

O'REILLY®

# Aprendendo TypeScript

Melhore suas habilidades de desenvolvimento  
web usando JavaScript Type-Safe



novatec

Josh Goldberg

# Elogios a *Aprendendo TypeScript*

Se você já se irritou com linhas sublinhadas em vermelho em seu código, leia *Aprendendo TypeScript*. Goldberg contextualiza tudo habilmente, mantendo ao mesmo tempo a praticidade e nos mostrando que o TypeScript não é nunca uma restrição e sim um recurso valioso.

—*Stefan Baumgartner, arquiteto de produtos sênior, Dynatrace; fundador, oida.dev*

Josh dá ênfase aos conceitos mais importantes do TypeScript e fornece explicações sobre eles com exemplos claros e um toque de humor. Leitura obrigatória para desenvolvedores JavaScript que quiserem escrever em TypeScript como um profissional.

—*Andrew Branch, engenheiro de software TypeScript, Microsoft*

*Aprendendo TypeScript* é um ótimo recurso para programadores que já codificaram, mesmo que pouco, mas evitam linguagens tipadas. Ele apresenta um aprofundamento maior do que o manual de TypeScript para que você fique mais confiante ao usar a linguagem em seus projetos.

—*Boris Cherny, engenheiro de software, Meta; autor, Programming TypeScript*

Não sabemos o que é código tipado, mas temos orgulho de Josh e estamos certos de que este é um livro maravilhoso.

—*Frances e Mark Goldberg*

Josh é aquele tipo raro de pessoa que tem paixão não só por dominar os aspectos básicos, mas também por explicar os conceitos para os iniciantes. Acho que este livro se tornará rapidamente um guia não só para iniciantes como também para especialistas em TypeScript.

—*Beyang Liu, Diretor de Tecnologia e cofundador, Sourcegraph*

*Aprendendo TypeScript* é uma ótima introdução e referência da linguagem

TS. A redação de Josh é clara e informativa e isso ajuda a explicar conceitos e sintaxes do TS que geralmente são confusos. É um ponto de partida ideal para qualquer iniciante em TypeScript!

—Mark Erikson, engenheiro front-end sênior, Replay; mantenedor, Redux

*Aprendendo TypeScript* é um excelente livro para você começar sua jornada em TypeScript. Ele lhe dará as ferramentas para entender a linguagem, o sistema de tipagem e a integração com o IDE e para saber como usar tudo isso e aproveitar ao máximo sua experiência com TypeScript.

—Titian Cernicova Dragomir, engenheiro de software, Bloomberg LP

Há muitos anos Josh é um integrante crucial da comunidade TypeScript e fico feliz pelo fato de as pessoas poderem se beneficiar de seu profundo conhecimento e estilo de ensino acessível em *Aprendendo TypeScript*.

—James Henry, arquiteto consultor, Nrwl; 4x vencedor do prêmio Microsoft MVP; criador, angular-eslint e typescript-eslint

Josh não é apenas um engenheiro de software talentoso: ele é um ótimo mentor; é possível sentir sua paixão pela educação neste livro. *Aprendendo TypeScript* foi habilmente estruturado e contém exemplos práticos do mundo real que levarão os iniciantes e entusiastas do TypeScript ao próximo nível. Posso dizer com certeza que *Aprendendo TypeScript* é o guia definitivo para quem deseja aprender ou aumentar seu conhecimento de TypeScript.

—Remo Jansen, Diretor Geral, Wolk Software

Em *Aprendendo TypeScript*, Josh Goldberg detalha os conceitos mais complexos do TypeScript usando descrições descomplicadas e diretas e exemplos fáceis de entender que certamente servirão como auxílio para a aprendizagem e referência durante vários anos. Do primeiro haiku à piada final, *Aprendendo TypeScript* é uma ótima introdução à linguagem, do tipo que me agrada. Desculpem o trocadilho.

—Nick Nisi, engenheiro consultor, C2FO

Costumava-se dizer “Confie sempre em JavaScript”. Agora diz-se “Confie sempre em TypeScript”, e este livro será o recurso mais recomendado do setor. Garanto.

—*Joe Previte, engenheiro open source de TypeScript*

Ler *Aprendendo TypeScript* é como passar algum tempo com um amigo acolhedor e inteligente que adora abordar assuntos fascinantes. No fim você terá se aprofundado em TypeScript e ficará informado sobre a linguagem mesmo se sabia muito ou pouco ao começar.

