `await using` gestisce una risorsa eliminabile in modo asincrono. Il valore deve avere un metodo `Symbol.asyncDispose`, che verrà atteso alla fine del blocco.

```
async function doWorkAsync() {
    await using work = doWorkAsync(); // La risorsa viene
dichiarata
} // La risorsa viene eliminata (ad esempio, viene valutata `await
work[Symbol.asyncDispose]()`)
```

Per una risorsa eliminabile in modo asincrono, deve aderire all'interfaccia `Disposable` o `AsyncDisposable`:

```
// lib.esnext.disposable.d.ts
interface AsyncDisposable {
    [Symbol.asyncDispose](): Promise<void>;
}

//@ts-ignore
Symbol.asyncDispose ??= Symbol('Symbol.asyncDispose'); // Simple
polyfill

class DatabaseConnection implements AsyncDisposable {
    // Un metodo che viene chiamato quando l'oggetto viene
eliminato in modo asincrono
    [Symbol.asyncDispose]() {
        // Chiude la connessione e restituisce una promessa
        return this.close();
    }

    async close() {
        console.log('Chiusura della connessione...');
        await new Promise(resolve => setTimeout(resolve, 1000));
        console.log('Connessione chiusa.');
```

```
    }
}

async function doWork() {
    // Crea una nuova connessione e la elimina in modo asincrono
quando esce dall'ambito
    await using connection = new DatabaseConnection(); // La
risorsa viene dichiarata
    console.log('Sto lavorando...');
} // La risorsa viene eliminata (ad esempio, viene valutato `await
connection[Symbol.asyncDispose]()`)

doWork();
```

Il codice registra:

```
Sto lavorando...
Chiusura della connessione...
Connessione chiusa.
```

Le dichiarazioni `using` e `await using` sono consentite nelle istruzioni: `for`, `for-in`, `for-of`, `for-await-of`, `switch`.

Attributi di importazione

Gli attributi di importazione di TypeScript 5.3 (etichette per le importazioni) indicano al runtime come gestire i moduli (JSON, ecc.). Questo migliora la sicurezza garantendo importazioni chiare e si allinea con la Content Security Policy (CSP) per un caricamento più sicuro delle risorse. TypeScript garantisce che siano validi, ma lascia che sia il runtime a gestirne l'interpretazione per la gestione di moduli specifici.

Esempio:

```
import config from './config.json' with { type: 'json' };
```

con importazione dinamica:

```
const config = import('./config.json', { with: { type: 'json' } });
```