TypeScript in Azione

Da JavaScript ad OOP o viceversa!

Aldo Prinzi

Scopri il Potere della Programmazione ad oggetti usndo Javascript!

Benvenuto nel mondo magico della programmazione ad oggetti con un linguaggio di scripting ovvero con TypeScript, un linguaggio che trasformerà il tuo approccio alla programmazione e ti aprirà le porte ai framework più avanzati come Angular, React e Node.js.

Perché questo manuale è unico?

Perchè è stato scritto per **tutti i programmatori**: sia per quelli con esperienza che per chi è alle prime armi.

1. **Per i programmatori JavaScript**: Se sei abituato alla libertà dello scripting ma ti sei scontrato con problemi di compatibilità del codice o difficoltà nel lavorare in team, TypeScript ti offrirà un'alternativa strutturata e potente. Questo manuale ti guiderà passo dopo passo, richiedendo solo una conoscenza di base della programmazione ad oggetti e degli strumenti di sviluppo.

2. **Per i programmatori OOP**: Se provieni dal mondo della programmazione orientata agli oggetti e trovi disorientante la flessibilità di JavaScript, TypeScript sarà la tua ancora di salvezza. Questo libro ti mostrerà come portare le tue competenze OOP nel mondo dello scripting, senza la necessità di conoscere HTML o CSS.

Se hai sempre desiderato padroneggiare un linguaggio di scripting tra i più usati al mondo, mantenendo la robustezza e l'organizzazione della programmazione ad oggetti, allora questo libro è per te.

Cosa imparerai da questo manuale?

- **Fondamenti di TypeScript**: Scoprirai come TypeScript estende JavaScript aggiungendo tipizzazione statica e altre funzionalità OOP, migliorando così la manutenibilità e la scalabilità del tuo codice.
- **Strumenti e tecniche avanzate**: Imparerai a utilizzare TypeScript con i framework più popolari, rendendo il tuo codice robusto e facilmente integrabile in progetti complessi.
- **Collaborazione e compatibilità**: Capirai come TypeScript può aiutarti a lavorare meglio in team, evitando i classici problemi di compatibilità e migliorando la coesione del codice condiviso.

E quando l'avrai completato?

Quando avrai completato questo manuale, sarai pronto a utilizzare TypeScript nel mondo reale, potrai aggiungere questa competenza al tuo curriculum, aprendo nuove opportunità di lavoro.

Non importa da dove parti: che tu sia un programmatore JavaScript che vuole adottare l'OOP, o un programmatore OOP che si avvicina per la prima volta a JavaScript, questo manuale ti fornirà tutte le conoscenze e le competenze necessarie per eccellere.

Inizia oggi il tuo viaggio con TypeScript e scopri come questo linguaggio può trasformare il tuo modo di programmare, rendendoti un professionista più versatile, efficiente e richiesto.

(pagina lasciata intenzionalmente bianca)

Sommario

Sommario

Sommario

Sommario

v1.2 - Luglio 2024

Aldo Prinzi

Introduzione

Questo manuale è una guida pratica e completa per chiunque desideri padroneggiare la programmazione orientata agli oggetti con JavaScript e TypeScript.

Che tu sia un programmatore JavaScript desideroso di strutturare meglio il tuo codice o un esperto di OOP che vuole esplorare il mondo dello scripting, questo libro è stato scritto per te, preparati dunque a dare una marcia in più alla tua carriera.

Questo manuale è stato concepito tenendo a mente le necessità che mergono da queste due diverse figure del mondo della programmazione: da un lato i programmatori che già conoscono JavaScript e desiderano approfondire e adottare la Programmazione Orientata agli Oggetti (OOP) e dall'altro i programmatori che hanno esperienza con l'OOP e vogliono esplorare e padroneggiare JavaScript, un linguaggio di scripting estremamente popolare e versatile.

Un solo manuale, due mondi

L'obiettivo principale di questo manuale è pertanto duplice: spiegare i concetti fondamentali dell'OOP a chi non li conosce e illustrare come questi concetti sono implementati e utilizzati in TypeScript per chi ha già familiarità con la programmazione orientata agli oggetti.

Se sai programmare in JavaScript questo manuale ti guiderà attraverso i principi fondamentali dell'OOP: imparerai concetti come incapsulamento, ereditarietà, polimorfismo e astrazione e come questi possono essere implementati nel contesto di JavaScript e TypeScript. Questo ti permetterà di scrivere codice più organizzato, modulare e manutenibile.

Sei già conosci e usi i paradigmi dell' OOP ovvero hai una buona esperienza con linguaggi di programmazione quali Java o C#, questo manuale ti mostrerà come JavaScript e TypeScript implementano perfettamente questi concetti e scoprirai come portare sfruttare tutta la tua competenza anche nel mondo dello scripting grazie a TypeScript.

Fondamenti di TypeScript e JavaScript

TypeScript è un superset di JavaScript che aggiunge tipizzazione statica e altre potenti funzionalità OOP e in questo manuale verrà fornita una solida comprensione dei fondamenti di TypeScript e di come si trasforma in JavaScript. Inoltre, verrà spiegato come l'European Computer Manufacturers Association (ECMA) ha definito uno standard per JavaScript, mantenendo nel tempo una base comune per lo sviluppo di applicazioni web moderne.

Perché l'OOP è importante?

Imparare e padroneggiare l'OOP non è solo un vantaggio tecnico; è una chiave per migliorare significativamente la qualità del tuo codice e la tua capacità di collaborare con altri sviluppatori,

costruendo applicazioni scalabili e manutenibili, con una chiara separazione dei compiti all'interno del team di sviluppo e un riutilizzo efficace del codice.

Un vantaggio competitivo

Alla fine di questo manuale, sarai pronto per entrare nel mondo del lavoro con fiducia, sapendo di avere le competenze necessarie per avere successo e aggiungere al tuo curriculum la capacità di programmare in OOP con JavaScript e TypeScript, aumentando le tue opportunità di carriera. Questa competenza è particolarmente preziosa nel mercato del lavoro attuale, dove le aziende cercano sviluppatori in grado di scrivere codice pulito, modulare e manutenibile.

La Grande Promessa

La grande promessa di questo manuale è che, con impegno e pratica, sarai in grado di dominare la programmazione ad oggetti, perché ti verrà fornita una visione completa e pratica di come sviluppare applicazioni moderne e professionali. Sia che tu voglia migliorare il tuo lavoro con il front-end o esplorare nuovi orizzonti nel back-end, questo manuale ti fornirà gli strumenti e le conoscenze necessarie per eccellere.

Adatto a programmatori di ogni livello

Nonostante la complessità dei temi trattati, questo manuale è stato progettato per essere accessibile a programmatori di ogni livello, sia i principianti che gli esperti alla ricerca della espansione le proprie competenze. Troverai spiegazioni dettagliate e chiare, corredate da esempi pratici che facilitano la comprensione e l'applicazione dei concetti.

I Fondamenti della OOP

Imparerai i concetti fondamentali della OOP, come classi e oggetti, incapsulamento, ereditarietà, polimorfismo e astrazione. Questi principi ti permetteranno di scrivere codice che non solo funziona, ma è anche organizzato, leggibile e facile da manutenere.

TypeScript: un ponte tra JavaScript e OOP

TypeScript agisce come un ponte tra JavaScript e OOP, portando i vantaggi della tipizzazione statica e della programmazione orientata agli oggetti in un linguaggio di scripting ampiamente utilizzato.

La forza della tipizzazione statica

La tipizzazione statica di TypeScript ti aiuterà a evitare molti degli errori comuni che possono verificarsi in JavaScript: sapere in anticipo quali tipi di dati una funzione si aspetta può prevenire bug e rendere il tuo codice più robusto e attraverso questo manuale ti guiderò attraverso l'uso della tipizzazione statica e ti mostrerò come può migliorare la qualità del tuo codice.

Imparare dai migliori

Attraverso questo manuale, imparerai a utilizzare le migliori pratiche di programmazione OOP. Vedrai come i principi dell'OOP sono applicati nel mondo reale e come puoi usarli per migliorare i tuoi progetti.

Versatilità e adattabilità

I principi che imparerai sono universali e applicabili a qualsiasi linguaggio di programmazione moderno. Questo significa che, oltre a migliorare le tue competenze in JavaScript e TypeScript, sarai anche preparato per lavorare con altri linguaggi di programmazione orientati agli oggetti come Java, C#, Python e molti altri.

Partecipazione alla community

Imparare a programmare non è un viaggio solitario, ricordati che la partecipazione attiva alle community di sviluppatori può offrirti supporto, ispirazione e opportunità di networking. Ti incoraggio pertanto a partecipare a forum, gruppi di discussione e conferenze, senza limiti e senza sosta: quando tu dai una mano, ricevi una mano.

Alla fine del percorso in cui ti sto introducendo, sarai in grado di affrontare progetti complessi con fiducia, collaborare efficacemente con altri sviluppatori e dimostrare le tue competenze ai potenziali datori di lavoro.

La tua nuova capacità di programmare sarà una risorsa inestimabile, aprendo nuove opportunità di carriera e permettendoti di eccellere nel competitivo mondo dello sviluppo software.

(pagina lasciata intenzionalmente bianca)

Introduzione a ECMAScript

Iniziamo dall'inizio, da ECMAScript, che è la linfa vitale di JavaScript.

ECMAScript è uno standard di scripting creato per standardizzare JavaScript e altri linguaggi simili. ECMA International, l'organizzazione di standardizzazione dietro ECMAScript, ha giocato un ruolo cruciale nel guidare l'evoluzione di JavaScript, assicurando che rimanesse un linguaggio robusto e versatile.

Prima di iniziare a lavorare con TypeScript, è fondamentale comprendere le diverse versioni di ECMAScript e le loro caratteristiche, per sfruttare appieno le potenzialità del linguaggio e prendere decisioni informate basate sui risultati che si desidera ottenere.

L'importanza di ECMA International

ECMA International è un'organizzazione di standardizzazione che sviluppa standard tecnici per una vasta gamma di tecnologie. Fondata nel 1961, ECMA si è dedicata a promuovere la standardizzazione e l'integrazione dei sistemi informatici, dei dispositivi di comunicazione e dei componenti elettronici. Il loro lavoro è essenziale per garantire l'interoperabilità e la compatibilità tra diverse tecnologie e piattaforme.

Nel contesto di JavaScript, ECMA International ha creato e mantenuto lo standard ECMAScript, che definisce le specifiche per i linguaggi di scripting come JavaScript ed è responsabile dello sviluppo e dell'evoluzione di ECMAScript. Il comitato è composto da esperti provenienti da diverse aziende tecnologiche leader, che collaborano per migliorare lo standard.

Com'è facilmente intuibile, la standardizzazione è fondamentale per assicurare che i programmatori abbiano una piattaforma di sviluppo affidabile e prevedibile, senza ogni browser o ambiente di esecuzione potrebbe implementare JavaScript in modo diverso, creando inconsistenze e incompatibilità che renderebbero lo sviluppo web molto più complesso e frustrante.

Grazie ad ECMA, gli sviluppatori possono scrivere codice JavaScript che funziona coerentemente su diverse piattaforme e browser e ciò è particolarmente importante nel contesto del web, dove gli utenti accedono ai siti e alle applicazioni utilizzando una discreta varietà di dispositivi e browser e le funzionalità e le caratteristiche di JavaScript sono implementate in modo affidabile.

Storia di ECMAScript

JavaScript in origine è stato pensato e creato da Brendan Eich per essere usato con il browser Netscape e l'origine di ECMAScript risale alla metà degli anni '90 perché con l'espansione di uso di un linguaggio di scripting per interoperare con i contenuti di una pagina HTML, e con l'introduzione di diversi browser, emerse il bisogno di un insieme di regole standard per garantire la compatibilità e l'interoperabilità tra questi browser e le piattaforme.

Nel 1997 è stato pubblicato il primo **standard ECMAScript** (ECMA-262) e man mano che diveniva necessario introdurre nuove funzionalità ne veniva rilasciata una nuova versione che

introduceva gli standard di definzione di queste nuove funzionalità e i miglioramenti, mantenendo il linguaggio stabile. Di seguito una necessaria panoramica delle principali versioni e delle loro caratteristiche distintive:

ECMAScript 1 (ES1) - 1997

E' la prima versione standard di ECMAScript e stabilisce le basi per la sintassi e le funzionalità di JavaScript.

ECMAScript 2 (ES2) - 1998

Aggiornamenti minori per mantenere l'allineamento con lo standard internazionale ISO/IEC 16262.

ECMAScript 3 (ES3) - 1999

Introduce importanti funzionalità come il supporto per le espressioni regolari, il miglioramento della gestione delle stringhe, e il blocco try/catch per la gestione delle eccezioni.

ECMAScript 4 (ES4) - *(Mai ufficialmente rilasciato)*

Conteneva molte proposte avanzate, ma non è stato adottato a causa di disaccordi tra i membri del comitato (!!!). Ricordo all'uopo che quel periodo iniziò l'introduzione di linguaggi di scripting da parte di Microsoft che tendeva a discostarsi dagli standard condivisi a causa del suo prodotto "Internet Explorer" che era diverso dagli altri.
Questi anni (1999-2009) sono caratterizzati dal fiorire di soluzioni più o meno strane e dall'assenza di standard e dalla confusione di sviluppo in cui i programmatori erano costretti a scegliere una piattaforma su cui specializzarsi.

ECMAScript 5 (ES5) - 2009

Introduce:

- La modalità "strict mode" per migliorare la gestione degli errori.
- Aggiunge nuove funzionalità come Array.forEach, Object.keys, e metodi di manipolazione degli oggetti.
- Migliora il supporto per JSON.

ECMAScript 6 (ES6 / ECMAScript 2015)

Una delle versioni più significative, con numerose nuove funzionalità come:

- **Let e Const**: nuove modalità per dichiarare variabili.
- **Arrow Functions**: sintassi più concisa per le funzioni anonime.
- **Template Literals**: modalità avanzate per la manipolazione delle stringhe.
- **Classi**: sintassi semplificata per la creazione di oggetti e ereditarietà.
- **Moduli**: supporto per l'importazione e l'esportazione di codice.
- **Promises**: gestione più semplice delle operazioni asincrone.

ECMAScript 7 (ES7 / ECMAScript 2016)

Introduce due principali nuove funzionalità:

- L'operatore di esponenziazione (**).
- **Array.prototype.includes** per verificare se un array contiene un determinato elemento.

ECMAScript 8 (ES8 / ECMAScript 2017)

Aggiunge funzionalità come:

- **Async/Await**: gestione asincrona del codice in modo più leggibile.
- **Object.values** e **Object.entries**: metodi per ottenere i valori e le coppie chiave-valore di un oggetto.
- **String Padding** (**padStart** e **padEnd**): aggiunta di caratteri all'inizio o alla fine delle stringhe.

ECMAScript 9 (ES9 / ECMAScript 2018)

Introduce funzionalità come:

- **Rest/Spread Properties**: estensione dell'operatore spread per oggetti.
- **Asynchronous Iteration**: supporto per l'iterazione asincrona.
- **Promise.prototype.finally**: metodo per eseguire codice una volta che una promessa è stata risolta o rigettata.

ECMAScript 10 (ES10 / ECMAScript 2019)

Nuove funzionalità come:

- **Array.prototype.flat** e **Array.prototype.flatMap**: metodi per appiattire array.
- **Object.fromEntries**: metodo per creare oggetti da array di coppie chiave-valore.
- **String trimStart** e **trimEnd**: metodi per rimuovere spazi bianchi dalle stringhe.

ECMAScript 11 (ES11/ECMAScript 2020)

Nuove funzionalità come:

- **Optional Chaining** (?.): accesso sicuro a proprietà annidate.
- **Nullish Coalescing** (??): operatore per fornire valori predefiniti per *null* o *undefined*.
- **Dynamic Import**: possibilità di importare moduli dinamicamente.

Importanza di ECMAScript

L'evoluzione di ECMAScript è cruciale per il progresso e l'adozione di JavaScript come linguaggio di programmazione universale. Con ogni nuova versione, gli sviluppatori ricevono strumenti più potenti e flessibili per scrivere codice efficiente, leggibile e mantenibile.

Mantenersi aggiornati con le ultime versioni di ECMAScript è fondamentale per sfruttare al meglio le potenzialità di JavaScript e rimanere competitivi nel mondo dello sviluppo web, così

come è importante conoscere e sapere cose è disponibile (e cosa no) rispetto le varie versioni di ECMAScript.

Per compilare Typescript infatti, si deve scegliere una versione di ECMAScript finale del proprio codice.

Compatibilità ECMAScript nei Vari Browser

Conoscere la compatibilità ECMAScript per i diversi browser aiuta a scrivere codice che funziona correttamente sul maggior numero di piattaforme e decidere quali siano quelle supportate.

Ma come faccio a scegliere quale versione di ECMAScript sia quella corretta per il mio progetto?

La compatibilità di ECMAScript varia tra i diversi browser web e le loro versioni. Ogni browser implementa le caratteristiche di ECMAScript in tempi diversi, e quindi è essenziale conoscere quali funzionalità sono supportate da ciascun browser e versione.

Qui di seguito una lista di livelli di compatibilità di ECMAScript per i principali browser web alla data di edizione di questo manuale, per le altre info usa il riquadro "**Can I Use**" in fondo a questo capitolo:

Google Chrome

- **Versione attuale:** Chrome è noto per essere uno dei browser più rapidi ad adottare nuove funzionalità ECMAScript. Le versioni moderne di Chrome supportano quasi tutte le caratteristiche di ECMAScript fino a ES11.
- **Compatibilità:** Ottima per le versioni ES6, ES7, ES8, ES9, ES10, e ES11.

Mozilla Firefox

- **Versione attuale:** Firefox è anche molto veloce nell'adozione delle nuove funzionalità ECMAScript. Le versioni recenti di Firefox supportano quasi tutte le caratteristiche di ECMAScript fino a ES11.
- **Compatibilità:** Ottima per le versioni ES6, ES7, ES8, ES9, ES10, e ES11.

Microsoft Edge

- **Versione attuale:** Le versioni moderne di Edge, basate su Chromium, supportano le stesse caratteristiche ECMAScript di Chrome. Le versioni precedenti (Edge Legacy) hanno una compatibilità inferiore.
- **Compatibilità:** Ottima per le versioni ES6, ES7, ES8, ES9, ES10, e ES11 nelle versioni basate su Chromium.

Safari

- **Versione attuale:** Safari tende ad essere un po' più lento nell'adozione delle nuove funzionalità ECMAScript rispetto a Chrome e Firefox, ma le versioni recenti supportano molte delle nuove caratteristiche.
- **Compatibilità:** Buona per ES6, ES7, ES8, ES9, e ES10. Il supporto per ES11 può essere parziale.

Opera

- **Versione attuale:** Essendo basato su Chromium, Opera offre un livello di compatibilità simile a quello di Chrome.
- **Compatibilità:** Ottima per le versioni ES6, ES7, ES8, ES9, ES10, e ES11.

Internet Explorer

- **Versione attuale:** Internet Explorer (IE) ha una compatibilità limitata con le versioni più recenti di ECMAScript. IE11 supporta alcune funzionalità di ES5 e poche di ES6.
- **Compatibilità:** Limitata, principalmente per ES5 e alcune caratteristiche di ES6.

Verifica della Compatibilità

Per verificare la compatibilità delle caratteristiche di ECMAScript con le versioni dei browser, puoi utilizzare il sito *Can I use*. Questo sito offre una panoramica completa delle caratteristiche di ECMAScript e la loro compatibilità con i diversi browser e le loro versioni.

Come Usare "Can I use"

- Vai al sito **https://caniuse.com/**
- Nella barra di ricerca, digita la caratteristica di ECMAScript che vuoi verificare (es. "arrow functions", "async/await", ecc.).
- Il sito mostrerà una tabella con la compatibilità della caratteristica nei vari browser, inclusi Google Chrome, Firefox, Edge, Safari, Opera, e Internet Explorer.

(pagina lasciata intenzionalmente bianca)

Programmatori JavaScript che vogliono usare l'OOP

Questo capitolo è dedicato ai programmatori JavaScript che desiderano imparare e adottare la Programmazione Orientata agli Oggetti (OOP).

Se invece sei già un programmatore OOP e vuoi approcciare questa tecnica con Javascript/Typescript, puoi direttamente passare all'introduzione a te dedicata, che è successiva a questa.

Se sei rimasto qui allora sei alle prime armi con l'OOP e ne hai sentito parlare, ma non sai esattamente cosa sia o come implementarla nel tuo codice, sei nel posto giusto.

La Programmazione Orientata agli Oggetti (OOP) è un paradigma di programmazione che offre una struttura organizzativa per il codice, facilitando la gestione, la manutenzione e l'espansione delle applicazioni.

JavaScript, sebbene inizialmente concepito come un linguaggio di scripting per il web, ha adottato molti concetti OOP nelle sue versioni più recenti, rendendo possibile scrivere codice più robusto e scalabile.

In questo capitolo, esploreremo i concetti fondamentali dell'OOP e vedremo come possono essere applicati con l'obiettivo di fornirti una solida comprensione teorica dei principi dell'OOP e mostrarti come questi principi possano migliorare la qualità e la manutenibilità del tuo codice JavaScript.

Introduzione

La programmazione orientata agli oggetti è un modello di programmazione basato sull'uso di oggetti e per comprendere meglio questo concetto, facciamo un parallelo con il mondo reale: immagina di avere un oggetto "bicchiere", un contenitore che può essere utilizzato per vari scopi, come contenere acqua da bere, liquidi da analizzare e altro ancora.

In questo contesto, il bicchiere rappresenta un oggetto che può essere riutilizzato in diverse situazioni, proprio come un oggetto in OOP può essere utilizzato in diverse parti del codice.

Oggetti e Classi

Gli oggetti sono *istanze di classi* che possono contenere dati (sotto forma di proprietà o attributi) e codice (sotto forma di metodi o funzioni).

Pensa ad una classe come a una "ricetta" o "modello" per creare oggetticome ad esempio la classe "Bicchiere". Questa può avere proprietà come "materiale" e "capacità", e metodi come "riempire()" e "svuotare()".

Quando creiamo un'istanza della classe Bicchiere, stiamo creando un oggetto che segue il modello definito dalla classe.

Ora, pensiamo a un esempio più pratico per un programmatore: immagina di voler creare una classe che rappresenti un dato, come un utente, un post o un prodotto e immagina di creare una classe per ciascuno di questi elementi.

Questo rende molto più facile lavorare con i dati, perché ogni oggetto creato dalla classe conterrà tutte le informazioni e le funzionalità necessarie per gestire quei dati, specie quando si lavora in team o dopo tanto tempo che non "tocchi" quel codice, tute le proprietà e le funzionalità saranno inserite nella classe.