—*John Reilly, engenheiro consultor, Investec; mantenedor, ts-loader; historiador na Definitely Typed*

*Aprendendo TypeScript* é um guia abrangente, porém acessível, sobre a linguagem e o ecossistema TypeScript. Ele aborda o amplo conjunto de recursos do TypeScript fornecendo ao mesmo tempo sugestões e explicando vantagens e desvantagens com base em uma longa experiência.

—*Daniel Rosenwasser, gerente de programas, TypeScript, Microsoft; representante do TC39*

Este é meu recurso favorito para aprender TypeScript. Da introdução aos tópicos avançados, ele é claro, conciso e abrangente. Acho Josh um excelente – e divertido – autor.

—*Loren Sands-Ramshaw, autor, The GraphQL Guide; engenheiro de SDK do TypeScript, Temporal*

Se você está tentando ser um desenvolvedor TypeScript eficiente, *Aprendendo TypeScript* permitirá que percorra dos conceitos básicos aos avançados.

—*Basarat Ali Syed, engenheiro consultor, SEEK; autor, Beginning NodeJS e TypeScript Deep Dive; Youtuber (Basarat Codes); vencedor do prêmio Microsoft MVP*

Este livro é uma ótima maneira de aprender a linguagem e um complemento perfeito ao Manual de TypeScript (TypeScript Handbook).

—*Orta Therox, ex-engenheiro na equipe do compilador TypeScript, Puzmo*

Josh é um dos mais didáticos e dedicados divulgadores de TypeScript que existem e seu conhecimento finalmente chegou ao formato livro!

Desenvolvedores iniciantes e experientes apreciarão a cuidadosa organização e a sequência dos tópicos. As dicas, observações e avisos no estilo clássico da O'Reilly são muito úteis.

—Shawn “swyx” Wang, chefe de DX, Airbyte

Este livro o ajudará realmente a aprender TypeScript. Os capítulos de teoria junto com os projetos práticos fornecem um bom equilíbrio para a aprendizagem e abordam todos os aspectos da linguagem. A leitura deste livro conseguiu ensinar novos truques até a um profissional experiente como eu. Finalmente aprendi as sutilezas dos Declaration files (*Arquivos de Declaração*). Altamente recomendável.

—Lenz Weber-Tronik, desenvolvedor full stack, Mayflower Germany; mantenedor, Redux

*Aprendendo TypeScript* é um livro acessível e envolvente que se beneficia dos anos de experiência de Josh no desenvolvimento de uma carreira em TypeScript para ensinar tudo o que você precisa saber na ordem certa. Independentemente de sua experiência em programação, você está em boas mãos com Josh e *Aprendendo TypeScript*.

—Dan Vanderkam, engenheiro consultor de software sênior, Google; autor, *Effective TypeScript*

*Aprendendo TypeScript* é o livro que eu gostaria de ter quando conheci TypeScript. A empolgação de Josh para passar conhecimento a novos usuários salta aos olhos em cada página. O livro foi organizado cuidadosamente em partes de fácil compreensão e aborda tudo o que é necessário para você tornar-se um especialista em TypeScript.

—Brad Zacher, engenheiro de software, Meta; principal mantenedor, *typescript-eslint*

# Aprendendo TypeScript

## Melhore suas habilidades de desenvolvimento web usando JavaScript Type-Safe

**Josh Goldberg**

O'REILLY®  
Novatec

Authorized Portuguese translation of the English edition of *Learning TypeScript*, ISBN 9781098110338 © 2022 Josh Goldberg. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Tradução em português autorizada da edição em inglês da obra *Learning TypeScript*, ISBN 9781098110338 © 2022 Josh Goldberg. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., detentora de todos os direitos para publicação e venda desta obra.

© Novatec Editora Ltda. [2022].