Incapsulamento

Uno dei pilastri fondamentali dell'OOP è l'incapsulamento, che consiste nel nascondere i dettagli implementativi di una classe e nel fornire un'interfaccia pubblica per l'interazione con l'oggetto per aiutare a proteggere lo stato interno dell'oggetto e a prevenire modifiche non autorizzate o accidentali.

Ad esempio, possiamo pensare a un bicchiere che ha una proprietà "contenuto".

L'incapsulamento definisce come il contenuto viene aggiunto o rimosso, assicurando che il bicchiere non venga riempito oltre la sua capacità.

Tornando al contesto della programmazione, questo significa che quando crei una classe Utente, puoi definire proprietà come nome, email, password e così anche i metodi per leggere e aggiornare queste informazioni da una "fonte dati", che potrà essere un database, un file di testo, un excel, che importa? L'implementazione sarà interna.

L'utente che utilizza l'oggetto non deve preoccuparsi di come questi dati vengono gestiti internamente; può semplicemente chiamare i metodi appropriati per ottenere o modificare i dati.

Ereditarietà

L'ereditarietà invece è un meccanismo che permette a una classe di derivare da un'altra classe, ereditandone le proprietà e i metodi.

Pensiamo al bicchiere come a una *classe base* dalla quale possiamo creare una *classe derivata* chiamata "Calice", che eredita le proprietà e i metodi della classe Bicchiere (è di vetro, serve a contenere liquidi), ma aggiunge anche alcune caratteristiche specifiche, come ad esempio una forma diversa, una capacità maggiore o la forzatura al tipo di liquido che si può usare all'interno.

Allo stesso modo, possiamo creare una classe "Tazza", che eredita dalla classe Bicchiere ma ha proprietà diverse, come un manico e un materiale di composizione in ceramica.

Nella programmazione, questo significa che puoi avere una funzione che accetta oggetti *di tipo* Prodotto, che saranno sicuramente oggetti specializzati (Elettronica, Abbigliamento o Alimentari che sono *di tipo* Prodotto).

La funzione può operare su questi oggetti senza sapere esattamente quale tipo di prodotto sta trattando, rendendo il codice più flessibile e riutilizzabile.

Polimorfismo

Il polimorfismo introduce il concetto di *tipo* e in sé rappresenta il modo di trattare *oggetti* di *classi diverse* come se fossero dello stesso *tipo*.

Ad esempio, possiamo avere un metodo che accetta un oggetto di tipo Bicchiere come parametro, ma possiamo passare sia un oggetto Calice che un oggetto Tazza a questo metodo, grazie proprio al polimorfismo.

Questo permette di scrivere codice più flessibile e riutilizzabile, poiché possiamo utilizzare lo stesso metodo per operare su oggetti di tipi diversi.

Nella programmazione, questo significa che puoi avere una funzione che accetta un oggetto di tipo Prodotto, ma puoi passare oggetti di tipo Elettronica, Abbigliamento o Alimentari a questa funzione. La funzione può operare su questi oggetti senza sapere esattamente quale tipo di prodotto sta trattando, rendendo il codice più flessibile e riutilizzabile.

Astrazione

L'astrazione è ciò che ci permette di nascondere i dettagli implementativi di un oggetto e di mostrarne solo le funzionalità essenziali.

Pensiamo a un bicchiere come a un concetto astratto che rappresenta qualsiasi tipo di contenitore. Possiamo quindi definire una *classe astratta* "Contenitore" che ha proprietà e metodi comuni a tutti i contenitori, come "materiale" e "capacità", e metodi come "riempire()" e "svuotare()", ma la cui implementazione dipende dalla classe definitiva.

Le classi Bicchiere, Calice e Tazza possono derivare dalla classe Contenitore, implementando i dettagli specifici per ciascun tipo di contenitore, ma possiamo scrivere del codice che accetti come parametro un Contenitore e/o che restituisca in risposta un Contenitore.

Nella programmazione, questo significa che puoi creare una classe astratta Utente che ha proprietà e metodi comuni a tutti i tipi di utenti, come nome e email. Poi puoi creare classi derivate come Amministratore e Cliente, che implementano i dettagli specifici per ciascun tipo di utente. Questo rende il codice più modulare e facile da mantenere.

Vantaggi dell'OOP

Adottare l'OOP in JavaScript offre numerosi vantaggi:

Organizzazione del Codice: L'OOP aiuta a organizzare il codice in moduli distinti e riutilizzabili, facilitando la manutenzione e l'espansione delle applicazioni.

Riutilizzo del Codice: L'ereditarietà permette di riutilizzare il codice esistente, riducendo la duplicazione e migliorando l'efficienza dello sviluppo.

Manutenibilità: L'incapsulamento e l'astrazione semplificano la gestione delle modifiche, poiché i dettagli implementativi sono nascosti e le modifiche possono essere fatte senza influenzare altre parti del codice.

Estensibilità: Il polimorfismo consente di estendere le funzionalità delle applicazioni senza modificare il codice esistente, rendendo più facile l'aggiunta di nuove funzionalità.

Implementazione dell'OOP in JavaScript

In JavaScript, le versioni più recenti del linguaggio hanno introdotto il supporto per le classi e i concetti OOP, utilizzando la parola chiave `class`.

Possiamo quindi definire *classi* e creare *oggetti* basati su queste *classi*.

Possiamo anche utilizzare la parola chiave `extends` per implementare l'ereditarietà, e la parola chiave `super` per chiamare il costruttore della classe base all'interno della classe derivata, ma vedremo le implementazioni di questi concetti mano mano in questo manuale.

Per un programmatore alle prime armi, è interessante notare come l'uso delle classi renda il codice più leggibile e manutenibile. Ad esempio, quando si lavora con dati complessi come utenti, post o prodotti, è molto utile creare una classe per ciascun tipo di dato. Questo permette di incapsulare tutte le proprietà e i metodi relativi a quel dato all'interno della classe, facilitando la gestione e l'aggiornamento del codice.

Quando un'applicazione cresce in complessità, il mantenimento del codice può diventare un problema se non si utilizza una struttura organizzativa adeguata. Con l'OOP, puoi apportare modifiche alla classe base e vedere queste modifiche propagate a tutte le classi derivate. Questo significa che se decidi di cambiare il modo in cui un bicchiere viene riempito, puoi fare la modifica nella classe base Bicchiere e tutte le classi.

Programmatori OOP che vogliono usare JavaScript

Questo capitolo è dedicato ai programmatori con esperienza nella Programmazione Orientata agli Oggetti (OOP) che desiderano trasferire le loro competenze nel mondo di JavaScript e TypeScript, coloro che, per capirci, sono già abituati a lavorare con linguaggi come Java, C++, o C#, e ora stanno cercando di espandere le loro abilità per includere lo sviluppo web. Se ti ritrovi in questa descrizione, beh, sei nel posto giusto! Attraverso questo manuale ti guiderò verso la transizione, mostrando come le tue conoscenze OOP possono essere applicate efficacemente in JavaScript e TypeScript.

Differenze tra JavaScript e TypeScript

JavaScript è un linguaggio di scripting che è diventato il fondamento dello sviluppo web front-end, conosciuto per la sua flessibilità e la sua capacità di eseguire codice direttamente nei browser. Tuttavia, per i programmatori abituati a linguaggi fortemente tipizzati e strutturati, JavaScript può sembrare caotico e privo di struttura.

TypeScript, d'altra parte, è un superset di JavaScript che introduce la tipizzazione statica e altre caratteristiche avanzate, che colma molte delle lacune di JavaScript, rendendolo un linguaggio più robusto e adatto per la programmazione OOP.

Ecco alcune delle principali differenze tra JavaScript e TypeScript:

- **Tipizzazione**: Mentre JavaScript è un linguaggio debolmente tipizzato, TypeScript introduce la tipizzazione statica, permettendo di definire tipi per variabili, funzioni, e oggetti. Questo aiuta a prevenire molti errori a tempo di esecuzione e rende il codice più leggibile e manutenibile.
- **Classi e Moduli**: JavaScript ha introdotto il concetto di classi con ES6, ma TypeScript estende queste funzionalità aggiungendo modificatori di accesso (public, private, protected) e supporto per le interfacce.
- **Compatibilità**: TypeScript viene compilato in JavaScript, il che significa che puoi usare TypeScript in qualsiasi progetto JavaScript esistente. Questo processo di

compilazione permette di utilizzare funzionalità avanzate senza perdere la compatibilità con i browser che supportano solo JavaScript.

- **Vantaggi di usare TypeScript per la programmazione OOP**
- Adottare TypeScript per la programmazione OOP offre numerosi vantaggi, specialmente per chi proviene da linguaggi OOP tradizionali. Vediamo i principali benefici:
- **Tipizzazione Statica**: La tipizzazione statica di TypeScript permette di rilevare errori a tempo di compilazione piuttosto che a tempo di esecuzione, riducendo i bug e migliorando la qualità del codice.
- **Manutenibilità**: Grazie alla presenza di classi, interfacce e moduli, TypeScript rende il codice più organizzato e manutenibile. Questo è particolarmente utile in grandi progetti con team di sviluppo numerosi.
- **Strumenti di Sviluppo**: Gli IDE moderni come Visual Studio Code offrono un eccellente supporto per TypeScript, inclusi il completamento del codice, il refactoring, e il debugging avanzato, migliorando significativamente l'esperienza di sviluppo.
- **Compatibilità con JavaScript**: TypeScript è completamente compatibile con JavaScript. Questo significa che puoi introdurre TypeScript gradualmente in un progetto JavaScript esistente, senza la necessità di riscrivere tutto il codice da zero.

Panoramica sulle versioni ECMAScript

Come vedrai più avanti, ECMAScript è lo standard su cui si basa JavaScript e le versioni successive di ECMAScript hanno introdotto nuove funzionalità che hanno reso JavaScript un linguaggio più potente e versatile. Ecco una panoramica delle versioni più rilevanti:

ES5 (ECMAScript 2009): Ha introdotto funzionalità fondamentali come JSON nativo e modalità strict.

ES6 (ECMAScript 2015): Una delle versioni più significative, ha introdotto classi, moduli, let e const, arrow functions, template literals, generatori, e molto altro. ES6 ha trasformato JavaScript in un linguaggio più moderno e potente.

ES7 (ECMAScript 2016): Ha introdotto l'operatore di esponenziazione (`**`) e il metodo `Array.prototype.includes`.

ES8 (ECMAScript 2017): Questa versione ha aggiunto async/await, Object.entries, Object.values, e le string padding.

ES9 (ECMAScript 2018): Ha introdotto rest/spread properties per oggetti, iterazione asincrona, e Promise.prototype.finally.

ES10 (ECMAScript 2019): Ha aggiunto Array.prototype.flat e Array.prototype.flatMap, Object.fromEntries, e string.prototype.trimStart e string.prototype.trimEnd.

Concetti di OOP in JavaScript

JavaScript, sebbene inizialmente concepito come linguaggio di scripting, supporta molti dei concetti della Programmazione Orientata agli Oggetti, specialmente con l'introduzione di ES6. Tuttavia, la sintassi e le funzionalità offerte da JavaScript possono risultare meno intuitive per i programmatori abituati ai linguaggi OOP tradizionali. Vediamo come JavaScript implementa i principali concetti OOP.

Come JavaScript implementa i concetti di OOP

JavaScript utilizza un modello basato sui prototipi per implementare l'ereditarietà e altri concetti OOP. Invece di utilizzare classi come in Java o C#, JavaScript permette di creare oggetti che possono essere utilizzati come prototipi per altri oggetti. Questo modello offre una grande flessibilità, ma può risultare meno intuitivo per chi è abituato alle classi tradizionali.

Con l'introduzione di ES6, JavaScript ha aggiunto il supporto per le classi, che forniscono una sintassi più familiare ai programmatori OOP. Le classi in JavaScript sono in realtà una "sugar syntax" sopra il modello a prototipi, ma rendono il codice più leggibile e manutenibile.

Uso delle funzioni costruttore

Prima dell'introduzione delle classi in ES6, il modo principale per creare oggetti in JavaScript era attraverso le funzioni costruttore. Una funzione costruttore è una normale funzione che viene utilizzata con l'operatore `new` per creare nuovi oggetti. Le funzioni costruttore permettono di definire proprietà e metodi per gli oggetti che creano, e possono essere utilizzate per implementare l'ereditarietà tramite l'uso dei prototipi.

Uso delle classi ES6

Le classi introdotte in ES6 forniscono una sintassi più concisa e leggibile per definire oggetti e implementare l'ereditarietà. Le classi in JavaScript supportano costruttori, metodi, getter, setter e metodi statici, e possono essere estese per creare classi derivate. Questo rende molto più facile implementare i concetti della OOP in JavaScript e permette di scrivere codice più organizzato e manutenibile.

Uso di TypeScript per la programmazione OOP

TypeScript aggiunge un ulteriore livello di funzionalità e sicurezza sopra JavaScript, rendendolo una scelta ideale per la programmazione OOP. Vediamo come TypeScript migliora la programmazione OOP.

Introduzione a TypeScript

TypeScript è un linguaggio di programmazione sviluppato da Microsoft che estende JavaScript aggiungendo tipizzazione statica e altre funzionalità avanzate. TypeScript viene compilato in JavaScript, il che significa che tutto il codice TypeScript può essere eseguito in qualsiasi ambiente che supporta JavaScript.

Tipizzazione statica vs. dinamica

Una delle principali differenze tra TypeScript e JavaScript è la tipizzazione. In JavaScript, i tipi sono determinati a tempo di esecuzione, il che può portare a errori difficili da individuare e correggere. TypeScript, d'altra parte, utilizza la tipizzazione statica, il che significa che i tipi sono conosciuti e controllati a tempo di compilazione. Questo permette di rilevare molti errori prima che il codice venga eseguito, migliorando la qualità e la sicurezza del software.

Installazione e configurazione di TypeScript

Per iniziare a utilizzare TypeScript, è necessario installarlo e configurarlo nel proprio ambiente di sviluppo. TypeScript può essere installato tramite npm, il gestore di pacchetti di Node.js. Una volta installato, è possibile configurare TypeScript creando un file `tsconfig.json` che definisce le opzioni di compilazione.

Implementazione di OOP con TypeScript

TypeScript rende l'implementazione della Programmazione Orientata agli Oggetti più intuitiva e potente. Ecco come puoi implementare OOP con TypeScript.

Creazione di classi in TypeScript

La creazione di classi in TypeScript è simile a quella in altri linguaggi OOP. Puoi definire una classe con proprietà, metodi, costruttori e utilizzare i modificatori di accesso per controllare la visibilità delle proprietà e dei metodi.

Modificatori di accesso: public, private, protected

TypeScript supporta i modificatori di accesso `public`, `private` e `protected`, che permettono di controllare la visibilità delle proprietà e dei metodi all'interno delle classi. Questo è un aspetto fondamentale della programmazione orientata agli oggetti, in quanto permette di implementare l'incapsulamento e di proteggere lo stato interno degli oggetti.

Public: Le proprietà e i metodi dichiarati come `public` sono accessibili da qualsiasi parte del programma. Per impostazione predefinita, tutte le proprietà e i metodi sono `public` se non viene specificato un modificatore di accesso.

Private: Le proprietà e i metodi dichiarati come `private` sono accessibili solo all'interno della classe in cui sono stati definiti. Questo permette di nascondere i dettagli implementativi e di proteggere l'integrità dell'oggetto.

Protected: Le proprietà e i metodi dichiarati come `protected` sono simili ai `private`, ma possono essere accessibili anche dalle classi derivate. Questo è utile per permettere l'ereditarietà senza esporre troppo l'implementazione interna.

Proprietà e metodi di classe

Le proprietà di classe in TypeScript possono essere dichiarate direttamente all'interno del corpo della classe. Queste proprietà possono avere valori di default e possono essere inizializzate nel costruttore della classe. I metodi di classe sono funzioni associate a un oggetto e possono operare sulle proprietà dell'oggetto stesso. TypeScript permette di definire metodi normali, metodi statici e metodi getter e setter per gestire l'accesso e la manipolazione delle proprietà.

Ereditarietà in TypeScript

L'ereditarietà è un concetto fondamentale della programmazione orientata agli oggetti, e TypeScript supporta pienamente l'ereditarietà attraverso le parole chiave `extends` e `super`.

Definizione e implementazione

L'ereditarietà permette a una classe di derivare da un'altra classe, ereditando le sue proprietà e i suoi metodi. In TypeScript, puoi definire una classe base e una classe derivata utilizzando la parola chiave `extends`. La classe derivata può aggiungere nuove proprietà e metodi, oltre a ridefinire quelli esistenti nella classe base.

Uso delle parole chiave extends e super

La parola chiave `extends` viene utilizzata per creare una classe derivata che eredita da una classe base. La parola chiave `super` viene utilizzata all'interno del costruttore della classe derivata per chiamare il costruttore della classe base. Questo permette di inizializzare correttamente le proprietà ereditate e di eseguire eventuali logiche di inizializzazione della classe base.

Polimorfismo in TypeScript

Il polimorfismo è un altro concetto chiave della programmazione orientata agli oggetti, che permette di trattare oggetti di classi diverse come se fossero dello stesso tipo. TypeScript supporta il polimorfismo attraverso il metodo di overloading e overriding dei metodi, oltre all'uso di interfacce e classi astratte.

Overloading e overriding dei metodi

Overloading: TypeScript permette di definire più versioni di un metodo con lo stesso nome ma con parametri diversi. Questo è utile per creare metodi che possono accettare diversi tipi di input.

Overriding: Le classi derivate possono ridefinire i metodi delle classi base per fornire una nuova implementazione. Questo è utile per modificare o estendere il comportamento delle classi base.

Interfacce e classi astratte

Interfacce: Le interfacce in TypeScript permettono di definire contratti di codice, specificando quali proprietà e metodi una classe deve implementare. Le interfacce sono uno strumento potente per garantire che le classi rispettino determinate strutture e comportamenti.

Classi astratte: Le classi astratte sono classi che non possono essere istanziate direttamente, ma devono essere derivate da altre classi. Le classi astratte possono contenere metodi concreti e astratti. I metodi astratti sono dichiarati ma non implementati, lasciando l'implementazione alle classi derivate.

Astrazione in TypeScript

L'astrazione è il processo di nascondere i dettagli implementativi di un oggetto e di mostrare solo le funzionalità essenziali. TypeScript supporta l'astrazione attraverso l'uso di classi astratte e interfacce.

Creazione di classi astratte e interfacce

Le classi astratte e le interfacce permettono di definire un'interfaccia comune per un gruppo di classi correlate. Le classi astratte possono contenere sia metodi implementati che non implementati, mentre le interfacce contengono solo dichiarazioni di metodi e proprietà senza implementazione. Questo permette di creare strutture di codice più flessibili e riutilizzabili.

Utilizzo delle interfacce per definire contratti di codice

Le interfacce sono uno strumento potente per definire contratti di codice in TypeScript. Permettono di specificare quali proprietà e metodi una classe deve implementare, garantendo che le classi rispettino determinate strutture e comportamenti. Le interfacce possono essere utilizzate anche per descrivere la forma degli oggetti, rendendo il codice più leggibile e manutenibile.

Compilazione di TypeScript in JavaScript

Uno degli aspetti più potenti di TypeScript è la sua capacità di essere compilato in JavaScript. Questo permette di utilizzare le funzionalità avanzate di TypeScript senza perdere la compatibilità con i browser che supportano solo JavaScript.

Come TypeScript compila in JavaScript

Il processo di compilazione di TypeScript in JavaScript è semplice e trasparente. TypeScript viene compilato in JavaScript utilizzando il comando `tsc` (TypeScript Compiler). Il compilatore converte il codice TypeScript in codice JavaScript, rimuovendo

i tipi e le altre caratteristiche specifiche di TypeScript, mantenendo comunque la logica e la struttura del codice.

Scelta della versione ECMAScript

Quando compili il tuo codice TypeScript, puoi scegliere quale versione di ECMAScript generare. Questo ti permette di scrivere codice TypeScript utilizzando le funzionalità più recenti del linguaggio, mentre il compilatore si occupa di generare JavaScript compatibile con i browser che supportano solo versioni precedenti di ECMAScript.

Configurazione del file tsconfig.json

Il file `tsconfig.json` è utilizzato per configurare il compilatore TypeScript. In questo file puoi specificare le opzioni di compilazione, come la versione ECMAScript da generare, le directory di input e output, e altre impostazioni avanzate. Configurare correttamente il file `tsconfig.json` è essenziale per ottimizzare il processo di compilazione e garantire che il codice generato sia compatibile con l'ambiente di esecuzione desiderato.

Conclusione

Questo capitolo ha fornito una panoramica completa su come i programmatori con esperienza OOP possono adottare JavaScript e TypeScript per implementare i concetti della Programmazione Orientata agli Oggetti. Abbiamo esplorato le differenze tra JavaScript e TypeScript, i vantaggi di utilizzare TypeScript, e come JavaScript implementa i concetti OOP. Inoltre, abbiamo discusso l'uso delle funzioni costruttore, delle classi ES6, e come TypeScript migliora ulteriormente l'implementazione della OOP con la tipizzazione statica, i modificatori di accesso, e il supporto per l'ereditarietà, il polimorfismo e l'astrazione.

Passando da un linguaggio OOP tradizionale a JavaScript/TypeScript, potrai sfruttare le tue competenze esistenti e applicarle in un contesto di sviluppo web, migliorando significativamente le tue opportunità di carriera e la tua capacità di lavorare in team di sviluppo moderni.

(pagina lasciata intenzionalmente bianca)

Installazione

Prima di iniziare a scrivere un bel codice TypeScript, devi ovviamente installare TypeScript. Questo può essere fatto con l'aiuto di npm.

Se non hai npm installato, dovrai installare npm prima di installare TypeScript, altrimenti salta questa parte e vai direttamente all'installazione di Typescript.

Come installare NPM su Windows

NPM (Node Package Manager) è un componente essenziale per lavorare con TypeScript e altri strumenti JavaScript. Segui questi passaggi per installarlo su Windows.

Passo 1: Scarica Node.js

NPM viene installato automaticamente con Node.js. Quindi, il primo passo è scaricare Node.js.

- Vai al sito ufficiale di Node.js: **https://nodejs.org**.
- Clicca sul pulsante verde "LTS" per scaricare la versione stabile consigliata per la maggior parte degli utenti.

Passo 2: Installa Node.js

- Apri il file di installazione appena scaricato.
- Apparirà una finestra di dialogo di installazione. Clicca su "Next" per continuare.
- Accetta il contratto di licenza e clicca su "Next".
- Scegli la cartella di destinazione per l'installazione e clicca su "Next".
- Nella schermata successiva, assicurati che l'opzione "Node.js runtime" sia selezionata, insieme all'opzione "npm package manager". Clicca su "Next".
- Clicca su "Install" per avviare l'installazione. Potrebbe essere richiesto di confermare le autorizzazioni amministrative.

Passo 3: Verifica l'installazione

Dopo che l'installazione è completata, verifica che Node.js e NPM siano stati installati correttamente.

- Apri il prompt dei comandi. Puoi farlo cercando "cmd" nel menu Start.

- Digita node -v e premi Invio. Dovresti vedere il numero di versione di Node.js.
- Digita npm -v e premi Invio. Dovresti vedere il numero di versione di NPM.
- Se vedi i numeri di versione per entrambi, significa che l'installazione è andata a buon fine!

Passo 4: Aggiorna NPM (opzionale)

Node.js viene fornito con una versione di NPM, ma potresti voler aggiornare NPM all'ultima versione disponibile.

- Nel prompt dei comandi, digita npm install -g npm e premi Invio. Questo comando aggiornerà NPM alla versione più recente disponibile.

Passo 5: Configurazione dell'ambiente (opzionale)

Potresti voler configurare alcune impostazioni dell'ambiente per migliorare la tua esperienza con NPM.

Per esempio, puoi configurare una directory globale per i pacchetti installati da NPM. Digita:

```
npm config set prefix "C:\Users\YourUsername\npm-global"
```

nel prompt dei comandi e premi Invio.

Aggiungi questa directory al tuo PATH di sistema:

vai a "Pannello di controllo"

> "Sistema"

> "Impostazioni avanzate di sistema".

Clicca su "Variabili d'ambiente" e modifica la variabile PATH aggiungendo *C:\Users\YourUsername\npm-global\bin*.

Come installare NPM su macOS

NPM (Node Package Manager) è un componente essenziale per lavorare con TypeScript e altri strumenti JavaScript. Segui questi passaggi per installarlo su macOS.

Passo 1: Scarica Node.js

NPM viene installato automaticamente con Node.js. Quindi, il primo passo è scaricare Node.js.

- Vai al sito ufficiale di Node.js: **https://nodejs.org**.
- Clicca sul pulsante verde "LTS" per scaricare la versione stabile consigliata per la maggior parte degli utenti.

Passo 2: Installa Node.js

Apri il file di installazione .pkg appena scaricato.

- Apparirà una finestra di dialogo di installazione. Clicca su "Continue" per continuare.
- Accetta il contratto di licenza e clicca su "Continue", quindi su "Agree".
- Scegli la destinazione per l'installazione e clicca su "Continue".
- Clicca su "Install" per avviare l'installazione. Potrebbe essere richiesto di inserire la password dell'amministratore.
- Una volta completata l'installazione, clicca su "Close".

Passo 3: Verifica l'installazione

Dopo che l'installazione è completata, verifica che Node.js e NPM siano stati installati correttamente.

- Apri il Terminale. Puoi farlo cercando "Terminal" nell'applicazione "Spotlight".
- Digita ***node -v*** e premi Invio. Dovresti vedere il numero di versione di Node.js.
- Digita ***npm -v*** e premi Invio. Dovresti vedere il numero di versione di NPM.
- Se vedi i numeri di versione per entrambi, significa che l'installazione è andata a buon fine!

Passo 4: Aggiorna NPM (opzionale)

Node.js viene fornito con una versione di NPM, ma potresti voler aggiornare NPM all'ultima versione disponibile.

- Nel Terminale, digita ***npm install -g npm*** e premi Invio. Questo comando aggiornerà NPM alla versione più recente disponibile.

Passo 5: Configurazione dell'ambiente (opzionale)

Potresti voler configurare alcune impostazioni dell'ambiente per migliorare la tua esperienza con NPM.

- Per esempio, puoi configurare una directory globale per i pacchetti installati da NPM. Digita ***mkdir ~/.npm-global*** nel Terminale e premi Invio.
- Configura NPM per usare questa directory: digita ***npm config set prefix '~/.npm-global'*** e premi Invio.
- Aggiungi questa directory al tuo PATH. Apri o crea il file ***~/.profile*** con un editor di testo e aggiungi la seguente riga: ***export PATH=~/.npm-global/bin:$PATH***.

- Ricarica il file di configurazione: digita source ***~/.profile*** nel Terminale e premi Invio.

Come installare TypeScript usando NPM

Una volta installato NPM, puoi facilmente installare TypeScript. Segui questi passaggi per installarlo su Windows e macOS.

Passo 1: Apri il Terminale

Su Windows premi il tasto Windows, digita "cmd" e premi Invio

Su macOS premi Command + Spazio e digita "Terminale".

Passo 2: Installa TypeScript

Nel terminale, digita il seguente comando e premi Invio:

```
npm install -g typescript
```

L'opzione -g indica che TypeScript sarà installato globalmente, rendendolo disponibile in qualsiasi progetto sul tuo sistema.

Passo 3: Verifica l'installazione

Una volta completata l'installazione, verifica che TypeScript sia stato installato correttamente digitando il seguente comando e premendo Invio:

```
tsc -v
```

Dovresti così vedere il numero di versione di TypeScript, il che significa che l'installazione è andata a buon fine.

Differenze tra Windows e macOS

In generale, i comandi per installare TypeScript sono gli stessi sia su Windows che su macOS. Tuttavia, ci sono alcune differenze nel modo in cui potresti configurare l'ambiente.

Configurazione dell'ambiente:

Su Windows, potresti dover aggiungere manualmente il percorso di NPM al PATH di sistema se non lo fa automaticamente. Questo di solito non è necessario su macOS.

Risoluzione dei Problemi

Se incontri problemi durante l'installazione, prova i seguenti suggerimenti:

Permessi Amministrativi:

- Su Windows, assicurati di eseguire il Prompt dei comandi come amministratore.
- Su macOS, potrebbe essere necessario usare sudo per alcuni comandi, ad esempio:

```
sudo npm install -g typescript
```

Ti verrà richiesta la password di amministratore.

Aggiorna NPM

Se l'installazione non riesce, potrebbe essere utile aggiornare NPM all'ultima versione usando:

```
npm install -g npm
```

IDE ed editor di testo con supporto a TypeScript

Forse non lo sapevi, ma TypeScript è stato creato da Microsoft, sicchè il fatto che **Visual Studio** sia stato il primo IDE a supportare TypeScript non dovrebbe essere una sorpresa, così come non lo è che l'editor leggero e open source di Microsoft, chiamato **Visual Studio Code** ha il supporto integrato per TypeScript.

Una volta che TypeScript ha iniziato a guadagnare popolarità, sempre più editor e IDE hanno aggiunto il supporto per questo linguaggio sia *out of the box* che attraverso plugin come anche **WebStorm** c eh sia stato creato un plugin TypeScript per **Sublime** gratuito.

NetBeans ha un plugin TypeScript che offre una varietà di funzionalità per la facilità di sviluppo e l'evidenziazione della sintassi è disponibile sia in **Vim** che in **Notepad++** con l'aiuto rispettivamente dei plugin typescript-vim e notepadplus-typescript.

È possibile visualizzare un elenco completo di tutti gli editor di testo e gli IDE che supportano TypeScript su GitHub.

Scegliere la Versione di ECMAScript da usare per la compilazione

Come immagini è possibile avere il controllo della compilazione, ovviamente potendo indicare esattamente la versione di ECMAScript che vuoi usare per compilare il tuo codice, garantendo così che sia compatibile con i browser e gli ambienti target.

Ciò si ottiene semplicemente utilizzando ***tsconfig.json*** e di seguito ti guiderò passo-passo per configurare il compilatore:

Passo 1: Creare o Aprire tsconfig.json

Creazione Automatica: Se non hai ancora un file tsconfig.json, puoi crearlo automaticamente aprendo il terminale integrato in VS Code (`Ctrl + ``) e digitando:

```
tsc --init
```

Questo comando genererà un file *tsconfig.json* con le opzioni di configurazione di base.

Passo 2: Configurare tsconfig.json per la Versione di ECMAScript

Apri il tuo file *tsconfig.json* nell'editor per specificare sia la versione di ECMAScript, modificando la proprietà target, sia usare altri valori per adattare lo strumento alle tue esigenze.

Ecco un esempio di come configurare *tsconfig.json* per diverse versioni di ECMAScript:

```
{
  "compilerOptions": {
     "target": "ES6", // oppure "ES2015", "ES2018", "ES2020", ecc.
     "module": "commonjs",
     "outDir": "./dist",
     "rootDir": "./src",
     "strict": true,
     "esModuleInterop": true,
     "skipLibCheck": true,
     "forceConsistentCasingInFileNames": true
  }
}
```

Descrizione delle singole Proprietà

- **target**: Specifica la versione di ECMAScript da utilizzare per la compilazione. Esempi di valori includono:
  - "ES5": Compatibile con la maggior parte dei browser, inclusi quelli più vecchi.
  - "ES6" o "ES2015": Introduce molte nuove funzionalità come let/const, arrow functions, classes, template literals, ecc.
  - Oppure "ES2016", "ES2017", "ES2018", "ES2019", "ES2020".
- **module**: Specifica il sistema di moduli da utilizzare. "commonjs" è comune per Node.js, ma può essere cambiato in "es6" o "esnext" per supportare i moduli ECMAScript.

- **outDir**: La directory in cui verrà posizionato il codice compilato.
- **rootDir**: La directory radice del codice sorgente TypeScript.
- **strict**: Abilita tutte le opzioni di controllo rigoroso.
- **esModuleInterop**: Abilita la compatibilità con i moduli ES.
- **skipLibCheck**: Salta il controllo dei tipi nei file dichiarativi.
- **forceConsistentCasingInFileNames**: Impone una corrispondenza coerente delle maiuscole nei nomi dei file.

Compilazione di TypeScript in JavaScript

Supponiamo che tu abbia scritto del codice TypeScript e lo abbia salvato in un file `.ts`.

I browser non saranno in grado di eseguire questo codice da soli, quindi dovrai compilare il TypeScript in un JavaScript che poi potrà essere compreso dai browser. Se utilizzi un IDE, il codice può essere compilato in JavaScript nell'IDE stesso.

Ad esempio, Visual Studio consente di compilare direttamente il codice TypeScript in JavaScript.

In pratica, dovrai creare un file **tsconfig.json** dove specifichi tutte le opzioni di configurazione per compilare il tuo file TypeScript in JavaScript.

L'approccio più amichevole per i principianti quando non si lavora su un progetto di grandi dimensioni è quello di utilizzare il terminale.

Per prima cosa, devi passare alla posizione del file TypeScript nel terminale e quindi eseguire il seguente comando.

```
cd /myTypescripts
tsc first.ts
```

Questo creerà un nuovo file chiamato *first.js* nella stessa posizione. Tieni presente che se disponi già di un file denominato *first.js*, questo verrà sovrascritto.

Se vuoi compilare tutti i file all'interno della directory corrente, puoi farlo con l'aiuto di caratteri jolly. Ricorda che questo non funzionerà in modo ricorsivo.

```
    tsc *.ts
```

Infine, puoi compilare solo alcuni file specifici fornendo esplicitamente i loro nomi in una singola riga. In questi casi, i file JavaScript verranno creati con nomi di file corrispondenti.

```
    tsc first.ts product.ts bot.ts
```

Diamo un'occhiata al seguente programma, che moltiplica due numeri in TypeScript.

```
let a: number = 12;
let b: number = 17;

function showProduct(first: number, second: number): void {
    console.log("The product is: " +  first * second);
}

showProduct(a, b);
```

La prima cosa che si nota di Typescript diversamente da Javascript, è che la variabili sono tipizzabili.

Il codice TypeScript sopra compila il seguente codice JavaScript quando la versione di destinazione è impostata su ES6.

```
let a = 12;
let b = 17;
function showProduct(first, second) {
    console.log("The product is: " + first * second);
}
showProduct(a, b);
```

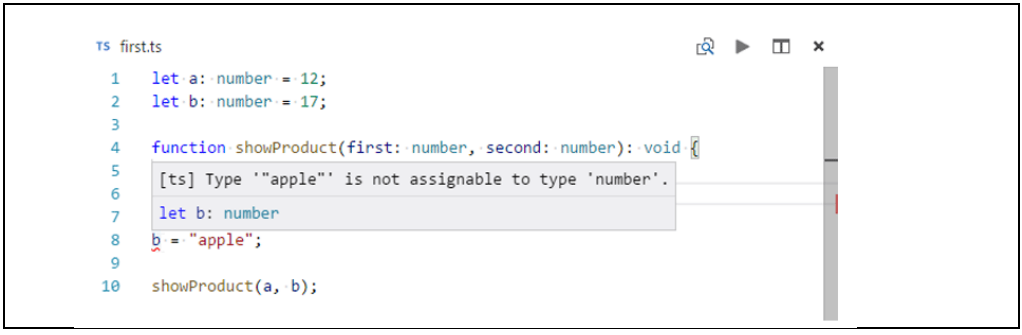
Si noti come tutte le informazioni sul tipo fornite in TypeScript non siano più presenti dopo la compilazione. In altre parole il codice compilato può essere compreso da qualsiasi browser.

Per esempio, nel codice precedente, abbiamo specificato il tipo delle variabili `a` e `b` come numeri e ciò significa che se si tenta di inserire in `b` un valore stringa, l'IDE mostrerà un errore.

Invece il codice Javascript, che permette di modificare il valore della variabile da numero a stringa, non produrrà nell'IDE un errore.

Questo tipo di aiuto è molto importante e farà la differenza quando dovrai sviluppare software in progetti ampi o complessi.

Di seguito uno screenshot che mostra il messaggio di errore in Visual Studio Code:



L'esempio sopra mostra come TypeScript continui a fornire suggerimenti su possibili errori nel codice.

Questo è stato un esempio molto semplice ma che contribuisce notevolmente a scrivere codice robusto con meno possibilità di errore.

Considerazioni finali

Questo capitolo introduttivo aveva lo scopo di darti una breve panoramica del senso e delle funzionalità di TypeScript, aiutarti nell'installazione del linguaggio insieme ad alcuni suggerimenti per gli IDE e gli editor di testo che puoi utilizzare per lo sviluppo.

successivamente abbiamo visto diversi modi per compilare il codice TypeScript in JavaScript e come TypeScript può aiutarti a evitare alcuni errori comuni.

Tipi di dati di base

Dopo aver letto il capitolo introduttivo e prima di iniziare a scrivere codice TypeScript in un IDE che lo supporta dovrai conoscere i diversi tipi di tipi di dati disponibili in TypeScript.

JavaScript dispone di sette tipi di dati diversi:

- `Null`,
- `Undefined`,
- `Boolean`,
- `Number`,
- `String`,
- `Symbol` *(introdotto in ES6)*
- `Object`.

TypeScript definisce alcuni altri tipi e tutti saranno trattati qui di seguito in dettaglio.

boolean

A differenza dei tipi di dati di `number` e di `string`, `boolean` ha solo due valori validi. È possibile impostare solo il valore su `true` o `false`.

Questi valori sono usati molto nelle strutture di controllo in cui un pezzo di codice viene eseguito se una condizione è `vera` ed un altro codice di codice viene eseguito se una condizione è `falsa`.

Ecco un esempio fondamentale di dichiarare le variabili booleane:

```
let a: boolean = true;
let b: boolean = false;
let c: boolean = 23; // Error
let d: boolean = "blue"; // Error
```

null

Proprio come in JavaScript, il tipo di dati `null` in TypeScript può avere un solo valore valido: `null`. Una variabile null non può contenere altri tipi di dati come il numero e la stringa. L'impostazione di una variabile a null cancellerà il suo contenuto se ne avesse alcuna.

Ricorda che quando il flag `strictNullChecks` è impostato su `true` in **tsconfig.json**, solo il valore `null` può essere assegnato a variabili con il tipo `null`. Questo flag è disattivato per impostazione predefinita, il che significa che puoi anche assegnare il valore `null` a variabili con altri tipi, come il `number` o il `void`.

```
// Con strictNullChecks posto a true
let a: null = null; // Ok
let b: undefined = null; // Error
let c: number = null; // Error
let d: void = null; // Error

// Con strictNullChecks posto a false
let a: null = null; // Ok
let b: undefined = null; // Ok
let c: number = null; // Ok
let d: void = null; // Ok
```

undefined

Qualsiasi variabile il cui valore non specificato è impostato su `undefined`. Tuttavia, è anche possibile impostare esplicitamente il tipo di una variabile non definita, come nell'esempio seguente.

Tieni presente che una variabile con il `tipo` impostato `undefined` può avere solo undefined come valore. Se l'opzione `strictNullChecks` è impostata su `false`, sarà anche possibile assegnare `undefined` a variabili con numeri e tipi di stringhe, ecc.

```
// Con strictNullChecks posto a true
let a: undefined = undefined; // Ok
let b: undefined = null; // Error
let c: number = undefined; // Error
let d: void = undefined; // Ok

// Con strictNullChecks posto a false
let a: undefined = undefined; // Ok
let b: undefined = null; // Ok
let c: number = undefined; // Ok
let d: void = undefined; // Ok
```

void

Il tipo di dati `void` viene utilizzato per indicare la mancanza di un `tipo` per una variabile. Impostare le variabili per avere un tipo `void` potrebbe non essere molto utile, ma è possibile impostare il tipo di ritorno delle funzioni che non restituiscono nulla per `void`. Quando viene utilizzato con variabili, il tipo `void` può avere solo due valori validi: `null` e `undefined`.

```
// Con strictNullChecks posto a true
let a: void = undefined; // Ok
let b: void = null; // Error
let c: void = 3; // Error
let d: void = "apple"; // Error

// Con strictNullChecks posto a false
let a: void = undefined; // Ok
let b: void = null; // Ok
let c: void = 3; // Error
let d: void = "apple";  // Error
```

string

Il tipo di dati di stringa viene utilizzato per memorizzare le informazioni testuali.

Sia JavaScript che TypeScript utilizzano le virgolette (") e le *virgolette singole* (') per limitare le informazioni testuali come una stringa.

Una stringa può contenere zero o più caratteri racchiusi tra virgolette.

```
// Con strictNullChecks posto a true

let a: string = undefined; // Error
let b: string = null; // Error
let c: string = "";
let d: string = "y";
let e: string = "building";
let f: string = 3; // Error
let g: string = "3";

// Con strictNullChecks posto a false
let a: string = undefined; // Ok
let b: string = null; // Ok
let c: string = "";
let d: string = "y";
let e: string = "building";
let f: string = 3; // Error
let g: string = "3";
```

TypeScript supporta anche stringhe di modelli o letterali di modello. Questi letterali di modello consentono di incorporare espressioni in una stringa.

I letterali del modello sono racchiusi dal carattere di back-tick (`) anziché da doppie virgolette e singole citazioni che racchiudono stringhe regolari.

Questa tipologia di rappresentazione di stringhe è stata introdotta in ES6. Ciò significa che otterrete un'output JavaScript diverso in base alla versione che hai selezionato.

Ecco un esempio di utilizzo di letterali di modello in TypeScript:

```
let e: string = "palazzo";
let m: number = 50;

let sentence: string = `Il ${e} di fronte alla mia finestra misura ${m} metri.`;
```

Al momento della compilazione, si avrà il seguente JavaScript:

```
//  Output in ES5
var e = "palazzo";
var m = 50;
var sentence = "Il " + e + " di fronte alla mia finestra misura " + m + "
matri.";

//  Output in ES6
let e = "palazzo";
let m = 50;
let sentence = `Il ${e} di fronte alla mia finestra misura ${m} metri.`;
```

Come si può vedere, il modello letterale è stato modificato in una stringa normale in ES5.

Questo esempio mostra come TypeScript consente di utilizzare tutte le funzioni JavaScript più recenti senza preoccuparsi della compatibilità.

number

Il tipo di dati `number` viene utilizzato per rappresentare sia valori interi sia valori a virgola mobile in JavaScript e TypeScript.

Tuttavia, è necessario ricordare che tutti i numeri sono rappresentati internamente come valori a virgola mobile. I numeri possono anche essere specificati come letterali esadecimali, ottali o binari. Tieni presente che le rappresentazioni ottali e binarie sono state introdotte in ES6, e ciò può comportare un'uscita di codice JavaScript diversa in base alla versione di destinazione.

Esistono inoltre tre valori simbolici speciali che rientrano nel `number`: `+Infinity`, `-Infinity` e `NaN`.

Ecco alcuni esempi di utilizzo del tipo di `number`.

```
// Con strictNullChecks posto a true
let a: number = undefined; // Error
let b: number = null; // Error
let c: number = 3;
let d: number = 0b111001; // Binary
let e: number = 0o436; // Octal
let f: number = 0xadf0d; // Hexadecimal
let g: number = "cat"; // Error

// Con strictNullChecks posto a false
let a: number = undefined; // Ok
let b: number = null; // Ok
let c: number = 3;
let d: number = 0b111001; // Binary
let e: number = 0o436; // Octal
let f: number = 0xadf0d; // Hexadecimal
let g: number = "cat"; // Error
```

Quando la versione di destinazione è impostata su ES6, il codice di cui sopra verrà compilato al seguente JavaScript:

```
let a = undefined;
let b = null;
let c = 3;
let d = 0b111001;
let e = 0o436;
let f = 0xadf0d;
let g = "cat";
```

Si noti che le variabili JavaScript vengono ancora dichiarate utilizzando `let`, introdotto in ES6. Inoltre, non vengono visualizzati messaggi di errore relativi al tipo di variabili diverse perché il codice JavaScript non è a conoscenza dei tipi utilizzati nel codice TypeScript.