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates GRA20220906

Tradução Aldir Coelho Corrêa da Silva

Revisão da tradução: William Bruno Moraes

Revisão gramatical: Alexandra Resende

ISBN impresso: 978-65-86057-99-7

ISBN ebook: 978-85-7522-830-2

Histórico de impressões:

Setembro/2022 Primeira edição

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

Email: [novatec@novatec.com.br](mailto:novatec@novatec.com.br)

Site: <https://novatec.com.br>

Twitter: [twitter.com/novateceditora](https://twitter.com/novateceditora)

Facebook: [facebook.com/novatec](https://facebook.com/novatec)

LinkedIn: [linkedin.com/in/novatec](https://linkedin.com/in/novatec)

GRA20220906

*Dedico este livro à minha incrível parceira, Mariah,  
que me mostrou a alegria de adotar gatos domésticos,  
algo do que ela acabou se arrependendo.*

*Virou um vício.*

# Sumário

[Prefácio](#)

[Parte I Conceitos](#)

[Capítulo 1 Do JavaScript ao TypeScript](#)

[História do JavaScript](#)

[Armadilhas do JavaScript vanilla](#)

[Liberdade cara](#)

[Documentação vaga](#)

[Ferramentas de desenvolvedor insatisfatórias](#)

[TypeScript!](#)

[Começando a usar o TypeScript Playground](#)

[O TypeScript em ação](#)

[Liberdade com restrições](#)

[Documentação precisa](#)

[Ferramentas de desenvolvedor mais robustas](#)

[Compilação da sintaxe](#)

[Comece localmente](#)

[Execução local](#)

[Recursos do editor](#)

[O que o TypeScript não é](#)

[Um remédio para códigos ruins](#)

[Extensões para JavaScript \(em grande parte\)](#)

[Mais lento do que o JavaScript](#)

[Término da evolução](#)

[Resumo](#)

[Capítulo 2 O sistema de tipos](#)

[O que o tipo fornece?](#)

[Sistemas de tipos](#)

[Tipos de erros](#)

[Capacidade de atribuição](#)

[Entendendo os erros de capacidade de atribuição](#)  
[Anotações de tipo](#)  
[Anotações de tipo desnecessárias](#)  
[Formas dos tipos](#)  
[Módulos](#)  
[Resumo](#)

## [Capítulo 3 Uniões e literais](#)

[Tipos união \(union\)](#)  
[Declaração de tipos união](#)  
[Propriedades da união](#)  
[Estreitamento \(narrowing\)](#)  
[Estreitamento por atribuição](#)  
[Verificações condicionais](#)  
[Verificações com typeof](#)  
[Tipos literais](#)  
[Capacidade de atribuição de literais](#)  
[Verificação estrita de nulos](#)  
[O erro de um bilhão de dólares](#)  
[Estreitamento por verdades](#)  
[Variáveis sem valores iniciais](#)  
[Aliases de tipos](#)  
[Os aliases de tipo não são JavaScript](#)  
[Combinando aliases de tipo](#)

[Resumo](#)

## [Capítulo 4 Objetos](#)

[Tipos objeto](#)  
[Declaração de tipos de objeto](#)  
[Apelidos de tipos](#)  
[Tipagem estrutural](#)  
[Verificação de uso](#)  
[Verificação de propriedades em excesso](#)  
[Tipos de objeto aninhados](#)  
[Propriedades opcionais](#)  
[Uniões de tipos de objeto](#)  
[Uniões de tipos de objeto inferidas](#)

[Unões de tipos objeto explícitas](#)  
[Estreitamento de tipos de objeto](#)  
[Unões discriminadas](#)  
[Tipos interseção](#)  
[Perigos dos tipos interseção](#)  
[Resumo](#)

## **Parte II Recursos**

### **Capítulo 5 Funções**

[Parâmetros de funções](#)  
[Parâmetros obrigatórios](#)  
[Parâmetros opcionais](#)  
[Parâmetros padrão](#)  
[Parâmetros rest](#)  
[Tipos de retorno](#)  
[Tipos de retorno explícitos](#)  
[Tipos função](#)  
[Parênteses dos tipos função](#)  
[Inferências do tipo do parâmetro](#)  
[Apelidos de tipos função](#)  
[Mais tipos de retorno](#)  
[Retorno void](#)  
[Retorno never](#)  
[Sobrecargas de funções](#)  
[Compatibilidade das assinaturas de chamadas](#)  
[Resumo](#)

### **Capítulo 6 Arrays**

[Tipos array](#)  
[Tipos de array e função](#)  
[Arrays de tipo união](#)  
[Arrays any modificável](#)  
[Arrays multidimensionais](#)  
[Membros do array](#)  
[Aviso: membros inconsistentes](#)  
[Spreads e Rests](#)

Spreads

Spreading de parâmetros rest

Tuplas

Capacidade de atribuição da tupla

Inferência de tuplas

Resumo

## **Capítulo 7 Interfaces**

Apelidos de tipo versus interfaces

Tipos de propriedades

Propriedades opcionais

Propriedades somente leitura

Funções e métodos

Assinaturas de chamadas

Assinaturas de índice

Interfaces aninhadas

Extensões de interfaces

Propriedades sobrescritas

Extensão de múltiplas interfaces

Mesclagem de interfaces

Conflitos de nomeação de propriedades

Resumo

## **Capítulo 8 Classes**

Métodos de classes

Propriedades de classes

Propriedades de função

Verificação de inicialização

Propriedades opcionais

Propriedades somente leitura

Classes como tipos

Classes e Interfaces

Implementação de múltiplas interfaces

Extensão de uma classe

Capacidade de atribuição na extensão

Construtores sobrescritos

Métodos sobrescritos

[Propriedades sobrescritas](#)

[Classes abstratas](#)

[Visibilidade dos membros](#)

[Modificadores de campo static](#)

[Resumo](#)

## [Capítulo 9 Modificadores de tipo](#)

[Tipos universais](#)

[any, novamente](#)

[unknown](#)

[Predicados de tipo](#)

[Operadores de tipos](#)

[keyof](#)

[typeof](#)

[Asserções de tipo](#)

[Asserção dos tipos de erro capturados](#)

[Asserções de não nulo](#)

[Advertências para as asserções de tipo](#)

[Asserções const](#)

[Literais para primitivos](#)

[Objetos somente leitura](#)

[Resumo](#)

## [Capítulo 10 Genéricos](#)

[Funções genéricas](#)

[Chamadas com tipos genéricos explícitos](#)

[Múltiplos parâmetros de tipo para funções](#)

[Interfaces genéricas](#)

[Tipos inferidos de interfaces genéricas](#)

[Classes genéricas](#)

[Tipos explícitos de classes genéricas](#)

[Extensão de classes genéricas](#)

[Implementação de interfaces genéricas](#)

[Métodos genéricos](#)

[Estáticos genéricos da classe](#)

[Aliases de tipo genéricos](#)

[Uniões discriminadas genéricas](#)

Modificadores de genéricos

Padrões genéricos

Tipos genéricos restritos

keyof e os parâmetros de tipo restritos

Promises

Criação de promises

Funções async

Uso correto dos genéricos

Regra de ouro dos genéricos

Convenções de nomeação dos genéricos

Resumo

## **Parte III Uso**

### **Capítulo 11 Arquivos de declaração**

Arquivos de declaração

Declaração de valores de runtime

Valores globais

Mesclagem de interface global

Aumentos globais

Declarações internas

Declarações de biblioteca

Declarações DOM

Declarações de módulo

Declarações de módulo curinga

Tipos de pacote

declaration

Tipos de pacote de dependências

Exposição de tipos de pacote

DefinitelyTyped

Disponibilidade dos tipos

Resumo

### **Capítulo 12 Uso de recursos do IDE**

Navegação no código

Busca de definições

Busca de referências

Busca de implementações  
Criação de código  
    Preenchimento de nomes  
    Atualizações de importações automáticas  
    Códigos de ação  
Trabalho efetivo com erros  
    Erros do serviço de linguagem  
Resumo  
**Capítulo 13 Opções de configuração**  
Opções do comando tsc  
    Modo pretty  
    Modo de observação  
Arquivos TSConfig  
    tsc --init  
    CLI versus arquivos de configuração  
Inclusões de arquivo  
    include  
    exclude  
Extensões alternativas  
    Sintaxe JSX  
    resolveJsonModule  
Emissão  
    outDir  
    target  
Emissão de declarações  
Mapas de fonte  
    noEmit  
Verificação de tipos  
    lib  
    skipLibCheck  
    Modo estrito  
Módulos  
    module  
    moduleResolution  
Interoperabilidade com o CommonJS

[isolatedModules](#)

[JavaScript](#)

[allowJs](#)

[checkJs](#)

[Suporte ao JSDoc](#)

[Extensões de configuração](#)

[extends](#)

[Bases de configuração](#)

[Referências de projeto](#)

[composite](#)

[references](#)

[Modo de build](#)

[Resumo](#)

## [Parte IV Material complementar](#)

### [Capítulo 14](#)

[Extensões de sintaxe](#)

[Propriedades de parâmetros de classe](#)

[Decorators experimentais](#)

[Enums](#)

[Valores numéricos automáticos](#)

[Enums com valores string](#)