Se la versione di destinazione è impostata su ES5, il codice TypeScript che abbiamo scritto in precedenza si compila per il seguente JavaScript:

```
var a = undefined;
var b = null;
var c = 3;
var d = 57;
var e = 286;
var f = 0xadf0d;
var g = "cat";
```

Come si può vedere, questa volta tutte le occorrenze della parola chiave `let` sono state modificate in `var`. Si noti inoltre che i numeri ottali e binari sono stati modificati nelle loro forme decimali.

enum

Il tipo di dati `enum` è presente in molti linguaggi di programmazione come C e Java, ma manca da JavaScript sicchè TypeScript ovviamente ci permette di creare e lavorare con gli `enum`, consentendo di creare una raccolta di valori correlati usando nomi mnemonici.

```
enum Animals {cat, lion, dog, cow, monkey}
let c: Animals = Animals.cat;

console.log(Animals[3]); // cow
console.log(Animals.monkey); // 4
```

Per impostazione predefinita, la numerazione di enum inizia a 0, ma è anche possibile impostare un valore diverso per i primi o gli altri membri manualmente. Questo modificherà il valore di tutti i membri che li seguono aumentando il loro valore per 1. È inoltre possibile impostare manualmente tutti i valori in un `enum`.

```
enum Animals {cat = 1, lion, dog = 11, cow, monkey}
let c: Animals = Animals.cat;
```

```
console.log(Animals[3]); // undefined
console.log(Animals.monkey); // 13
```

A differenza dell'esempio precedente, il valore degli `Animals[3]` è `undefined` questa volta. Questo perché il valore 3 sarebbe stato assegnato al cane, ma abbiamo esplicitamente impostato il suo valore su 11. Il valore per le mucche rimane a 12 e non 3 in quanto il valore dovrebbe essere superiore al valore dell'ultimo membro.

array e tuple

È possibile definire tipi di array in due modi diversi in JavaScript.

Nel primo metodo, si specifica il tipo di elementi di array seguito da `[]` che indica un array di quel tipo.

Un altro metodo consiste nell'utilizzare l'array generico `Array<elemType>`.

L'esempio seguente mostra come creare matrici con entrambi questi metodi.

Specificare `null` o `undefined` come uno degli elementi produrrà errori quando il flag `strictNullChecks` è `true`.

```
// Con strictNullChecks posto a false
let a: number[] = [1, 12, 93, 5];
let b: string[] = ["a", "apricot", "mango"];
let c: number[] = [1, "apple", "potato"]; // Error

let d: Array<number> = [null, undefined, 10, 15];
let e: Array<string> = ["pie", null, ""];

// Con strictNullChecks posto a true
let a: number[] = [1, 12, 93, 5];
let b: string[] = ["a", "apricot", "mango"];
let c: number[] = [1, "apple", "potato"]; // Error

let d: Array<number> = [null, undefined, 10, 15]; // Error
let e: Array<string> = ["pie", null, ""]; // Error
```

Il tipo di dati della tupla consente di creare una matrice in cui il tipo di un numero fisso di elementi è noto in anticipo.

Il tipo di resto degli elementi può essere solo uno dei tipi già specificati per la tupla. Ecco un esempio che lo renderà più chiaro:

```
let a: [number, string] = [11, "monday"];
let b: [number, string] = ["monday", 11]; // Error
let c: [number, string] = ["a", "monkey"]; // Error
let d: [number, string] = [105, "owl", 129, 45, "cat"];
let e: [number, string] = [13, "bat", "spiderman", 2];

e[13] = "elephant";
e[15] = false; // Error
```

Per tutte le tuple nel nostro esempio abbiamo impostato il `tipo` del primo elemento a un `numero` e il `tipo` del secondo elemento a una `stringa`.

Poiché abbiamo specificato solo un `tipo` per i primi due elementi, il resto di essi può essere una stringa o un numero. La creazione di tuple `b` e `c` comporta un errore perché abbiamo cercato di utilizzare una stringa come valore per il primo elemento quando avevamo menzionato che il primo elemento sarebbe un numero.

Allo stesso modo, non possiamo impostare il valore di un elemento di tuple su `falso` dopo aver specificato che contiene solo stringhe e numeri. Ecco perché l'ultima riga produce un errore.

I tipi Any e Never

Poniamo il caso che stai scrivendo un programma dove il valore di una variabile è determinato dagli utenti o dal codice scritto in una libreria di terze parti. In questo caso, non sarà possibile impostare correttamente il tipo di tale variabile.

La variabile potrebbe essere di qualsiasi tipo come una stringa, un numero o un booleano. Questo problema può essere risolto utilizzando il tipo `any`.

Questo è utile anche quando si creano array con elementi di tipi misti.

```
let a: any = "apple";
let b: any = 14;
let c: any = false;
let d: any[] = ["door", "kitchen", 13, false, null];

b = "people";
```

Nel codice precedente abbiamo potuto assegnare un numero a `b` e modificarlo in una stringa senza ottenere alcun errore perché il tipo `any` può accettare tutti i tipi di valori.

Il tipo `never` viene utilizzato per rappresentare valori che non si prevedono mai.

Ad esempio, è possibile assegnare `never` come tipo di ritorno di una funzione che non restituisce mai. Ciò può accadere quando una funzione genera sempre un errore o quando è bloccata in un ciclo infinito.

```
let a: never; // Ok
let b: never = false; // Error
let c: never = null; // Error
let d: never = "monday"; // Error

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

Considerazioni finali

In questo capitolo abbiamo visto tutti i tipi disponibili in TypeScript.

Abbiamo appreso come assegnare un tipo di valore diverso a una variabile mostrerà errori in TypeScript.

Questo controllo può aiutare a evitare molti errori durante la scrittura di grandi programmi. Abbiamo anche imparato come individuare diverse versioni di JavaScript

(pagina lasciata intenzionalmente bianca)

Interfacce

Nel precedente capitolo abbiamo coperto i diversi tipi di dati disponibili in TypeScript e come il loro utilizzo può aiutarti a evitare molti errori: assegnare un tipo di dati come una stringa a una particolare variabile premette a TypeScript di sapere che vuoi solo assegnargli una stringa. Basandosi su queste informazioni infatti TypeScript può segnalarti in seguito quando provi a eseguire un'operazione che non dovrebbe essere eseguita su una stringa.

In questo capitolo imparerai a conoscere le interfacce in TypeScript.

Con le interfacce, puoi fare un ulteriore passo avanti e definire la struttura o il tipo di oggetti più complessi nel tuo codice. Proprio come i tipi di variabili semplici, anche questi oggetti dovranno seguire una serie di regole create da te.

Questo può aiutarti a scrivere codice più sicuro, con minori possibilità di errore.

Creiamo la nostra prima interfaccia

Supponiamo che tu abbia un oggetto `lake` nel tuo codice, per memorizzare informazioni su alcuni dei laghi più grandi di tutto il mondo, questo oggetto avrà proprietà come il nome del lago, la sua area, lunghezza, profondità e paesi in cui quel lago è presente.

I nomi dei laghi verranno memorizzati come una stringa, le dimensioni saranno espresse in chilometri e le aree in chilometri quadrati (entrambe proprietà numeriche). Le profondità saranno in metri, e questo potrebbe anche essere una variabile di tipo `float`.

Poiché tutti questi laghi sono grandi, probabilmente le loro coste non sono limitate ad un singolo paese quindi useremo quindi una serie di stringhe per memorizzare i nomi di tutti i paesi in cui la riva del lago risiede ed infine un valore booleano può essere utilizzato per specificare se il lago è acqua salta o acqua dolce.

```
interface Lakes {
    name: string,
    area: number,
    length: number,
    depth: number,
    isFreshwater: boolean,
    countries: string[]
}
```

L'interfaccia `Lakes` contiene il tipo di ogni proprietà che utilizzeremo durante la creazione dei nostri oggetti `lake`.

Se ora proviamo ad assegnare diversi tipi di valori a una di queste proprietà, otterremo un errore, come nell'esempio seguente, che memorizza le informazioni sul nostro primo lago:

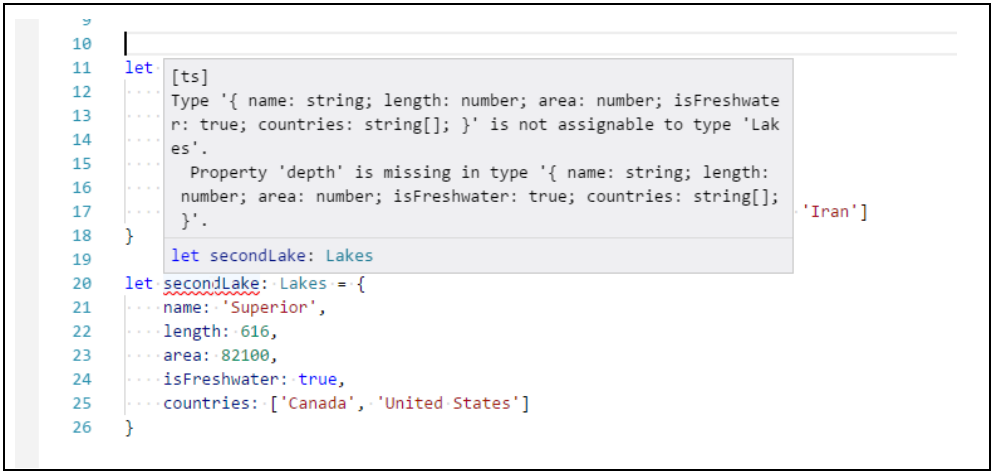
```
let firstLake: Lakes = {
    name: 'Caspian Sea',
    length: 1199,
    depth: 1025,
    area: 371000,
    isFreshwater: false,
    countries: ['Kazakhstan', 'Russia', 'Turkmenistan',
                'Azerbaijan', 'Iran']
}
```

Come puoi vedere, l'ordine in cui assegni un valore a queste proprietà non ha importanza. Tuttavia, non puoi omettere un valore: dovrai assegnare un valore a ogni proprietà per evitare errori durante la compilazione del codice, in questo modo, TypeScript si assicura di non aver perso nessuno dei valori richiesti per errore.

Ecco un esempio in cui ci siamo dimenticati di assegnare il valore della proprietà depth per un lago.

```
let firstLake: Lakes = {
    name: 'Superior',
    length: 616,
    area: 82100,
    isFreshwater: true,
    countries: ['Canada', 'United States']
}
```

Lo screenshot seguente mostra il messaggio di errore in Visual Studio Code dopo aver dimenticato di specificare la profondità e, come puoi vedere, l'errore indica chiaramente che ci siamo dimenticati di indicare il valore per la proprietà `depth` del nostro oggetto lago.



Rendere opzionali le proprietà di una interfaccia

A volte, potresti aver bisogno di una proprietà solo per alcuni oggetti specifici. Ad esempio, supponiamo di voler aggiungere una proprietà per specificare i mesi in cui un lago è ghiacciato.

Se aggiungi la proprietà direttamente all'interfaccia, come abbiamo fatto fin'ora, otterrai un errore per gli altri laghi che non si ghiacciano e quindi non hanno proprietà `frozen`.

Allo stesso modo, se aggiungi quella proprietà ai laghi che sono congelati ma non l'hai inserita nella dichiarazione dell'interfaccia, riceverai comunque un errore.

In questi casi, è possibile aggiungere un punto interrogativo (?) dopo il nome di una proprietà per far si che risulti opzionale nella dichiarazione dell'interfaccia.

In questo modo, non riceverai un errore né per le proprietà mancanti né per quelle sconosciute.

Il seguente esempio dovrebbe renderlo chiaro.

```
interface Lakes {
    name: string,
    area: number,
    length: number,
```

```
        depth: number,
        isFreshwater: boolean,
        countries: string[],
        frozen?: string[]
    }

    let primoLago: Lakes = {
        name: 'Superior',
        depth: 406.3,
        length: 616,
        area: 82100,
        isFreshwater: true,
        countries: ['Canada', 'United States']
    }

    let secondoLago: Lakes = {
        name: 'Baikal',
        depth: 1637,
        length: 636,
        area: 31500,
        isFreshwater: true,
        countries: ['Russia'],
        frozen: ['January', 'February', 'March', 'April', 'May']
    }
```

Uso degli index signatures

Le proprietà opzionali sono utili quando verranno utilizzate da alcuni dei tuoi oggetti, tuttavia, cosa succederebbe se ogni lago avesse anche il suo insieme unico di proprietà come le attività economiche, la popolazione di diversi tipi di flora e fauna che fiorisce in quel lago o gli insediamenti intorno al lago?

Aggiungere così tante proprietà diverse all'interno della dichiarazione dell'interfaccia stessa e renderle opzionali non è l'ideale.

Come soluzione, TypeScript ti consente di aggiungere proprietà extra a oggetti specifici con l'aiuto delle *index signatures*.

L'aggiunta di una *index signature* alla dichiarazione dell'interfaccia consente di specificare un numero qualsiasi di proprietà per i diversi oggetti che si stanno creando.

È necessario apportare le seguenti modifiche all'interfaccia e in questo esempio, ho utilizzato una *index signature* per aggiungere informazioni su diversi insediamenti intorno ai laghi.

Poiché ogni lago avrà i propri insediamenti, l'utilizzo di proprietà opzionali non sarebbe stata una buona idea.

```
interface Lakes {
    name: string,
    area: number,
    length: number,
    depth: number,
    isFreshwater: boolean,
    countries: string[],
    frozen?: string[],
    [extraProp: string]: any
}

let terzoLago: Lakes = {
   name: 'Tanganyika',
   depth: 1470,
   length: 676,
   area: 32600,
   isFreshwater: true,
   countries: ['Burundi', 'Tanzania', 'Zambia', 'Congo'],
   kigoma:'Tanzania',
    kalemie: 'Congo',
    bujumbura: 'Burundi'
}
```

Come altro esempio, diciamo che devi creare un gioco con diversi tipi di nemici.

Tutti questi nemici avranno alcune proprietà comuni come la dimensione dell'esercio e il loro stato di salute. Queste proprietà possono essere incluse direttamente nella dichiarazione dell'interfaccia.

Se ogni categoria di questi nemici ha un set unico di armi, quelle armi possono essere incluse con l'aiuto di una *index signature*.

Proprietà in sola lettura

Quando si lavora con oggetti diversi, potrebbe essere necessario lavorare con proprietà che dovrebbero essere modificate solo quando creiamo l'oggetto per la prima volta. Come immagini è possibile contrassegnare queste proprietà `readonly` nella dichiarazione dell'interfaccia.

Questo è simile all'uso della parola chiave `const`, ma si suppone che `const` dovrebbe essere usato con le variabili, mentre `readonly` è pensato per le proprietà.

TypeScript consente inoltre di rendere gli array di sola lettura utilizzando `ReadonlyArray <T>`.

La creazione di un array di sola lettura comporterà la rimozione di tutti i suoi *mutating methods*. Questo viene fatto per assicurarsi che non sia possibile modificare il valore dei singoli elementi in un secondo momento.

Di seguito è riportato un esempio di utilizzo di proprietà e array di sola lettura nelle dichiarazioni di interfaccia.

```
interface Enemy {
    readonly size: number,
    health: number,
    range: number,
    readonly damage: number
}

let tank: Enemy = {
    size: 50,
    health: 100,
    range: 60,
    damage: 12
}

// Questo va bene
tank.health = 95;

// Questo va in errore perchè 'damage' è in sola lettura.
tank.damage = 10;
```

Funzioni e Interfacce

È inoltre possibile utilizzare le interfacce per descrivere un tipo di funzione.

Ciò richiede di dare alla funzione una firma di chiamata con il suo elenco di parametri e il tipo restituito. È inoltre necessario fornire sia un nome che un tipo per ciascuno dei parametri.

Ecco un esempio:

```
interface EnemyHit {
    (name: Enemy, damageDone: number): number;
}

let tankHit: EnemyHit = function(tankName: Enemy, damageDone: number) {
    tankName.health -= damageDone;
    return tankName.health;
}
```

Nel superiore codice abbiamo dichiarato un'interfaccia di funzione e l'abbiamo usata per definire una funzione che sottrae al valore dello stato di salute di un carro armato il danno inflitto.

Come puoi notare, non devi usare lo stesso nome per i parametri nella dichiarazione dell'interfaccia e nella definizione affinché il codice funzioni.

Considerazioni finali

Questo capitolo ti ha presentato le interfacce e come usarle per assicurarti di scrivere codice più robusto. Ora dovresti essere in grado di creare le tue interfacce con proprietà opzionali e di sola lettura.

Hai anche imparato a usare le firme di indice per aggiungere una varietà di altre proprietà a un oggetto che non sono incluse nella dichiarazione dell'interfaccia.

Questo capitolo aveva lo scopo di farti iniziare con le interfacce in TypeScript e puoi leggere di più su questo argomento nella documentazione ufficiale.

Nel prossimo capitolo imparerai a conoscere le classi in TypeScript.

(pagina lasciata intenzionalmente bianca)

Classi

Abbiamo fatto molta strada nell'apprendimento di TypeScript dall'inizio di questo manuale, e come forse già saprai, JavaScript ha aggiunto solo di recente il supporto nativo per le classi e la programmazione orientata agli oggetti (OOP).

Introduzione alla Programmazione Orientata agli Oggetti (OOP) e TypeScript

Cos'è la Programmazione Orientata agli Oggetti (OOP)?

La Programmazione Orientata agli Oggetti (OOP) è un paradigma di programmazione che utilizza "oggetti" per rappresentare dati e metodi. Questi oggetti sono istanze di "classi", che definiscono proprietà (attributi) e comportamenti (metodi) che l'oggetto può avere. L'OOP mira a rendere il software più modulare, flessibile e facile da mantenere.

I quattro pilastri fondamentali della OOP sono:

Incapsulamento: Consente di raggruppare dati e metodi che operano su quei dati all'interno di una singola unità (classe). Questo protegge lo stato interno di un oggetto dai cambiamenti esterni incontrollati e rende il codice più comprensibile e gestibile.

Ereditarietà: Permette di creare nuove classi basate su classi esistenti. Questo meccanismo consente il riuso del codice e facilita l'estensione delle funzionalità senza duplicare il codice.

Polimorfismo: Consente a diverse classi di essere trattate come istanze della stessa classe attraverso una comune interfaccia. Questo facilita la gestione di oggetti diversi attraverso la stessa interfaccia, rendendo il codice più flessibile e estendibile.

Astrazione: Consiste nel ridurre la complessità nascondendo i dettagli implementativi e mostrando solo le funzionalità essenziali. Questo aiuta a focalizzarsi sulla logica principale del programma senza essere distratti dai dettagli.

Perché la OOP è Importante?

La OOP è fondamentale per lo sviluppo di software su larga scala per diversi motivi:

Modularità: I programmi possono essere suddivisi in unità più piccole e gestibili (classi e oggetti), facilitando la manutenzione e l'aggiornamento del software.

Riutilizzo del Codice: Grazie all'ereditarietà e alla composizione, il codice può essere riutilizzato in diverse parti dell'applicazione, riducendo la duplicazione e gli errori.

Facilità di Manutenzione: Le modifiche a una parte del sistema possono essere fatte senza influenzare altre parti, grazie all'incapsulamento.

Flessibilità e Scalabilità: Il polimorfismo e l'astrazione permettono di estendere e scalare il sistema più facilmente, adattandosi ai cambiamenti nei requisiti.

TypeScript e la OOP

JavaScript, sebbene potente e flessibile, ha alcune limitazioni quando si tratta di OOP, perché è stato iriginariamente concepito come un linguaggio di scripting utile ad aggiungere interattività alle pagine web.

JavaScript non possiede nativamente tutte le caratteristiche avanzate della OOP presenti in altri linguaggi come Java o C# ed è proprio in questo che TypeScript entra in gioco.

TypeScript è un superset di JavaScript che aggiunge il supporto per la tipizzazione statica e le funzionalità avanzate della OOP.

Ecco come TypeScript migliora JavaScript:

Classi e Interfacce: TypeScript introduce classi e interfacce, permettendo una chiara definizione delle strutture dati e dei comportamenti. Questo facilita l'organizzazione del codice in moduli ben definiti.

Tipizzazione Statica: La possibilità di definire tipi rende il codice più robusto e riduce gli errori, facilitando il refactoring e la manutenzione.

Moduli: TypeScript supporta nativamente i moduli, permettendo di suddividere il codice in unità riutilizzabili e gestibili.

Visibilità dei Membri: Con TypeScript, è possibile specificare la visibilità dei membri delle classi (pubblico, privato, protetto), migliorando l'incapsulamento e la sicurezza del codice.

Supporto per le Caratteristiche Avanzate: TypeScript supporta caratteristiche avanzate della OOP come l'ereditarietà, il polimorfismo e l'astrazione, consentendo agli sviluppatori di scrivere codice più elegante e mantenibile.

Riuso del codice

Uno dei maggiori benefici della programmazione ad oggetti è il riuso del codice. Attraverso l'ereditarietà, il codice realizzato per far funzionare una classe, può essere riutilizzato anche per creare nuove classi che "estendono" altre classi.

In pratica una classe può ***ereditare*** le proprietà e i metodi di una classe esistente senza che sia necessario duplicare il (e duplicare la manutenzione del) codice.

Questo non solo riduce la quantità di codice necessario, ma assicura anche la coerenza e facilita la manutenzione.

Ad esempio, una classe base può definire metodi comuni per la gestione dei dati, mentre le classi derivate possono estendere queste funzionalità con specifiche implementazioni.

Questo approccio modulare consente di costruire sistemi complessi in modo incrementale, riutilizzando e adattando il codice esistente.

Esempio

Immagina di dover sviluppare un software per gestire un parco zoologico.

Nel tuo sistema, ci sono diversi tipi di animali, ognuno con le proprie caratteristiche e comportamenti, tuttavia molte caratteristiche del "animale" sono comuni e possono essere riutilizzate.

La Superclasse "Animale"

Inizi a progettare il sistema definendo una classe base chiamata "Animale".

Questa classe rappresenta concettualmente un animale generico e include proprietà e metodi comuni a tutti gli animali.

Ad esempio, ogni animale ha un nome, un'età, un metodo per muoversi, un sistema di alimentazione e un metodo per la replicazione.

Questi attributi e comportamenti comuni sono definiti nella superclasse "Animale".

Le Sottoclassi "Mammifero" e "Uccello"

Ora, crei delle sottoclassi per rappresentare specifici gruppi di animali.

Ad esempio, potresti creare una classe "Mammifero" e una classe "Uccello".

Queste sottoclassi ereditano tutte le proprietà e i metodi della superclasse "Animale".

In altre parole, un "Mammifero" e un "Uccello" hanno un nome, un'età e possono muoversi, proprio come un animale generico.

Specializzazione delle Sottoclassi

Oltre alle caratteristiche ereditate dalla classe "Animale", le sottoclassi "Mammifero" e "Uccello" possono avere proprie proprietà e metodi specifici.

Ad esempio, la classe "Mammifero" può includere un metodo per allattare i piccoli, mentre la classe "Uccello" può avere un metodo per volare.

Queste caratteristiche aggiuntive sono specifiche del tipo di animale e non sono presenti nella classe base "Animale".

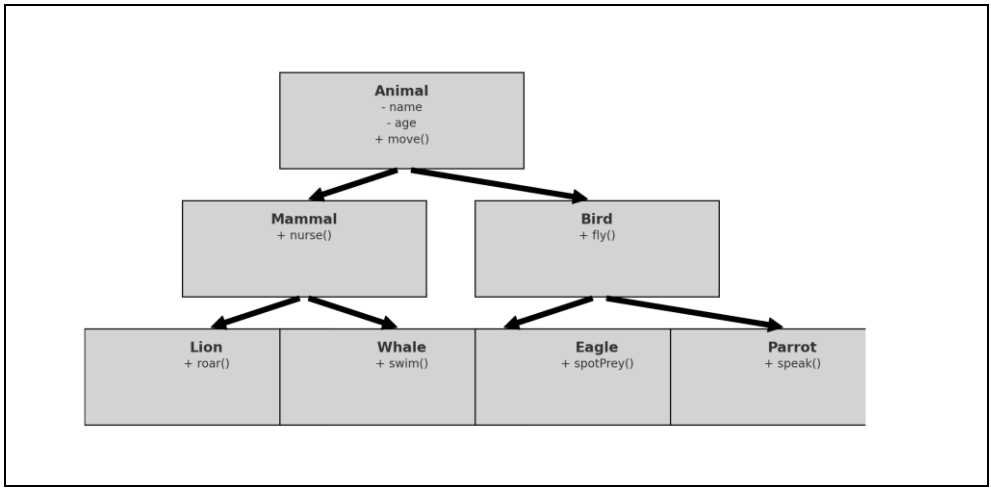
Ulteriori Sottoclassi Specifiche

Per rappresentare ancora più specificatamente gli animali nel parco zoologico, puoi creare ulteriori sottoclassi.

Ad esempio, potresti avere una classe "Leone" che estende "Mammifero" e una classe "Aquila" che estende "Uccello".

La classe "Leone" erediterà tutte le proprietà e i metodi di "Animale" e "Mammifero", aggiungendo magari un metodo per ruggire.

Allo stesso modo, la classe "Aquila" erediterà le caratteristiche di "Animale" e "Uccello", aggiungendo un metodo per avvistare la preda.



Ecco lo schema che rappresenta l'esempio delle superclassi e delle classi derivate:

1. Animal (Superclasse)
   - Proprietà: name, age
   - Metodo: move()
2. Mammal (Sottoclasse di Animal)
   - Metodo: nurse()
3. Bird (Sottoclasse di Animal)
   - Metodo: fly()
4. Lion (Sottoclasse di Mammal)
   - Metodo: roar()
5. Whale (Sottoclasse di Mammal)
   - Metodo: swim()
6. Eagle (Sottoclasse di Bird)
   - Metodo: spotPrey()
7. Parrot (Sottoclasse di Bird)

- Metodo: speak()

Questo schema mostra come le classi derivate ereditano le proprietà e i metodi della superclasse e aggiungono le loro caratteristiche specifiche. L'ereditarietà consente il riuso del codice, garantendo coerenza e facilitando la manutenzione.

La Programmazione Orientata agli Oggetti, combinata con le potenti funzionalità di TypeScript, offre un modo strutturato e efficiente per sviluppare applicazioni JavaScript.

L'incapsulamento, l'ereditarietà, il polimorfismo e l'astrazione, insieme alla tipizzazione statica di TypeScript, migliorano significativamente la qualità del codice, facilitano il riuso e rendono la manutenzione del software più semplice e sicura.

Adottando TypeScript per la OOP, gli sviluppatori possono sfruttare al meglio le potenzialità di JavaScript per creare applicazioni scalabili e robuste.

TypeScript ha consentito agli sviluppatori di utilizzare questo tipo di programmazione da molto più tempo, compilandolo poi in JavaScript che funzionerà su tutti i principali browser.

Adesso vedremo le classi in TypeScript, che sono simili alle loro controparti ES6 ma più rigorose.

Crea la tua prima classe

Cominciamo con le basi, le classi sono una parte fondamentale della programmazione orientata agli oggetti e si usano per rappresentare qualsiasi entità che abbia alcune proprietà e alcune funzioni che possono agire su determinate proprietà.

TypeScript ti dà il pieno controllo sulle proprietà e le funzioni che sono accessibili all'interno e all'esterno della propria classe contenitore.

Ecco un esempio molto semplice della creazione di una classe Person.

```
class Person {
    name: string;
    constructor(theName: string) {
        this.name = theName;
    }
    introduceSelf() {
        console.log("Hi, I am " + this.name + "!");
    }
}

let myPerson = new Person("Sally");
```

```
myPerson.introduceSelf();
```

Il codice superiore è relativo ad una *classe* molto semplice chiamata `Person`, una *classe* che ha una proprietà chiamata `name` e una funzione chiamata `introduceSelf` ed ha anche un costruttore, che è (anche fondamentalmente) una funzione. Tuttavia, i costruttori sono speciali perché vengono chiamati ogni volta che creiamo una nuova *istanza* della nostra *classe*.

È inoltre possibile passare parametri ai costruttori per inizializzare proprietà diverse e nel nostro caso, stiamo usando il costruttore per inizializzare il nome della persona che stiamo creando usando la *classe* `Person`.

La funzione `introduceSelf` è un metodo della *classe* `Person` e qui viene utilizzata per visualizzare il nome della persona sulla console.

Tutte queste *proprietà*, *metodi* e il *costruttore* di una *classe* sono chiamati collettivamente membri della *classe* ma bisogna tenere presente che la *classe* `Person` non crea automaticamente un oggetto *persona*, la classe agisce più come un progetto con tutte le informazioni sugli attributi che un oggetto *persona* dovrebbe avere una volta creati.

Con questo in mente, abbiamo creato una nuova *persona* (abbiamo ottenuto una nuova *istanza* della classe `Person`) e l'abbiamo chiamata `myPerson`. Durante la creazione ci viene chiesto di passare un parametro, sicchè abbiamo deciso di dare come valore alla proprietà `nome` dell'*istanza* e la stringa "`Sally`". A questo punto possiamo utilizzare il metodo `introduceSelf()` che visualizzerà, nella console, la riga "`Ciao, sono Sally!`".

Modificatori pubblici e privati

Nella sezione precedente, abbiamo creato una *istanza* della classe `Person` e le abbiamo dato nome "Sally". Vediamo adess come fare a cambiare il *nome* della *persona* da Sally a Mindy, in qualsiasi punto del nostro codice:

```
let P = new Person("Sally");
P.introduceSelf();  // Visualizzerà Hi, I am Sally!

P.name = "Mindy";
P.introduceSelf();  // Visualizzerà Hi, I am Mindy!
```

Avrai notato che siamo stati in grado di utilizzare sia la proprietà `name` sia il metodo `introduceSelf` al di fuori della classe contenente, questo perché tutti i membri di una classe in TypeScript sono definiti *pubblici* (utilizzabili da chiunque) per impostazione predefinita.

È necessario dire che è comunque possibile, anche per migliorare la lettura del codice, specificare esplicitamente che una proprietà o un metodo è pubblico aggiungendo la parola chiave `public` prima di esso, se è quello che vogliamo.

A volte però potrebbe succedere che non vogliamo che una proprietà o un metodo sia accessibile al di fuori della sua classe contenente e in questo caso possiamo usare la parola chiave `private` che, come è facilemnte immaginabile, rende privati i membri così segnalati.

Se nel codice precedente rendi privata la proprietà name per impedire che venga modificata al di fuori della classe contenitore, TypeScript ti mostrerà un errore con il quale ti ricorda che la proprietà `name` è privata e puoi accedervi solo all'interno della classe `Person`.

Lo screenshot seguente mostra l'errore in Visual Studio Code.

Ereditarietà in TypeScript

L'ereditarietà è un concetto fondamentale della programmazione orientata agli ogetti (OOP) che permette di creare nuove classi basate su classi esistenti.

Questo meccanismo è fondamentale nel promuovere il riuso del codice, che è alla base della programmazione orientata agli oggetti (OOP) e comprendere come funziona l'ereditarietà ti aiuterà a scrivere codice più organizzato e mantenibile, sfruttando al massimo le potenzialità di TypeScript e rende la progettazione di software più organizzata e manutenibile, che poi è il motivo stesso per cui TypeScript è stato pensato.

L'ereditarietà in TypeScript è implementata in modo simile a come viene gestita in altri linguaggi OOP come Java e C#, sicchè comprenderne il senso aiuta il programmatore in tutti i casi in cui si parla di ereditarietà degli oggetti, per questo motivo espando il capitolo spiegandone il concetto e l'uso delle parole chiave ***extends*** e ***super***.

Cos'è l'Ereditarietà?

L'ereditarietà consente a una classe (chiamata "sottoclasse" o "classe derivata") di ereditare le proprietà e i metodi di un'altra classe (chiamata "superclasse" o "classe base"), in modo da creare una gerarchia di classi che condividono funzionalità comuni, ma che possono anche avere comportamenti specifici.

L'ereditarietà quindi, permette di creare classi più complesse partendo da una classe di base.

Per spiegarlo meglio userò degli esempi: possiamo utilizzare la classe `Person` della sezione precedente come base per creare una classe `Friend` che avrà tutti i membri della Persona e aggiungerà alcune caratteristiche proprie che la differenziano da `Person`, e allo stesso modo, potresti anche aggiungere una classe `Teacher`, tutti erediteranno i metodi e le proprietà della classe `Person` aggiungendo però nuovi alcuni metodi e proprietà che li distinguono.

Nel seguente esempio dove ho riportato anche il codice per la classe `Person` qui in modo da poter confrontare facilmente il codice sia della classe base che della classe derivata, dovrebbe renderlo più chiaro

```
class Person {
    private name: string;
    constructor(theName: string) {
        this.name = theName;
    }
    introduceSelf() {
        console.log("Hi, I am " + this.name + "!");
    }
}

class Friend extends Person {
```

```
        yearsKnown: number;
        constructor(name: string, yearsKnown: number) {
            super(name);
            this.yearsKnown = yearsKnown;
        }
        timeKnown() {
            console.log("We have been friends for " + this.yearsKnown +
                        " years.")
        }
    }

    let friendA = new Friend("Jacob", 6);

    friendA.introduceSelf();
    // Visualizza: Hi, I am Jacob!

    friendA.timeKnown();
    // Visualizza: We have been friends for 6 years.
```

Come puoi vedere, devi usare la parola chiave **`extends`** per la classe `Friend` per ereditare tutti i membri della classe `Person`.

È importante ricordare che il costruttore di una classe derivata deve sempre invocare il costruttore della classe base con una chiamata a `super()`.

Potresti aver notato che il costruttore di `Friend` non aveva bisogno di avere lo stesso numero di parametri della classe base, tuttavia, il primo parametro name è stato passato a `super()` per richiamare il costruttore del genitore, che accettava anche un parametro.

Infine, non abbiamo dovuto ridefinire la funzione `introduceSelf` all'interno della classe Friend **perché è stata ereditata** dalla classe `Person` da cui è derivata!

Utilizzo del modificatore protect

Fino a questo punto, abbiamo solo reso i membri di una classe privati o pubblici: renderli pubblici ci consente di accedervi da qualsiasi *ambiente*, renderli privati ne limita l'uso al solo codice della classe contenente.

Ci sono volte che potresti volere che i membri di una classe base siano accessibili solo all'interno di tutte le *classi derivate*, e questo è reso possibile dal modificatore `protect`, ossia per limitare l'accesso di un membro solo alle classi derivate.

Si può inoltre utilizzare la parola chiave `protected` con il costruttore di una classe base, per impedire a chiunque di creare un'istanza di quella classe. Questo ovviamente non limiterà di estendere le classi basate su questa classe di base.

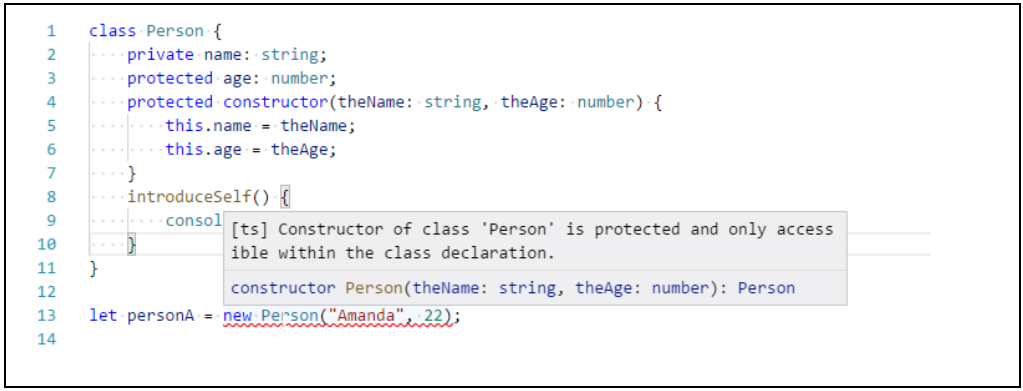
```
class Person {
    private name: string;
    protected age: number;
    protected constructor(theName: string, theAge: number) {
        this.name = theName;
        this.age = theAge;
    }
    introduceSelf() {
        console.log("Hi, I am " + this.name + "!");
    }
}

class Friend extends Person {
    yearsKnown: number;
    constructor(name: string, age: number, yearsKnown: number) {
        super(name, age);
        this.yearsKnown = yearsKnown;
    }
    timeKnown() {
        console.log("We have been friends for " + this.yearsKnown +
                    " years.")
    }
    friendSince() {
        let firstAge = this.age - this.yearsKnown;
        console.log(`We have been friends since I was ${firstAge}
                     years old.`)
    }
}

let friendA = new Friend("William", 19, 8);
friendA.friendSince();
// Visualizza: We have been friends since I was 11 years old.
```

Nel codice qui sopra, puoi vedere che abbiamo protetto la proprietà dell'età (`protected age: number;`) per impedire l'uso dell'età al di fuori di qualsiasi classe derivata da `Person`, ma abbiamo anche usato la parola chiave `protected` per il costruttore della classe `Person`, farlo significa che non saremo più in grado di istanziare direttamente la classe `Person`.

Lo screenshot seguente mostra un errore che viene visualizzato durante il tentativo di istanziare una classe con il costruttore protetto.



Considerazioni finali

In questo capitolo, ho cercato di coprire le basi delle classi in TypeScript.

Abbiamo iniziato creando una classe `Person` molto semplice che ha visualizzato il nome della persona sulla console.

Successivamente, hai appreso la parola chiave `private`, che può essere utilizzata per impedire l'accesso ai membri di una classe in qualsiasi punto arbitrario del programma.

Infine, hai imparato come estendere classi diverse nel tuo codice usando una classe base con ereditarietà.

C'è molto di più che puoi imparare sulle classi nella documentazione ufficiale.

(pagina lasciata intenzionalmente bianca)

Generics

Fino a qui ci si è concentrati sui tipi di dati di base disponibili in TypeScript.

Il controllo del tipo in TypeScript ci consente di assicurarci che le variabili nel nostro codice possano avere solo tipi specifici di valori assegnati.

In questo modo possiamo evitare molti errori durante la scrittura del codice perché l'IDE sarà in grado di dirci quando stiamo eseguendo un'operazione su un tipo che non supporta.

Questo rende il controllo del tipo una delle migliori caratteristiche di TypeScript.

Adesso ci concentreremo su un'altra importante caratteristica di questo linguaggio: i tipi generici.

Con i *generics*, TypeScript ti consente di scrivere codice che può agire su una varietà di tipi di dati invece di essere limitato a uno solo.

Imparerai in dettaglio la necessità di generici e come è meglio che usare semplicemente qualsiasi tipo di dati disponibile in TypeScript.

La necessità dei generics

Se non hai familiarità con i `generics`, ti starai chiedendo perché ne abbiamo bisogno… Cominciamo quindi scrivendo una funzione che restituirà un elemento casuale da un array di numeri.

```
function randomIntElem(theArray: number[]): number {
    let randomIndex = Math.floor(Math.random()*theArray.length);
    return theArray[randomIndex];
}

let positions: number[] = [103, 458, 472, 458];
let randomPosition: number = randomIntElem(positions);
```

La funzione `randomElem` che abbiamo appena definito accetta un array di numeri come unico parametro e anche il tipo di ritorno della funzione è stato specificato come numero.

Stiamo usando la funzione `Math.random()` per restituire un numero casuale in virgola mobile compreso tra 0 e 1.

Moltiplicandolo per la lunghezza di un dato array e chiamando `Math.floor()` sul risultato si ottiene un indice casuale.

Una volta ottenuto l'indice casuale, restituiamo l'elemento a quell'indice specifico.

Qualche tempo dopo, diciamo che ha la necessità di ottenere un elemento stringa casuale da un array di stringhe, a questo punto, potresti decidere di creare un'altra funzione che ha come target le stringhe.

```
function randomStrElem(theArray: string[]): string {
    let randomIndex = Math.floor(Math.random()*theArray.length);
    return theArray[randomIndex];
}

let colors: string[] = ['violet', 'indigo', 'blue', 'green'];
let randomColor: string = randomStrElem(colors);
```

E se fosse necessario selezionare un elemento casuale da un array di un'interfaccia definita? La creazione di una nuova funzione ogni volta che si desidera ottenere un elemento casuale da un array di diversi tipi di oggetti non è una buona pratica.

Una soluzione per questo problema è impostare il tipo di parametro di matrice passato alle funzioni come `any[]`. In questo modo puoi scrivere la tua funzione solo una volta e questa poi funzionerà con un *array* di qualsiasi *tipo*.

```
function randomElem(theArray: any[]): any {
    let randomIndex = Math.floor(Math.random()*theArray.length);
    return theArray[randomIndex];
}

let positions = [103, 458, 472, 458];
let randomPosition = randomElem(positions);

let colors = ['violet', 'indigo', 'blue', 'green'];
let randomColor = randomElem(colors);
```

Come si può vedere, possiamo usare la funzione sopra per ottenere posizioni casuali e colori casuali. Uno dei problemi principali di questa soluzione è la perdita delle informazioni sul tipo di valore restituito.

In precedenza, eravamo sicuri che `randomPosition` fosse un numero e `randomColor` una stringa e questo ci ha aiutato a utilizzare questi valori di conseguenza.

Ora, tutto ciò che sappiamo è che l'elemento restituito potrebbe essere di qualsiasi tipo. Nel codice sopra, potremmo specificare il tipo di `randomColor` come un `number` e ancora non ottenere alcun errore.

```
// Questo codice compilerà senza alcun errore.

let colors: string[] = ['violet', 'indigo', 'blue', 'green'];

let randomColor: number = randomElem(colors);
```

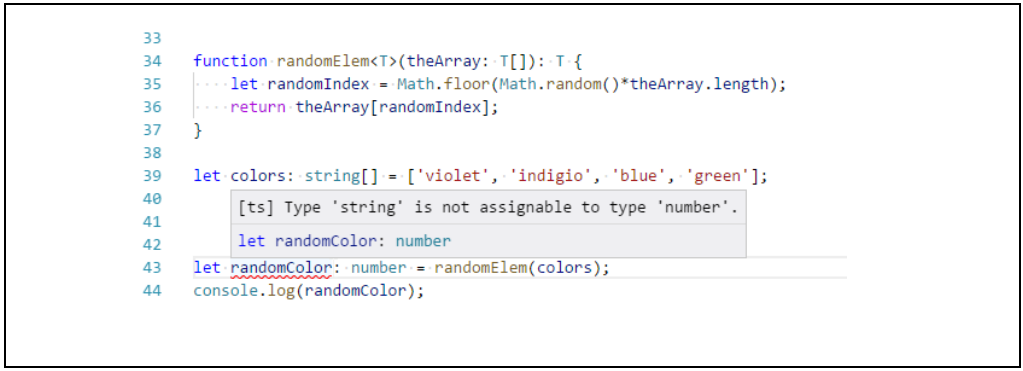
Una soluzione migliore per evitare la duplicazione del codice pur preservando le informazioni sul tipo consiste nell'usare i *generics*. Ecco una funzione generica che restituisce elementi casuali da un array.

```
function randomElem<T>(theArray: T[]): T {
    let randomIndex = Math.floor(Math.random()*theArray.length);
    return theArray[randomIndex];
}

let colors: string[] = ['violet', 'indigo', 'blue', 'green'];

let randomColor: string = randomElem(colors);
```

Ora però riceverò un errore se proverò a cambiare il tipo di `randomColor` da `string` a `number`. Ciò dimostra che l'uso dei *generics* è molto più sicuro rispetto all'utilizzo di tipi `any` in tali situazioni.



L'uso di generics può essere limitante

Nella sezione precedente abbiamo discusso come è possibile utilizzare i *generics* invece di qualsiasi *tipo* per scrivere una singola funzione ed evitare di sacrificare i vantaggi del controllo del tipo.

Un problema con la funzione generica che abbiamo scritto nella sezione precedente è che TypeScript non ci consente di eseguire molte operazioni sulle variabili passate.

Questo perché TypeScript non può fare alcuna supposizione sul tipo di variabile che verrà passato in anticipo alla nostra funzione generica.

Di conseguenza, la nostra funzione generica può utilizzare solo quelle operazioni applicabili su tutti i *tipi* di dati.

Il seguente esempio dovrebbe rendere più chiaro questo concetto.

```
function removeChar(theString: string, theChar: string): string {
    let theRegex = new RegExp(theChar, "gi");
    return theString.replace(theRegex, '');
}
```

La funzione precedente rimuoverà tutte le occorrenze del carattere specificato dalla stringa data.

Potresti voler creare una versione generica di questa funzione in modo da poter rimuovere anche cifre specifiche da un dato numero e caratteri da una stringa.

Ecco la corrispondente funzione generica.

```
function removeIt<T>(theInput: T, theIt: string): T {
    let theRegex = new RegExp(theIt, "gi");
    return theInput.replace(theRegex, '');
}
```

La funzione `removeChar` non ha mostrato un errore. Tuttavia, se usi la `replace` all'interno di `removeIt`, TypeScript ti dirà che la `replace` non esiste per il *tipo* "T". Questo perché TypeScript non può più presumere che `theInput` sarà una stringa.

Questa restrizione sull'utilizzo di metodi diversi in una funzione generica potrebbe indurti a pensare che il concetto di generici non sarà di grande utilità, dopotutto.

Non c'è davvero molto che puoi fare con una manciata di metodi che devono essere applicabili a tutti i tipi di dati per poterli utilizzare all'interno di una funzione generica.

Una cosa importante da ricordare a questo punto è che in genere non è necessario creare funzioni che verranno utilizzate con tutti i tipi di dati, è più comune creare una funzione che verrà utilizzata con un set o un intervallo specifico di tipi di dati.

Questo vincolo sui tipi di dati rende le funzioni generiche molto più utili.

Creare funzioni generics utilizzando vincoli

La funzione `removeIt` generica della sezione precedente ha mostrato un errore perché il metodo di `replace` al suo interno è pensato per essere utilizzato con le stringhe, mentre i parametri ad essa passati potrebbero avere qualsiasi tipo di dati.

È possibile utilizzare la parola chiave `extends` per vincolare i tipi di dati passati a una funzione generica in TypeScript.

Tuttavia, `extends` è limitato solo alle interfacce e alle classi. Ciò significa che la maggior parte delle funzioni generiche create avrà parametri che estendono un'interfaccia o una classe di base.

Ecco una funzione generica che visualizza il nome di persone, familiari o celebrità che le sono state passate.

```
interface People {
    name: string
}

interface Family {
    name: string,
    age: number,
    relation: string
}

interface Celebrity extends People {
    profession: string
}

function printName<T extends People>(theInput: T): void {
    console.log(`My name is ${theInput.name}`);
}

let serena: Celebrity = {
    name: 'Serena Williams',
    profession: 'Tennis Player'
}

printName(serena);
```

Nell'esempio precedente, abbiamo definito tre interfacce e ciascuna di esse ha una proprietà name e la funzione `printName` generica che abbiamo creato accetterà qualsiasi oggetto che estende `People`.

In altre parole, puoi passare una famiglia o un oggetto di celebrità a questa funzione e il suo nome verrà visualizzato senza problemi.

È possibile definire molte più interfacce e, a condizione che abbiano una proprietà `name`, sarà possibile utilizzare la funzione `printName` senza alcun problema.

Questo era un esempio molto semplice, ma puoi creare funzioni generiche più utili una volta che ti senti più a tuo agio con l'intero processo.

Ad esempio, è possibile creare una funzione generica che calcola il valore totale di diversi articoli venduti in un dato mese purché ogni articolo abbia una proprietà `prezzo` per memorizzare il prezzo e una proprietà `venduta` che memorizza il numero di articoli venduti.

Utilizzando i *generics*, sarai in grado di utilizzare la stessa funzione purché gli elementi estendano la stessa interfaccia o classe.

Considerazioni finali

In questo capitolo ho cercato di coprire le basi dei generici in TypeScript in modo adatto ai principianti.

Abbiamo iniziato discutendo della necessità dei *generics* e successivamente abbiamo visto il modo giusto di utilizzarli per evitare la duplicazione del codice senza sacrificare la capacità di controllo del tipo.

Dopo aver compreso le basi discusse qui, puoi leggere di più sui *generics* nella documentazione ufficiale (*https://www.typescriptlang.org/docs/handbook/generics.html*)

(pagina lasciata intenzionalmente bianca)

Moduli in TypeScript

Introduzione ai Moduli

I moduli in TypeScript sono una parte fondamentale per organizzare il codice in unità riutilizzabili e mantenibili. Essi permettono di suddividere il codice in file distinti e di gestire le dipendenze in modo chiaro e conciso. In questo capitolo, esploreremo i concetti di base dei moduli, come esportare e importare moduli, l'utilizzo delle parole chiave export e import, e la differenza tra moduli interni ed esterni.

Esportazione e Importazione di Moduli

Per utilizzare il codice di un modulo in un altro, è necessario esportarlo dal modulo di origine e importarlo nel modulo di destinazione. TypeScript utilizza le parole chiave export e import per gestire questa operazione.

Utilizzo delle Parole Chiave export e import

Esportazione

Esistono due modi principali per esportare elementi da un modulo: esportazioni nominate ed esportazioni di default.

Esportazioni Nominate:

Permettono di esportare più elementi da un modulo.

```
//mathUtils.ts
export function add(x: number, y: number): number {
        return x + y;
}
export function subtract(x: number, y: number): number {
       return x - y;
}
   export const PI = 3.14;
```

Esportazione di Default:

Permettono di esportare un singolo valore, funzione o classe come esportazione predefinita del modulo.

```
// calculator.ts
export default class Calculator {
    add(x: number, y: number): number {
        return x + y;
    }
    subtract(x: number, y: number): number {
        return x - y;
    }
}
```

Importazione

Per importare gli elementi esportati da un modulo, utilizziamo la parola chiave import.

Importazioni Nominate:

Specifichiamo esattamente quali elementi importare.

```
// app.ts
import { add, subtract, PI } from './mathUtils';

console.log(add(5, 3)); // Output: 8

console.log(subtract(5, 3)); // Output: 2

console.log(PI); // Output: 3.14
```

Importazione di Default:

Importa l'elemento predefinito esportato da un modulo.

```
// main.ts
import Calculator from './calculator';
const calculator = new Calculator();

console.log(calculator.add(5, 3)); // Output: 8

console.log(calculator.subtract(5, 3)); // Output: 2
```

Importazioni Nominate con Alias:

È possibile rinominare gli elementi importati utilizzando as.

```
// app.ts
import {add as addition,subtract as subtraction} from './mathUtils';

console.log(addition(5, 3)); // Output: 8

console.log(subtraction(5, 3)); // Output: 2
```

Importazione di Tutti gli Elementi:

È possibile importare tutti gli elementi di un modulo in un oggetto.

```
// app.ts
import * as MathUtils from './mathUtils';

console.log(MathUtils.add(5, 3)); // Output: 8

console.log(MathUtils.subtract(5, 3)); // Output: 2

console.log(MathUtils.PI); // Output: 3.14
```

Moduli Interni ed Esterni

Moduli Interni

I moduli interni, chiamati anche namespace, erano utilizzati nelle versioni precedenti di TypeScript per raggruppare il codice, tuttavia, con l'introduzione dei moduli ES6, l'uso è diventato meno comune e generalmente sconsigliato.

```
// shapes.ts
namespace Shapes {
    export class Circle {
        constructor(public radius: number) {}

        area(): number {
            return Math.PI * this.radius * this.radius;
        }
    }

    export class Square {
        constructor(public side: number) {}

        area(): number {
            return this.side * this.side;
        }
    }
}

// app.ts
/// <reference path='shapes.ts' />

const circle = new Shapes.Circle(5);
console.log(circle.area()); // Output: 78.53981633974483

const square = new Shapes.Square(10);
console.log(square.area()); // Output: 100
```

Moduli Esterni

I moduli esterni seguono lo standard ES6 e sono il metodo preferito per organizzare il codice moderno in TypeScript. Utilizzano le parole chiave import ed export per gestire le dipendenze tra i file.

```
// shapes/circle.ts
export class Circle {
    constructor(public radius: number) {}

    area(): number {
        return Math.PI * this.radius * this.radius;
    }
}

// shapes/square.ts
export class Square {
    constructor(public side: number) {}

    area(): number {
        return this.side * this.side;
    }
}

// app.ts
import { Circle } from './shapes/circle';
import { Square } from './shapes/square';

const circle = new Circle(5);
console.log(circle.area()); // Output: 78.53981633974483

const square = new Square(10);
console.log(square.area()); // Output: 100
```

Conclusione

I moduli in TypeScript attraverso le parole chiave export e import forniscono un modo potente e flessibile per organizzare il codice, premettendo di suddividere il codice in file distinti e gestire le dipendenze in modo chiaro e conciso, così come accade per i moderni linguaggi di programmazione.

Come detto l'adozione dei moduli esterni seguendo lo standard ES6 è il metodo preferito per il codice moderno, mentre l'uso dei namespace è generalmente sconsigliato.

Con una buona comprensione dei moduli, sarà possibile creare applicazioni TypeScript modulari, riutilizzabili e facili da mantenere.

Namespace in TypeScript

I namespace in TypeScript sono una funzionalità utilizzata per raggruppare funzioni, variabili, interfacce e classi correlate in un unico spazio di nomi.

I namespace aiutano a organizzare il codice in modo logico e a prevenire conflitti di nomi globali, specialmente in progetti di grandi dimensioni.

Sono simili ai moduli, ma hanno un utilizzo e una gestione leggermente diversi.

Creazione e Utilizzo dei Namespace

Per creare un namespace in TypeScript, utilizziamo la parola chiave namespace seguita dal nome del namespace.

All'interno del namespace, possiamo dichiarare funzioni, classi, interfacce e variabili che saranno accessibili utilizzando il nome del namespace come prefisso.

Esempio di Creazione e Utilizzo di un Namespace

```
// shapes.ts
namespace Shapes {
    export class Circle {
        constructor(public radius: number) {}

        area(): number {
            return Math.PI * this.radius * this.radius;
        }
    }
    export class Square {
        constructor(public side: number) {}

        area(): number {
            return this.side * this.side;
        }
    }
}

// app.ts
/// <reference path='shapes.ts' />
```

```
const circle = new Shapes.Circle(5);
console.log(circle.area());
// Output: 78.53981633974483

const square = new Shapes.Square(10);
console.log(square.area());
// Output: 100
```

Differenze tra Namespace e Moduli

Sebbene i namespace e i moduli in TypeScript siano simili nel loro scopo di organizzare e strutturare il codice, ci sono alcune differenze chiave tra loro:

Scopo e Utilizzo: I namespace sono generalmente utilizzati per raggruppare codice correlato all'interno di un singolo file o di più file, mentre i moduli sono utilizzati per organizzare il codice in unità riutilizzabili e mantenibili che possono essere importate ed esportate tra diversi file.

Sintassi di Dichiarazione: I namespace sono dichiarati utilizzando la parola chiave namespace, mentre i moduli utilizzano la sintassi di importazione ed esportazione di ES6 con le parole chiave import ed export.

Caricamento del Codice: I namespace sono parte del codice globale e non richiedono un sistema di caricamento del modulo. I moduli, d'altra parte, richiedono un sistema di caricamento del modulo come CommonJS, AMD o ES6 Modules.

Compatibilità: I namespace sono più compatibili con il codice JavaScript legacy, mentre i moduli sono preferiti per il codice moderno e seguono lo standard ES6.

Conclusioni

In questo capitolo, abbiamo esplorato i namespace in TypeScript, una funzionalità utile per organizzare il codice e prevenire conflitti di nomi.

Abbiamo visto come creare e utilizzare i namespace, nonché le differenze tra namespace e moduli. Mentre i namespace sono utili per il codice legacy e per raggruppare funzionalità correlate, i moduli sono la scelta preferita per il codice moderno in TypeScript.

Comprendere quando utilizzare namespace e moduli è fondamentale per scrivere codice ben organizzato e mantenibile.

Decoratori in TypeScript

I decoratori in TypeScript sono una funzionalità sperimentale che permette di annotare e modificare classi, metodi, proprietà e parametri.

Forniscono un modo potente per aggiungere metadati e modificare il comportamento degli elementi a cui sono applicat e sono ampiamente utilizzati nei framework moderni, come Angular, per semplificare e organizzare il codice.

Decoratori di Classe

I decoratori di classe vengono applicati alla dichiarazione di una classe. Un decoratore di classe riceve un parametro, che è il costruttore della classe, e può essere utilizzato per osservare, modificare o sostituire una definizione di classe.

Esempio di decoratore di classe:

```
@sealed
function sealed(constructor: Function) {
    Object.seal(constructor);
    Object.seal(constructor.prototype);
}

@sealed
class Greeter {
    constructor(public greeting: string) {}
    greet() {
        return 'Hello, ' + this.greeting;
    }
}
```

Decoratori di Metodo

I decoratori di metodo sono quelli che vengono applicati ai metodi di una classe. Un decoratore di metodo riceve tre parametri: il prototipo della classe, il nome del metodo e il descrittore delle proprietà del metodo.

Esempio di decoratore di metodo:

```
function enumerable(value: boolean) {
    return function (target: any, propertyKey: string, descriptor:
        PropertyDescriptor) {
            descriptor.enumerable = value;
        };
}



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

    @enumerable(false)
    greet() {
        return 'Hello, ' + this.greeting;
    }
}
```

Decoratori di Proprietà

I decoratori di proprietà vengono applicati alle proprietà di una classe. Un decoratore di proprietà riceve due parametri: il prototipo della classe e il nome della proprietà.

Esempio di decoratore di proprietà:

```
function format(target: any, propertyKey: string) {
    let _val = target[propertyKey];

    const getter = () => _val;
    const setter = (newVal) => {
        _val = newVal.toUpperCase();
    };

    Object.defineProperty(target, propertyKey, {
        get: getter,
```

```
            set: setter,
            enumerable: true,
            configurable: true,
        });
    }

    class Greeter {
        @format
        greeting: string;
        constructor(message: string) {
            this.greeting = message;
        }
        greet() {
            return 'Hello, ' + this.greeting;
        }
    }
```

Conclusioni

In questo capitolo, abbiamo esplorato i decoratori in TypeScript, una potente funzionalità che permette di aggiungere metadati e di modificare il comportamento di classi, metodi e proprietà. Abbiamo

visto come creare e applicare decoratori di classe, di metodo e di proprietà, e come questi possono semplificare e organizzare il codice, specialmente in contesti di sviluppo complessi come quelli dei framework moderni.

I decoratori sono uno strumento potente, ma devono essere usati con attenzione. È importante comprendere come e quando utilizzarli per evitare di complicare eccessivamente il codice. In conclusione, l'uso dei decoratori può migliorare notevolmente la modularità e la leggibilità del codice, rendendo lo sviluppo di applicazioni TypeScript più efficiente e mantenibile.

(pagina lasciata intenzionalmente bianca)

Mixin in TypeScript

I mixin sono un modo flessibile per condividere funzionalità tra classi in TypeScript. A differenza dell'ereditarietà tradizionale, che permette a una classe di estendere un'altra classe, i mixin consentono di combinare più comportamenti da diverse classi in una singola classe. Questo approccio permette una maggiore modularità e riutilizzo del codice.

Implementazione dei Mixin in TypeScript

Per implementare i mixin in TypeScript, possiamo creare una funzione che accetta una classe base e restituisce una nuova classe che estende la classe base con funzionalità aggiuntive. Questo modello di progettazione è noto come 'funzione mixin', e qui di seguito un esempio:

```
function Jumpable(Base: new () => {}) {
    return class extends Base {
        jump() {
            console.log('Jumping');
        }
    };
}
function Speakable(Base: new () => {}) {
    return class extends Base {
        speak() {
            console.log('Speaking');
        }
    };
}
class Person {
    constructor(public name: string) {}
}

const JumpablePerson = Jumpable(Person);
const SpeakablePerson = Speakable(Person);
const MultiTalentedPerson = Jumpable(Speakable(Person));
const person = new MultiTalentedPerson('John');
person.jump(); // Output: Jumping
person.speak(); // Output: Speaking
```

Utilizzo dei Mixin per la Composizione di Classi

I mixin sono particolarmente utili per la composizione di classi, permettendo di combinare comportamenti da diverse sorgenti in una singola classe. Questo può essere estremamente utile per evitare la duplicazione del codice e per creare classi più modulari e manutenibili.

Esempio di Utilizzo dei Mixin per la Composizione di Classi

```
class FlyingPerson extends Jumpable(Speakable(Person)) {
    fly() {
        console.log('Flying');
    }
}

const flyingPerson = new FlyingPerson('Jane');
flyingPerson.jump(); // Output: Jumping
flyingPerson.speak(); // Output: Speaking
flyingPerson.fly(); // Output: Flying
```

Conclusioni

In questo capitolo, abbiamo esplorato i mixin in TypeScript, un potente strumento per la condivisione di funzionalità tra classi.

Abbiamo visto come implementare i mixin e come utilizzarli per la composizione di classi, consentendo una maggiore modularità e riutilizzo del codice.

I mixin sono un'ottima alternativa all'ereditarietà tradizionale, offrendo maggiore flessibilità nella progettazione delle applicazioni.

Introduzione ai Tipi Utility

TypeScript offre una vasta gamma di tipi utility che consentono di manipolare e trasformare i tipi esistenti in modi molto utili e potenti.

Questi tipi utility sono strumenti essenziali per creare software manutenibile, leggibile e robusto.

Essi permettono di applicare concetti avanzati di programmazione orientata agli oggetti (OOP) e di garantire una maggiore sicurezza e flessibilità nel codice.

Manutenibilità

Uno dei principali vantaggi dei tipi utility è la loro capacità di migliorare la manutenibilità del codice. Quando si sviluppa un software complesso, è fondamentale poter effettuare modifiche e aggiornamenti senza introdurre bug o compromettere il funzionamento del sistema.

I tipi utility permettono di definire tipi derivati che possono adattarsi facilmente alle evoluzioni del progetto, riducendo la necessità di modifiche pervasive.

```
interface User {
  id: number;
  name: string;
  email: string;
}

type PartialUser = Partial<User>;
```

Leggibilità

I tipi utility migliorano anche la leggibilità del codice. Utilizzando tipi derivati chiari e ben definiti, è possibile esprimere in modo conciso e preciso l'intento del codice.

Questo rende più facile comprendere e navigare il codice, sia per gli sviluppatori che lo scrivono sia per quelli che lo mantengono in futuro.

```
type ReadonlyUser = Readonly<User>;
```

Robustezza

La robustezza del codice è un altro beneficio chiave dei tipi utility. Garantendo che le trasformazioni sui tipi siano corrette e coerenti, i tipi utility aiutano a prevenire errori e a migliorare la sicurezza del tipo.

Questo significa che il compilatore può rilevare più errori durante la fase di sviluppo, riducendo il numero di bug che raggiungono la fase di produzione.

Flessibilità

I tipi utility forniscono una grande flessibilità nella definizione, permettendo di creare tipi specifici per qualsiasi particolare necessità, senza dover riscrivere interamente le definizioni dei tipi esistenti.

Questo è essenziale per adattare il codice a nuove esigenze e contesti senza compromettere la qualità del software.

```
type UserNameAndEmail = Pick<User, 'name' | 'email'>;
```

Incapsulamento

L'incapsulamento è uno dei principi fondamentali della programmazione orientata agli oggetti (OOP) e i tipi utility supportano l'incapsulamento consentendo di controllare l'accessibilità e la mutabilità delle proprietà degli oggetti.

Ad esempio, utilizzando il tipo `Readonly`, è possibile garantire che certe proprietà di un oggetto non possano essere modificate, proteggendo così lo stato interno dell'oggetto.

```
interface Config {
  apiUrl: string;
  timeout: number;
}

type ReadonlyConfig = Readonly<Config>;

const config: ReadonlyConfig = {
  apiUrl: 'https://api.example.com',
  timeout: 5000
};

config.apiUrl = 'https://api.newexample.com'; // Errore: readonly
```

Ereditarietà

L'ereditarietà è un altro principio cardine dell'OOP, e tipi utility come `Partial` e `Required` possono essere utilizzati per creare versioni modificate di tipi esistenti, supportando così l'ereditarietà e il riutilizzo del codice.

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

type CompleteUser = Required<User>;
```

Polimorfismo

I tipi utility come `Extract` e `Exclude` possono essere utilizzati per creare tipi polimorfici che rappresentano sottoinsiemi o sovrainsiemi di tipi esistenti.

```
type StringOrNumber = string | number;
type ExtractedType = Extract<StringOrNumber, string>; // string
```

Composizione

La composizione è un altro principio OOP che consiste nel costruire oggetti complessi aggregando oggetti più semplici. Fanno parte di questi i tipi utility come `Record` possono essere utilizzati per definire tipi di oggetti composti in modo chiaro e conciso.

```
type UserRoles = 'admin' | 'user' | 'guest';

type UserPermissions = Record<UserRoles, boolean>;

const permissions: UserPermissions = {
  admin: true,
  user: true,
  guest: false
};
```

Esplorazione dei Tipi Utility

Come detto, i tipi utility di TypeScript sono strumenti essenziali per la creazione di software manutenibile, leggibile e robusto che offrono una grande flessibilità nella definizione dei tipi e migliorano la sicurezza del codice, permettendo di evitare molti errori comuni.

Vediamo di seguito nel dettaglio i tipi utility disponibili:

Partial

Il tipo Partial rende tutte le proprietà di un tipo opzionali ed è utile quando si vuole creare una versione di un tipo in cui alcune proprietà possono mancare.

Ad esempio, se si ha un tipo User con proprietà obbligatorie e si vuole creare una funzione che aggiorna solo alcune di queste proprietà, si può usare Partial per permettere di omettere le altre.

```
interface User {
    id: number;
    name: string;
    age: number;
}

const updateUser = (id: number, user: Partial<User>) => {
    // Funzione per aggiornare un utente
};
updateUser(1, { name: 'Alice' });
```

Readonly

Il tipo Readonly rende tutte le proprietà di un tipo immutabili per creare, ad esempio, oggetti che non devono essere modificati dopo la loro creazione.

Dovendo garantire che un oggetto di configurazione rimanga costante durante l'esecuzione del programma, si può usare Readonly, come di seguito:

```
interface User {
    id: number;
    name: string;
    age: number;
```

```
}


const user: Readonly<User> = {
    id: 1,
    name: 'Alice',
    age: 30
};

// user.age = 31; // Errore: la proprietà è di sola lettura
```

Record

Il tipo Record crea un tipo con un insieme di proprietà chiave-valore.

È utile quando si vuole creare un oggetto con chiavi dinamiche. Ad esempio, se si vuole creare un oggetto che mappa gli ID degli utenti alle informazioni sugli utenti:

```
interface User {
    id: number;
    name: string;
    age: number;
}

const users: Record<string, User> = {
    'user1': { id: 1, name: 'Alice', age: 30 },
    'user2': { id: 2, name: 'Bob', age: 25 }
};
```

Pick

Il tipo Pick crea un nuovo tipo selezionando un sottoinsieme delle proprietà di un tipo esistente. È utile per estrarre solo le proprietà necessarie, come nel caso in cui si ha un tipo `User` con molte proprietà ma si vuole creare un tipo che include solo il nome e l'email:

```
interface User {
    id: number;
    name: string;
    age: number;
    email: string;
```

```
}

type UserContactInfo = Pick<User, 'name' | 'email'>;


const contactInfo: UserContactInfo = {
    name: 'Alice',
    email: 'alice@example.com'
};
```

Omit

Il tipo Omit crea un nuovo tipo escludendone un sottoinsieme delle proprietà. Si usa quando serve rimuovere proprietà non necessarie.

Ad esempio, se si ha un tipo User con molte proprietà ma si vuole creare un tipo che esclude l'email:

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

type UserWithoutEmail = Omit<User, 'email'>;

const user: UserWithoutEmail = {
    id: 1,
    name: 'Alice',
    age: 30
};
```

Required

Il tipo Required per garantire che tutte le proprietà di un tipo siano sempre presenti. Se si vuole creare una versione in cui tutte le proprietà sono obbligatorie, si veda l'esempio seguente:

```
interface User {
    id?: number;
```

```
    name: string;
    age?: number;
}



const user: Required<User> = {
 id: 1,
 name: 'Alice',
 age: 30
};
```

Exclude

Il tipo `Exclude` in TypeScript è utile quando si desidera rimuovere uno o più tipi da un'unione di tipi.

Sembra complicato ma ad esempio, se hai un'unione di tipi che include `string`, `number`, e `Function`, e vuoi creare un tipo che escluda `Function`, puoi usare `Exclude`.

Questo è particolarmente utile per garantire che il tuo codice non accetti tipi indesiderati.

È una funzionalità potente per migliorare la tipizzazione e garantire che le variabili e i parametri abbiano solo i tipi desiderati e rendere il tuo codice più sicuro e specifico, riducendo il rischio di errori:

```
type T0 = Exclude<"a" | "b" | "c", "a">; // "b" | "c"
type T1 = Exclude<string | number | (() => void), Function>; //
string | number
```

Extract

Il tipo `Extract` in TypeScript è utile quando si desidera ottenere solo i tipi comuni tra due insiemi di tipi.

Ad esempio, se hai un'unione di tipi che include `string` e `number`, e vuoi creare un tipo che include solo i tipi che sono anche presenti in un altro insieme, puoi usare `Extract` per filtrare e ottenere solo i tipi specifici che ti interessano.

Utilizzando `Extract`, puoi creare tipi più precisi e specifici, garantendo che il tuo codice lavori solo con i tipi desiderati e rendere il tuo codice più robusto e sicuro.

```
type T0 = Extract<"a" | "b" | "c", "a" | "f">; // "a"
type T1 = Extract<string | number | (() => void), Function>; //
() => void
```

NonNullable

Il tipo NonNullable rimuove null e undefined da un tipo e si usa per creare un tipo che non ammette null e undefined.

Ad esempio, se si ha un tipo che include string, number e null, e si vuole creare un tipo che esclude null, si può usare NonNullable.

```
type T0 = NonNullable<string | number | undefined>; // string | number
type T1 = NonNullable<string[] | null | undefined>; // string[]
```

Parameters

Il tipo `Parameters` in TypeScript è utilizzato per estrarre i tipi dei parametri di una funzione e rappresentarli come una tupla (una struttura dati simile a un array, ma con un numero e un tipo fissi di elementi).

Utilizzando `Parameters`, puoi ottenere i tipi dei parametri di una funzione e riutilizzarli altrove ad esempio, se hai una funzione con parametri specifici e vuoi creare un nuovo tipo che rappresenti questi parametri.

Questo è utile per garantire la coerenza dei tipi tra diverse parti del codice e per evitare la duplicazione della definizione dei tipi dei parametri:

```
type T0 = Parameters<() => string>; // []
type T1 = Parameters<(s: string) => void>; // [string]
type T2 = Parameters<<T>(arg: T) => T>; // [unknown]
```

ConstructorParameters

Il tipo ConstructorParameters estrae i tipi dei parametri di un costruttore come tupla.

Si usa per ottenere i tipi dei parametri di un costruttore e riutilizzarli altrove come ad esempio avendo una classe e volendo creare un nuovo tipo che rappresenta i suoi parametri del costruttore:

```
type T0 = ConstructorParameters<ErrorConstructor>; //[message?: string]
type T1 = ConstructorParameters<FunctionConstructor>; //string[]
type T2 = ConstructorParameters<RegExpConstructor>;
                // [pattern: string | RegExp, flags?: string]
```

ReturnType

Il tipo ReturnType estrae il tipo di ritorno di una funzione e server ad ottenere il tipo di ritorno di una funzione per riutilizzarlo altrove.

Ad esempio, se si ha una funzione e si vuole creare un nuovo tipo che rappresenta il suo tipo di ritorno:

```
type T0 = ReturnType<() => string>; // string
type T1 = ReturnType<(s: string) => void>; // void
type T2 = ReturnType<<T>() => T>; // T
```

InstanceType

Il tipo InstanceType estrae il tipo di istanza di una classe o di una funzione costruttore per riutilizzarlo altrove.

Ad esempio, se si ha una classe e si vuole creare un nuovo tipo che rappresenta il tipo delle sue istanze:

```
class C {
 constructor(public s: string) {}
}
type T0 = InstanceType<typeof C>; // C
type T1 = InstanceType<any>; // any
type T2 = InstanceType<never>; // never
```

ThisType

Il tipo `ThisType` in TypeScript è un tipo marker che non crea un tipo concreto da solo, ma viene utilizzato per migliorare il supporto del controllo dei tipi nei contesti in cui `this` è ampiamente utilizzato. È particolarmente utile quando si lavora con oggetti che utilizzano una

sintassi fluente, dove i metodi restituiscono `this` per permettere l'aggancio di più chiamate di metodo in una singola espressione.

Ad esempio, consideriamo un oggetto builder che costruisce una configurazione complessa. Con `ThisType`, possiamo definire metodi che restituiscono `this` con il corretto tipo, garantendo così che il compilatore TypeScript sappia quale tipo è `this` in ogni metodo.

```
interface Config {
  width: number;
  height: number;
}

const configBuilder = {
  width: 0,
  height: 0,
  setWidth(width: number) {
    this.width = width;
    return this;
  },
  setHeight(height: number) {
    this.height = height;
    return this;
  },
  build(): Config {
    return {
      width: this.width,
      height: this.height
    };
  }
} as ThisType<Config>;

const config =
         configBuilder.setWidth(100).setHeight(200).build();

console.log(config); // Output: { width: 100, height: 200 }
```

In questo esempio, `ThisType` aiuta TypeScript a capire che `this` nei metodi `setWidth` e `setHeight` si riferisce all'oggetto `configBuilder`. Questo permette di concatenare chiamate di metodo (`setWidth(100).setHeight(200)`) in modo fluente e sicuro, con il supporto completo del controllo dei tipi.

Senza `ThisType`, TypeScript potrebbe non essere in grado di determinare correttamente il tipo di `this`, portando a errori o alla perdita dei benefici del controllo dei tipi. Utilizzando `ThisType`, si garantisce che i metodi restituiscano il corretto tipo di `this`, migliorando la robustezza e la manutenibilità del codice.

Node utility types

Alcuni tipi utility sono specifici per l'ambiente di Node.js e possono non essere applicabili in altri contesti. È importante essere consapevoli delle differenze tra i vari ambienti di esecuzione quando si utilizzano questi tipi utility.

Come Utilizzare i Tipi Utility per Migliorare il Codice

I tipi utility di TypeScript permettono di scrivere codice più flessibile e riutilizzabile. Utilizzando tipi come Partial, Readonly, Record, Pick e Omit, è possibile creare tipi derivati in modo dichiarativo, riducendo la duplicazione del codice e migliorando la manutenibilità.

Ad esempio, con Partial, è possibile creare facilmente funzioni di aggiornamento per oggetti complessi senza dover specificare tutte le proprietà.

Readonly aiuta a prevenire modifiche accidentali ad oggetti immutabili, migliorando la sicurezza del codice. Record è utile per mappare chiavi a valori dinamici, mentre Pick e Omit permettono di creare tipi più precisi e focalizzati.

(pagina lasciata intenzionalmente bianca)

Tipizzazione Avanzata

Tipi unione e intersezione

I tipi unione e intersezione in TypeScript sono strumenti che aiutano a combinare più tipi in modi flessibili, adattandoli alle esigenze di programmazione.

Per esempio un *tipo unione* consente a una variabile di essere di uno tra più tipi specificati, cioè se hai una variabile che può essere sia un numero che una stringa, puoi definire un tipo unione che li includa entrambi. Ma a che serve? E' certamente utile quando si lavora con dati che possono avere più forme o quando si desidera scrivere funzioni che possono accettare diversi tipi di input. In pratica, una variabile con un tipo unione può contenere un valore che appartiene a uno qualsiasi dei tipi inclusi nell'unione.

Un *tipo intersezione* d'altra parte combina più tipi in un singolo tipo che deve soddisfare tutte le caratteristiche dei tipi specificati. Significa che una variabile con un tipo intersezione deve avere tutte le proprietà e metodi di tutti i tipi inclusi nell'intersezione e si usa quando si desidera creare un nuovo tipo che eredita le proprietà di altri tipi, garantendo che l'oggetto finale abbia tutte le caratteristiche richieste.

I tipi unione e intersezione aumentano la flessibilità del codice, consentendo di definire con precisione quali tipi sono accettabili in diverse situazioni per esempio, si possono gestire casi in cui i dati possono variare, e coi tipi intersezione creare tipi complessi che combinano più aspetti.

Questi concetti sono fondamentali per scrivere codice TypeScript robusto e adattabile, permettendo di gestire una varietà di scenari con maggiore sicurezza e chiarezza.

Esempio di tipo unione

Nell'esempio vediamo come ad `id` possa essere assegnato sia un numero che una stringa. In questo caso infatti il tipo unione viene dichiarato utilizzando il simbolo `|` (or), che permette alla variabile di assumere *uno di questi tipi*. Se si tenta di assegnare un valore di tipo booleano, verrà genato un errore, in modo che `id` possa essere solo un numero o una stringa.

```
let id: number | string;
id = 10;   // Ok
id = "10"; // Ok
id = true; // Errore
```

Esempio di tipo intersezione

Nell'esempio si combinano due interfacce `User` e `Admin` per creare un nuovo tipo `AdminUser`. Il nuovo tipo includerà tutte le proprietà di entrambe le interfacce, infatti l'oggetto `admin` creato dall'`AdminUser` possiede le proprietà `id`, `name` e `role`, dimostrando come un tipo intersezione richieda la presenza di tutte le proprietà definite nei tipi combinati:

```
interface User {
    id: number;
    name: string;
}

interface Admin {
    role: string;
}

type AdminUser = User & Admin;

const admin: AdminUser = {
    id: 1,
    name: 'Alice',
    role: 'Administrator'
};
```

Tipi letterali

I tipi letterali permettono di specificare un insieme limitato di valori che una variabile può assumere e possono essere utilizzati per definire stringhe, numeri o booleani specifici.

Esempio di tipo letterale

Nell'esempio è mostrato come definire un tipo `CardinalDirection` che può assumere solo uno dei quattro valori: 'North', 'East', 'South' o 'West'. Assegnare uno qualsiasi dei valori non consentiti, ma qualsiasi altro valore, come 'north' con la 'n' minuscola, genererà un errore, garantendo che `direction` sia sempre uno dei valori specificati.

```
type CardinalDirection = 'North' | 'East' | 'South' | 'West';
let direction: CardinalDirection;
direction = 'North'; // Ok
direction = 'north'; // Errore
```

Tipi discriminati

I tipi discriminati sono una tecnica per gestire in modo sicuro e tipizzato l'unione di tipi, spesso utilizzata con le unioni discriminanti. Essi si basano su una proprietà comune (discriminante) che può assumere valori diversi, consentendo di identificare univocamente il tipo di unione.

In questo esempio il tipo discriminato utilizza le interfacce `Square` e `Rectangle`, ciascuna con una proprietà discriminante `kind`. La funzione `area` accetta un parametro di tipo `Shape`, un'unione di `Square` e `Rectangle`.

Utilizzando un `switch` sulla proprietà `kind`, TypeScript può determinare in modo sicuro il tipo specifico e calcolare l'area correttamente, dimostrando la sicurezza e chiarezza fornite dai tipi discriminati.

```
interface Square {
    kind: 'square';
    size: number;
}
interface Rectangle {
    kind: 'rectangle';
    width: number;
    height: number;
}

type Shape = Square | Rectangle;

function area(shape: Shape) {
    switch (shape.kind) {
        case 'square':
            return shape.size * shape.size;
        case 'rectangle':
            return shape.width * shape.height;
    }
}
```

Condizionali sui tipi

Attraverso i tipi condizionali si possono definire tipi che dipendono da una condizione logica, utilizzano la sintassi condizionale (T extends U ? X : Y) per scegliere un tipo in base al risultato della condizione.

Nell'esempio viene definito il tipo `` `IsString<T>` ``, che utilizza un tipo condizionale per valutare se `` `T` `` estende `` `string` ``. Questo tipo condizionale impiega la sintassi dell'operatore ternario, che è una forma concisa di espressione condizionale comunemente utilizzata in molti

linguaggi di programmazione. La sintassi generale dell'operatore ternario è `` `condizione ? valoreSeVero : valoreSeFalso` ``. Nel contesto dei tipi condizionali in TypeScript, questa espressione verifica una condizione sui tipi e restituisce uno di due possibili risultati in base all'esito della verifica.

Nel caso di `IsString<T>`, l'operatore ternario verifica se `T` estende il tipo `string`. Se la condizione `T extends string` è vera, il tipo condizionale restituisce `yes`; altrimenti, restituisce `no`. Questo tipo di verifica è estremamente utile per creare tipi che si adattano dinamicamente in base ai tipi su cui sono applicati.

```
type IsString<T> = T extends string ? 'yes' : 'no';

type A = IsString<string>;  // 'yes'
type B = IsString<number>;  // 'no'
```

Nell'esempio, `IsString<string>` restituisce `yes` perché `string` estende `string`, mentre `IsString<number>` restituisce `no` perché `number` non estende `string`. Questo semplice ma potente meccanismo consente di creare logiche di tipo flessibili e dinamiche, che possono reagire e adattarsi a diversi tipi in modo automatico.

L'uso dei tipi condizionali è particolarmente utile quando si desidera implementare comportamenti diversi basati sui tipi dei dati, come ad esempio sei si volesse creare una funzione che accetti parametri di tipi diversi e restituendo risultati differenti a seconda del tipo del parametro.

I tipi condizionali sono un potente strumento nel toolkit di TypeScript, che permettono di scrivere codice più preciso e adattabile, facilitando la gestione di scenari complessi, riducendo il rischio di errori di tipo e migliorando l'esperienza di sviluppo complessiva.

Per un programmatore che si avvicina a TypeScript, comprendere e saper utilizzare i tipi condizionali è fondamentale per sfruttare appieno le potenzialità del linguaggio e scrivere codice di alta qualità.

Conclusioni

La tipizzazione avanzata in TypeScript offre potenti strumenti per definire tipi complessi e flessibili, migliorando la sicurezza e la manutenzione del codice. Utilizzare tipi unione, intersezione, letterali,

discriminati e condizionali permette di scrivere codice più robusto e chiaro, adattandosi a diverse esigenze e scenari di programmazione.

(pagina lasciata intenzionalmente bianca)

Manipolazione dei Tipi

Utilizzo di keyof, typeof, in e as

TypeScript offre diverse parole chiave potenti per la manipolazione dei tipi.

Queste includono 'keyof' per ottenere le chiavi di un tipo, 'typeof' per ottenere il tipo di un'espressione, 'in' per iterare sulle chiavi, e 'as' per l'asserzione dei tipi.

Esempio di keyof

Vediamo come ottenere le chiavi di un tipo sotto forma di un'unione di stringhe.

Definendo l'interfaccia `User` con le proprietà `id`, `name` e `age`, il tipo `UserKeys` diventa un'unione di stringhe 'id' | 'name' | 'age' e ciò consente di lavorare in modo tipizzato con le chiavi di un oggetto, migliorando la sicurezza del codice.

```
interface User {
    id: number;
    name: string;
    age: number;
}

type UserKeys = keyof User; // 'id' | 'name' | 'age'
```

Esempio di typeof

Ti illustro come ottenere il tipo di un'espressione: creando un oggetto `user` con proprietà `id`, `name` e `age`, il tipo `UserType` viene derivato utilizzando `typeof user`.

Questo permette di riutilizzare i tipi di oggetti esistenti senza doverli ridefinire manualmente, facilitando la manutenzione del codice.

```
let user = {
    id: 1,
    name: 'Alice',
    age: 30
};

type UserType = typeof user; /* { id: number;
```

```
                                    name: string, age: number; } */
```

Esempio di in

Di seguito vedrai come iterare sulle chiavi di un tipo per costruire un nuovo tipo.

Definendo `Keys` come un'unione di 'id', 'name' e 'age', il tipo `UserPartial` viene creato utilizzando una mappatura di tipi con `in`.

Questo tipo diventa

`{ id?: string; name?: string; age?: string; }`

permettendo di generare tipi basati su chiavi esistenti.

```
type Keys = 'id' | 'name' | 'age';

type UserPartial = {
    [K in Keys]?: string;
};

// UserPartial diventa { id?: string; name?: string; age?: string; }
```

Esempio di as

L'esempio mostra come eseguire un'asserzione di tipo assegnando un valore di tipo `any` alla variabile `value`.

Viene utilizzata l'asserzione `as string` per indicare a TypeScript che `value` dovrebbe essere trattato come una stringa e ciò permette di accedere alle proprietà e ai metodi specifici del tipo asserito, migliorando la sicurezza e la precisione del codice.

```
let value: any = "Hello, TypeScript";

let strLength: number = (value as string).length;
                          // 'as' per asserzione di tipo
```

Inferenza di tipi

TypeScript è in grado di dedurre automaticamente i tipi in molte situazioni, riducendo la necessità di annotazioni esplicite.

L'inferenza di tipi rende il codice più conciso e leggibile, mantenendo al contempo la sicurezza tipica.

Esempio di inferenza di tipi

L'esempio di inferenza di tipi evidenzia come TypeScript possa dedurre automaticamente i tipi dichiarando la variabile `x` e assegnandole il valore 3.

In questo caso TypeScript inferisce che `x` è di tipo `number` e analogamente, l'oggetto `user` viene inferito come tipo `{ id: number; name: string; }`.

L'inferenza di tipi riduce la necessità di annotazioni esplicite, mantenendo il codice conciso e leggibile.

```
let x = 3; // TypeScript deduce automaticamente che x è di tipo number

const user = {
    id: 1,
    name: 'Alice'
};

// TypeScript deduce che user è di tipo { id: number; name: string; }
```

Conclusioni

La manipolazione dei tipi in TypeScript attraverso 'keyof', 'typeof', 'in' e 'as', insieme all'inferenza di tipi, offre potenti strumenti per creare codice flessibile e sicuro.

Comprendere e utilizzare queste funzionalità permette di scrivere codice più chiaro, conciso e manutenibile, sfruttando appieno le capacità di TypeScript.

(pagina lasciata intenzionalmente bianca)

Funzioni Avanzate in TypeScript

Overloading delle funzioni

L'overloading delle funzioni permette di definire più firme per una singola funzione, consentendo di gestire diversi tipi di input e output, rendendo il codice più flessibile e robusto e migliorando la capacità di riutilizzo delle funzioni.

Esempio di overloading delle funzioni

Vediamo come definire più firme per una singola funzione.

La funzione `add` può accettare due numeri e restituire la loro somma oppure accettare due stringhe e restituire la loro concatenazione.

L'implementazione effettiva della funzione utilizza `any` per gestire entrambi i tipi e ciò permette alla funzione di essere versatile e di gestire diversi tipi di input, migliorando la flessibilità del codice.

```
function add(a: number, b: number): number;
function add(a: string, b: string): string;
function add(a: any, b: any): any {
    return a + b;
}

const result1 = add(1, 2); // 3
const result2 = add("Hello, ", "World!"); // "Hello, World!"
```

Funzioni Asincrone e Promises

Passiamo adesso ad affrontare il concetto di chiamata asincrona, una funzionalità davvero fondamentale per la moderna programmazione perchè permette di gestire operazioni che richiedono tempo, come il ciclo rappresentato da una richiesta HTTP, senza che l'attesa dell'esecuzione blocchi l'esecuzione dell'intero programma.

In JavaScript e TypeScript, ci sono vari modi per gestire l'asincronia, tra cui le callback, le Promises e le funzioni asincrone (async/await).

Callback

Una callback è una funzione passata come argomento a un'altra funzione, che viene eseguita dopo che un'operazione asincrona è completata.

Bisogna fare attenzione all'uso eccessivo di callback perchè ciò può portare ad un fenomeno noto come 'callback hell', che ti esporrò alla fine di questo capitolo, che si rende evidente quando si nidificano trope callback.

Esempio di Callback

La funzione `fetchData` che accetta una callback come argomento. La funzione `fetchData` utilizza `setTimeout` per simulare una chiamata asincrona, richiamando la callback con i dati "Dati ricevuti" dopo un secondo.

La callback viene eseguita una volta completata l'operazione asincrona, e il risultato viene stampato nella console.

Questo esempio dimostra come le callback possono essere utilizzate per gestire operazioni asincrone, ma evidenzia anche la potenziale complessità e il rischio di annidamento delle callback (vedi 'callback hell' alla fine del capitolo).

```
function fetchData(callback: (data: string) => void) {
    setTimeout(() => {
        callback("Dati ricevuti");
    }, 1000);
}

fetchData((data) => {
    console.log(data); // "Dati ricevuti"
});
```

Promises

Una Promise è un oggetto che rappresenta l'eventuale completamento o fallimento di un'operazione asincrona e il suo valore risultante. Una Promise può essere in uno di tre stati:

- pending (in attesa),
- fulfilled (completata con successo)
- rejected (fallita).

Esempio di Promises

L'esempio di Promises mostra una funzione `fetchData` che restituisce una Promise all'interno della quale, viene utilizzato `setTimeout` per simulare un'operazione asincrona.

La Promise viene risolta un secondo dopo con "Dati ricevuti".

La funzione `fetchData` viene poi chiamata, e il metodo `then` viene utilizzato per gestire il valore risolto della Promise, stampando i dati nella console e il metodo `catch` è utilizzato per gestire eventuali errori.

Questo esempio evidenzia come le Promises possono migliorare la leggibilità e la gestione degli errori rispetto alle callback.

```
function fetchData(): Promise<string> {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Dati ricevuti");
        }, 1000);
    });
}

fetchData().then((data) => {
    console.log(data); // "Dati ricevuti"
}).catch((error) => {
    console.error(error);
});
```

Funzioni Asincrone (async/await)

Le funzioni asincrone, dichiarate con la parola chiave 'async', restituiscono sempre una Promise.

La parola chiave 'await' può essere usata all'interno delle funzioni asincrone per aspettare il completamento di una Promise, rendendo il codice più leggibile e simile a un flusso di operazioni sincrone.

Esempio di Funzioni Asincrone

L'esempio di funzioni asincrone dimostra l'uso delle parole chiave `async` e `await`.

In questo esempio la funzione `fetchData` è dichiarata come asincrona e restituisce una Promise che risolve con "Dati ricevuti" dopo un secondo.

La funzione `main` utilizza `await` per aspettare il completamento della Promise restituita da `fetchData`, e stampa i dati ricevuti nella console e il blocco `try/catch` gestisce eventuali errori.

Questo esempio mostra come le funzioni asincrone possono rendere il codice asincrono più leggibile, simile a operazioni sincrone, migliorando la gestione degli errori e la manutenzione del codice.

```
async function fetchData(): Promise<string> {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve("Dati ricevuti");
        }, 1000);
    });
}

async function main() {
    try {
        const data = await fetchData();
        console.log(data); // "Dati ricevuti"
    } catch (error) {
        console.error(error);
    }
}

main();
```

Vantaggi delle Funzioni Asincrone

Utilizzare le funzioni asincrone offre diversi vantaggi: rende il codice più leggibile e facile da capire ed evita il fenomeno del 'callback hell'.

Ultimo, ma non ultimo, permette una gestione più efficiente degli errori tramite i blocchi try/catch.

Infine, le funzioni asincrone permettono di scrivere codice non bloccante che può migliorare le performance delle applicazioni.

Callback Hell

Il concetto di 'Callback Hell' si riferisce a una situazione in cui le funzioni di callback nidificate diventano difficili da leggere e mantenere.

Come detto le callback sono funzioni passate come argomenti ad altre funzioni, quindi vengono eseguite dopo che un'operazione asincrona è completata.

Quando si lavora con operazioni asincrone in JavaScript, come richieste HTTP o letture di file, si finisce spesso per annidare molte callback, portando a codice complicato e poco chiaro.

Esempio di Callback Hell

Consideriamo un esempio in cui vogliamo eseguire una serie di richieste HTTP asincrone in sequenza.

Il seguente codice mostra come le callback nidificate possono rapidamente diventare ingombranti:

```
firstRequest(function(response1) {
    secondRequest(response1, function(response2) {
        thirdRequest(response2, function(response3) {
            fourthRequest(response3, function(response4) {
                console.log('Tutte le richieste sono complete!');
            });
        });
    });
});
```

In questo esempio, ci sono quattro richieste asincrone eseguite in sequenza.

Dato che ogni richiesta dipende dalla risposta della precedente, le callback sono nidificate una dentro l'altra e questo inevitabilmente porta a un codice che è difficile da leggere e mantenere, poiché la struttura indentata diventa rapidamente complessa.

Questo fenomeno è noto come 'Callback Hell' o 'Pyramid of Doom' a causa della forma piramidale del codice.

Problemi con Callback Hell

Sono diversi I problemi associati al callback hell:

Leggibilità: Il codice diventa difficile da leggere e comprendere a causa dell'eccessiva indentazione e della complessità delle funzioni nidificate.

Manutenibilità: Aggiungere o modificare funzionalità diventa complicato, poiché bisogna navigare attraverso molteplici livelli di callback.

Gestione degli errori: La gestione degli errori in un contesto di callback nidificate può diventare confusa e difficile da implementare correttamente.

Soluzioni al Callback Hell

Fortunatamente, ci sono diverse soluzioni per evitare il callback hell:

Promises: Le Promises offrono un modo più pulito per gestire le operazioni asincrone. Consentono di concatenare le operazioni asincrone senza nidificare le funzioni di callback.

Async/Await: Async/Await è una sintassi introdotta in ECMAScript 2017 che permette di scrivere codice asincrono in modo sincrono. Rende il codice più leggibile e facile da mantenere.

Esempio con Promises

Il seguente esempio mostra come lo stesso scenario può essere implementato utilizzando Promises per evitare il callback hell:

```
firstRequest()
    .then(response1 => secondRequest(response1))
    .then(response2 => thirdRequest(response2))
    .then(response3 => fourthRequest(response3))
    .then(response4 => {
        console.log('Tutte le richieste sono complete!');
    })
    .catch(error => {
        console.error('Si è verificato un errore!', error);
    });
```

Esempio con Async/Await

Il seguente esempio mostra come utilizzare Async/Await per rendere il codice asincrono ancora più leggibile e manutenibile e, come diventa evidente, acon tutto ciò che serve anche a rendere più efficiente la gestione degli errori e le reazioni agli errori stessi:

```
async function makeRequests() {
    try {
        const response1 = await firstRequest();
        const response2 = await secondRequest(response1);
        const response3 = await thirdRequest(response2);
        const response4 = await fourthRequest(response3);
        console.log('Tutte le richieste sono complete!');
    } catch (error) {
        console.error('Si è verificato un errore!', error);
    }
}

makeRequests();
```

Conclusioni

Comprendere e utilizzare le callback, le Promises e le funzioni asincrone è fondamentale per scrivere codice JavaScript e TypeScript efficiente e manutenibile avendo a disposizione diversi modi per evitare il **callback hell**.

Fortunatamente l'introduzione di Promises e Async/Await permette di scrivere codice asincrono in modo più pulito e mantenibile, rendendo anche più semplice la gestione degli errori e il codice stesso più robusto.

(pagina lasciata intenzionalmente bianca)

Gestione degli Errori

La gestione degli errori in TypeScript, offre un modo potente e flessibile per trattare situazioni anomale nel codice.

E' possibile creare errori personalizzati consente di fornire messaggi chiari e contestuali, mentre la gestione delle eccezioni con i blocchi try/catch permette di gestire in modo sicuro e strutturato gli errori che si verificano durante l'esecuzione del programma.

Tipi di errore personalizzati

In TypeScript, è possibile creare tipi di errore personalizzati per gestire situazioni specifiche in modo più efficace. I tipi di errore personalizzati consentono di fornire messaggi di errore più chiari e di catturare condizioni di errore specifiche in modo strutturato.

Esempio di tipi di errore personalizzati

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

function doSomethingRisky() {
    throw new CustomError("Qualcosa è andato storto");
}

try {
    doSomethingRisky();
} catch (error) {
    console.error(error.name); // "CustomError"
    console.error(error.message); // "Qualcosa è andato storto"
}
```

Gestione delle eccezioni con TypeScript

La gestione delle eccezioni in TypeScript segue ovviamente lo stesso principio di JavaScript, utilizzando i blocchi try, catch e finally.

TypeScript aggiunge sicurezza tipica alla gestione delle eccezioni, permettendo di catturare e gestire gli errori in modo più strutturato.

Esempio di gestione delle eccezioni

```
function parseJSON(jsonString: string): any {
   try {
       return JSON.parse(jsonString);
   } catch (error) {
       console.error("Errore nel parsing del JSON:", error.message);
       return null;
   }
}

const jsonData = '{"name": "Alice", "age": 25}';
const parsedData = parseJSON(jsonData);
console.log(parsedData); // { name: "Alice", age: 25 }

const invalidJsonData = '{"name": "Alice", "age": }';
const invalidParsedData = parseJSON(invalidJsonData); // Errore nel
parsing del JSON
```

Testare il Codice TypeScript:

Introduzione al Testing con TypeScript

Il testing del codice è una pratica fondamentale nello sviluppo software, soprattutto quando si tratta di progetti di grandi dimensioni. La creazione di test per il codice TypeScript non solo assicura che il tuo software funzioni come previsto, ma previene anche regressioni future, migliora la qualità del codice e facilita la manutenzione.

Quando il codice cresce in complessità e dimensione, diventa sempre più difficile gestirlo senza un'adeguata copertura di test. Un singolo errore in un progetto grande può causare problemi significativi, interrompere il flusso di lavoro e persino portare alla perdita di dati. Per questo motivo, è essenziale integrare un robusto sistema di testing nel tuo processo di sviluppo.

Utilizzo di Framework di Test come *Jest* o *Mocha*

Per testare il codice TypeScript, esistono diversi framework di testing che offrono potenti funzionalità e integrazioni. Tra i più popolari troviamo Jest e Mocha.

Jest: Jest è un framework di testing JavaScript sviluppato da Facebook. È molto apprezzato per la sua semplicità, la sua capacità di mock e la sua integrazione con TypeScript. Jest fornisce una configurazione zero per TypeScript, rendendo facile iniziare rapidamente.

Mocha: Mocha è un framework di testing JavaScript flessibile e ricco di funzionalità. È spesso utilizzato insieme ad altre librerie come Chai per le asserzioni e Sinon per il mocking. Mocha offre una grande flessibilità nella configurazione e nell'esecuzione dei test.

Vediamo ora come utilizzare questi framework per testare il codice TypeScript.

Configurazione di Jest per TypeScript

Per configurare Jest per TypeScript, segui questi passaggi:

Installazione delle dipendenze:

```
npm install --save-dev jest @types/jest ts-jest typescript
```

Configurazione di Jest:

Crea un file `jest.config.js` nella root del progetto e aggiungi la seguente configurazione:

```
module.exports = {
   preset: 'ts-jest',
   testEnvironment: 'node',
   testMatch: ['**/__tests__/**/*.ts', '**/?(*.)+(spec|test).ts']
};
```

Aggiunta dello script di test:

Aggiungi lo script di test nel file `package.json`:

```
"scripts": {
    "test": "jest"
}
```

Configurazione di Mocha per TypeScript

Per configurare Mocha per TypeScript, segui questi passaggi:

Installazione delle dipendenze:

```
npm install --save-dev mocha @types/mocha ts-node typescript chai
@types/chai
```

Configurazione di Mocha:

Crea un file `mocha.opts` nella directory 'test' e aggiungi la seguente configurazione:

```
--require ts-node/register
--recursive
```

Aggiunta dello script di test:

Aggiungi lo script di test nel file `package.json`:

```
"scripts": {
  "test": "mocha 'test/**/*.ts'"
}
```

Scrivere Test Unitari per il Codice TypeScript

Scrivere test unitari è essenziale per garantire che ogni unità del tuo codice funzioni come previsto. I test unitari verificano il comportamento di singole funzioni, metodi o classi isolandole dal resto dell'applicazione.

Vediamo alcuni esempi completi di test unitari utilizzando Jest e Mocha.

Esempio di Test Unitario con Jest

Consideriamo una semplice classe `Calculator` che esegue operazioni aritmetiche di base.

`calculator.ts`:

```
export class Calculator {
  add(a: number, b: number): number {
    return a + b;
  }

  subtract(a: number, b: number): number {
    return a - b;
  }
```

```
  multiply(a: number, b: number): number {
    return a * b;
  }

  divide(a: number, b: number): number {
    if (b === 0) {
      throw new Error('Division by zero');
    }
    return a / b;
  }
}
```

Ora, scriviamo i test unitari per questa classe utilizzando **Jest**.

`calculator.test.ts`:

```
const { Calculator } = require('../calculator');

describe('Calculator', () => {
  let calculator: Calculator;

  beforeEach(() => {
    calculator = new Calculator();
  });

  test('should add two numbers correctly', () => {
    expect(calculator.add(1, 2)).toBe(3);
  });

  test('should subtract two numbers correctly', () => {
    expect(calculator.subtract(5, 3)).toBe(2);
  });

  test('should multiply two numbers correctly', () => {
    expect(calculator.multiply(2, 3)).toBe(6);
  });

  test('should divide two numbers correctly', () => {
    expect(calculator.divide(6, 3)).toBe(2);
  });
```

```
  test('should throw an error when dividing by zero', () => {
    expect(() => calculator.divide(6, 0)).toThrow('Division by zero');
  });
});
```

Esempio di Test Unitario con Mocha

Scriviamo adesso gli stessi test ma utilizzando Mocha e Chai.

`calculator.test.ts`:

```
const { expect } = require('chai');
const { Calculator } = require('../calculator');

describe('Calculator', () => {
  let calculator: Calculator;

  beforeEach(() => {
    calculator = new Calculator();
  });

  it('should add two numbers correctly', () => {
    expect(calculator.add(1, 2)).to.equal(3);
  });

  it('should subtract two numbers correctly', () => {
    expect(calculator.subtract(5, 3)).to.equal(2);
  });

  it('should multiply two numbers correctly', () => {
    expect(calculator.multiply(2, 3)).to.equal(6);
  });

  it('should divide two numbers correctly', () => {
    expect(calculator.divide(6, 3)).to.equal(2);
  });

  it('should throw an error when dividing by zero', () => {
    expect(() => calculator.divide(6, 0)).to.throw('Division by zero');
  });
});
```

L'importanza dei Test nelle Applicazioni di Grandi Dimensioni

In un progetto di grandi dimensioni, la presenza di test è cruciale. Immagina di lavorare su un'applicazione con migliaia di linee di codice e decine di sviluppatori. Senza una suite di test, diventa quasi impossibile garantire che le nuove funzionalità non introducano bug o non rompano funzionalità esistenti.

Benefici dei test unitari:

- **Prevenzione delle regressioni**: I test garantiscono che le nuove modifiche non compromettano le funzionalità esistenti.
- **Facilitazione della manutenzione**: I test aiutano a mantenere il codice. Ogni modifica può essere verificata per assicurarsi che tutto funzioni correttamente.
- **Documentazione del codice**: I test servono come documentazione vivente, mostrando come le diverse parti del sistema dovrebbero comportarsi.
- **Migliore progettazione**: Scrivere test incoraggia una progettazione del codice più modulare e manutenibile.

Rischi senza test unitari:

- **Errori difficili da tracciare**: Senza test, gli errori possono passare inosservati e diventare difficili da tracciare.
- **Costi di manutenzione elevati**: La mancanza di test aumenta il costo di manutenzione del codice, poiché ogni modifica può introdurre nuovi bug.
- **Ridotta affidabilità del software**: Senza una copertura di test adeguata, la qualità e l'affidabilità del software diminuiscono.

Esempi Completi di Test Unitari

Oltre all'esempio della classe `Calculator`, consideriamo un esempio più complesso che coinvolge una classe `User` e una classe `UserService` che gestisce gli utenti.

`user.ts`:

```
export class User {
```

```
    constructor(public id: number, public name: string) {}
}

export class UserService {
  private users: User[] = [];

  addUser(user: User): void {
      this.users.push(user);
  }

  getUser(id: number): User | undefined {
    return this.users.find(user => user.id === id);
  }

  removeUser(id: number): void {
    this.users = this.users.filter(user => user.id !== id);
  }
}
```

User.*test*.ts (con Jest)

```
const { User, UserService } = require('../user');

describe('UserService', () => {
  let userService: UserService;
  let user: User;

  beforeEach(() => {
      userService = new UserService();
      user = new User(1, 'John Doe');
  });

  test('should add a user', () => {
      userService.addUser(user);
      expect(userService.getUser(1)).toEqual(user);
  });

  test('should get a user by ID', () => {
      userService.addUser(user);
      expect(userService.getUser(1)).toEqual(user);
  });
```

```
  test('should return undefined for non-existent user', () =>
  {
      expect(userService.getUser(999)).toBeUndefined();
  });

  test('should remove a user by ID', () => {
      userService.addUser(user);
      userService.removeUser(1);
      expect(userService.getUser(1)).toBeUndefined();
  });
});
```

User.*test*.ts (con Mocha)

```
const { expect } = require('chai');
const { User, UserService } = require('../user');

describe('UserService', () => {
  let userService: UserService;
  let user: User;

  beforeEach(() => {
    userService = new UserService();
    user = new User(1, 'John Doe');
  });

  it('should add a user', () => {
    userService.addUser(user);
    expect(userService.getUser(1)).to.equal(user);
  });

  it('should get a user by ID', () => {
    userService.addUser(user);
    expect(userService.getUser(1)).to.equal(user);
  });

  it('should return undefined for non-existent user', () => {
    expect(userService.getUser(999)).to.be.undefined;
  });
```

```
  it('should remove a user by ID', () => {
    userService.addUser(user);
    userService.removeUser(1);
    expect(userService.getUser(1)).to.be.undefined;
  });
});
```

Conclusioni

Il testing del codice TypeScript è una pratica essenziale per garantire la qualità e la manutenibilità del software e avvalendosi di framework come Jest e Mocha, è possibile scrivere test unitari che verifichino il comportamento delle singole unità di codice, prevenendo errori e regressioni.

Per progetti di grandi dimensioni, una suite di test robusta è fondamentale e aiuta a prevenire bug, facilitano la manutenzione. Inoltre servono quale documentazione vivente del codice.

L'assenza di test può portare a costi elevati di manutenzione, errori difficili da tracciare e una ridotta affidabilità del software, integrare i test nel processo di sviluppo migliora significativamente la qualità del codice e la fiducia nel software prodotto.

Implementare test unitari fin dall'inizio di un progetto non solo protegge il codice da regressioni future, ma rende anche il processo di sviluppo più sicuro e controllato.

In sintesi, la pratica del testing non è solo una buona abitudine, ma una necessità per qualsiasi progetto software di successo. Investire tempo nel testare il codice TypeScript porterà enormi benefici in termini di qualità, affidabilità e manutenibilità del software.

(pagina lasciata intenzionalmente bianca)

Configurazioni Avanzate

La configurazione avanzata di un progetto TypeScript è essenziale per gestire codebase di grandi dimensioni e mantenere il codice organizzato e manutenibile. Con l'aumento della complessità dei progetti, è necessario configurare attentamente il file `tsconfig.json` per sfruttare appieno le funzionalità di TypeScript. Inoltre, l'utilizzo di un monorepo per gestire più progetti può semplificare la gestione delle dipendenze e migliorare la coerenza del codice. Questo capitolo esplorerà le configurazioni avanzate del `tsconfig.json` e come gestire progetti multipli in un monorepo.

Configurazioni avanzate del tsconfig.json

Il file `tsconfig.json` è il cuore della configurazione di un progetto TypeScript. Esso definisce le opzioni del compilatore e specifica quali file devono essere inclusi o esclusi dalla compilazione. Una configurazione avanzata del `tsconfig.json` può migliorare significativamente l'efficienza e la manutenibilità del progetto.

Opzioni del compilatore

Il file `tsconfig.json` offre numerose opzioni per personalizzare la compilazione:

```
{
  "compilerOptions": {
       "target": "ES6",
       "module": "commonjs",
       "strict": true,
       "esModuleInterop": true,
       "skipLibCheck": true,
       "forceConsistentCasingInFileNames": true,
       "outDir": "./dist",
       "rootDir": "./src",
       "baseUrl": "./",
       "paths": {
         "@app/*": ["src/app/*"],
         "@components/*": ["src/components/*"]
       },
       "typeRoots": ["./node_modules/@types", "./src/types"],
       "lib": ["dom", "es6"]
  },
```

```
  "include": ["src/**/*.ts"],
  "exclude": ["node_modules", "dist"]
}
```

Legenda delle opzioni:

- **target**: Specifica la versione di JavaScript verso cui TypeScript deve compilare. In questo esempio, ES6 (o ECMAScript 2015) è il target, garantendo che il codice generato sia compatibile con i moderni ambienti JavaScript.
- **module**: Definisce il sistema di modulo da utilizzare. commonjs è comunemente usato per i progetti Node.js, ma si può usare anche esnext o altri sistemi di moduli.
- **strict**: Abilita tutte le opzioni di controllo rigoroso, come noImplicitAny e strictNullChecks, migliorando la sicurezza del codice e riducendo i potenziali bug.
- **esModuleInterop**: Abilita l'interoperabilità con i moduli ECMAScript, facilitando l'importazione di moduli CommonJS.
- **skipLibCheck**: Disabilita il controllo dei tipi nei file dichiarativi delle librerie, riducendo i tempi di compilazione.
- **forceConsistentCasingInFileNames**: Enforce case-sensitive file naming, ensuring consistent casing.
- **outDir**: Directory di output per i file compilati. Questo aiuta a separare i file di output dai file di origine.
- **rootDir**: Directory radice per i file di origine, utile per mantenere una struttura di progetto organizzata.
- **baseUrl**: Imposta la directory di base per risolvere i moduli, facilitando l'uso degli alias nei percorsi dei moduli.
- **paths**: Configura gli alias dei percorsi dei moduli, rendendo i riferimenti ai moduli più leggibili e manutenibili.
- **typeRoots**: Specifica le directory in cui cercare i file dei tipi dichiarativi, inclusi i tipi personalizzati.
- **lib**: Specifica le librerie TypeScript da includere, in questo caso il DOM e ES6.

Inclusione ed esclusione dei file

include: Specifica i file e le directory da includere nella compilazione. In questo esempio, tutti i file TypeScript nella directory src sono inclusi.

exclude: Specifica i file e le directory da escludere dalla compilazione, come node_modules e dist.

Vantaggi delle configurazioni avanzate

- **Maggiore Manutenibilità**: Una configurazione ben strutturata rende il progetto più facile da mantenere e aggiornare.
- **Ottimizzazione delle Prestazioni**: Escludere i file non necessari e saltare il controllo dei tipi sui file delle librerie può ridurre i tempi di compilazione.
- **Migliore Organizzazione del Codice**: Specificare outDir e rootDir aiuta a mantenere una chiara separazione tra i file sorgente e i file compilati.
- **Facilità di Integrazione**: Usare esModuleInterop e paths facilita l'integrazione con librerie esterne e l'uso degli alias nei percorsi dei moduli.

Monorepo e configurazione di progetti multipli

Un ***monorepo*** è una strategia di gestione del codice in cui il codice di più progetti è archiviato in un unico repository ed è utile a semplificare la gestione delle dipendenze e la condivisione del codice tra i progetti, rendendo il processo di sviluppo più efficiente.

Esempio di struttura di un ***monorepo***

```
monorepo/
│
├── packages/
│       ├── projectA/
│       │       ├── src/
│       │       ├── tsconfig.json
│       │       └── package.json
│       │
│       ├── projectB/
│       │       ├── src/
│       │       ├── tsconfig.json
│       │       └── package.json
│       └── tsconfig.base.json
│
├── node_modules/
├── package.json
└── tsconfig.json
```

tsconfig.base.json

```
{
  "compilerOptions": {
```

```
    "target": "ES6",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  }
}
```

`tsconfig.json` (root)

```
{
  "extends": "./packages/tsconfig.base.json",
  "files": [],
  "include": [],
  "references": [
    { "path": "./packages/projectA" },
    { "path": "./packages/projectB" }
  ]
}
```

`tsconfig.json` (Project A)

```
{
  "extends": "../tsconfig.base.json",
  "compilerOptions": {
    "outDir": "./dist",
    "rootDir": "./src"
  },
  "include": ["src/**/*.ts"],
  "exclude": ["node_modules", "dist"]
}
```

`tsconfig.json` **(**Project B)

```
{
```

```
  "extends": "../tsconfig.base.json",
  "compilerOptions": {
    "outDir": "./dist",
    "rootDir": "./src"
  },
  "include": ["src/**/*.ts"],
  "exclude": ["node_modules", "dist"]
}
```

Vantaggi del Monorepo

Gestione delle Dipendenze Centralizzata: Le dipendenze condivise sono gestite centralmente, riducendo la duplicazione e i conflitti.

Condivisione del Codice: Facilita la condivisione del codice tra i progetti, migliorando la riusabilità e la coerenza.

Semplificazione della Build: Una singola configurazione di build può essere applicata a tutti i progetti, semplificando il processo di build e deployment.

Semplificazione della Collaborazione: I team possono lavorare su progetti diversi all'interno dello stesso repository, facilitando la collaborazione e la revisione del codice.