[Enums const](#)

[Namespaces](#)

[Exportações de namespaces](#)

[Namespaces aninhados](#)

[Namespaces em definições de tipo](#)

[Prefira módulos em vez de namespaces](#)

[Importações e exportações somente de tipo](#)

[Resumo](#)

### [Capítulo 15 Operações com tipos](#)

[Tipos mapeados](#)

[Tipos mapeados a partir de outros tipos](#)

[Alteração de modificadores](#)

[Tipos mapeados genéricos](#)  
[Tipos condicionais](#)  
    [Tipos condicionais genéricos](#)  
    [Distributividade dos tipos](#)  
    [Tipos inferidos](#)  
    [Tipos condicionais mapeados](#)  
[never](#)  
    [never e as interseções e uniões](#)  
    [never e os tipos condicionais](#)  
    [never e os tipos mapeados](#)  
[Tipos template literals](#)  
    [Tipos intrínsecos de manipulação de strings](#)  
    [Chaves de template literals](#)  
    [Remapeamento de chaves de tipos mapeados](#)  
[Operações com tipos e a complexidade](#)  
[Resumo](#)  
[\*\*Glossário\*\*](#)  
[\*\*Sobre o autor\*\*](#)  
[\*\*Colofão\*\*](#)

# Prefácio

Minha migração para o TypeScript não foi direta ou rápida. Comecei minha aprendizagem na escola, principalmente escrevendo código Java, depois C++, e como muitos desenvolvedores iniciantes acostumados com linguagens estaticamente tipadas, desprezava JavaScript como “apenas” uma linguagenzinha de script desleixada que as pessoas usavam descuidadamente em sites.

Meu primeiro grande projeto com a linguagem foi uma versão simplificada do videogame *Super Mario Bros* original apenas em HTML5/CSS/JavaScript e, como é típico em muitos projetos iniciais, ficou uma grande bagunça. No começo do projeto desaprovei instintivamente a estranha flexibilidade e a falta de padrão do JavaScript. Somente quase no final é que comecei a respeitar seus recursos e peculiaridades: sua flexibilidade como linguagem e sua capacidade de combinar pequenas funções e de *simplesmente funcionar* nos navegadores dos usuários após segundos de carregamento da página.

Quando terminei esse primeiro projeto, o JavaScript tinha me conquistado.

Inicialmente, a análise estática (ferramentas que analisam código sem executá-lo), como a executada pelo TypeScript, também me incomodava. Eu pensava: *O JavaScript é tão fácil e fluido, por que devemos nos preocupar com estruturas e tipos rígidos?* Estariamos retornando ao mundo do Java e C++, que abandonei?

Voltando aos meus projetos antigos, precisei de 10 minutos para conseguir ler meu complicado velho código JavaScript e entender como tudo pode ficar mais confuso sem a análise estática. Refatorar o código me mostrou todos os locais nos quais eu teria me beneficiado de alguma estrutura. Daquele momento em diante, me convenci de que devia adotar o máximo possível a análise estática em meus projetos.

Já faz mais de uma década desde que comecei a usar TypeScript e continuo gostando da linguagem. Ela ainda está evoluindo com novos recursos e atualmente é mais útil do que jamais foi para o fornecimento de *segurança* e *estrutura* para o JavaScript.

Espero que ao ler *Aprendendo TypeScript* você aprenda a apreciar o TypeScript da mesma forma que eu: não como um simples meio de encontrar bugs e erros de digitação – e certamente ele não é uma alteração significativa nos padrões de código do JavaScript – mas como um JavaScript *com tipos*: um belo sistema que declara como nosso código JavaScript deve funcionar e que pode nos levar a adotá-lo.

## Quem deve ler este livro

Se você sabe escrever código JavaScript, consegue executar comandos básicos em um terminal e está interessado em aprender TypeScript, este livro é para você.

Talvez você tenha ouvido falar que o TypeScript pode ajudá-lo a escrever grandes volumes de código JavaScript com menos bugs (*o que é verdade!*) ou a documentar bem seu código para as outras pessoas que o lerem (*também é verdade!*). Você pode ter visto o TypeScript em postagens de vagas de emprego ou em uma nova função na qual esteja começando.

Seja qual for o motivo, contanto que você conheça os elementos básicos do JavaScript – variáveis funções, closures/escopo e classes – este livro o levará do desconhecimento de TypeScript ao domínio dos fundamentos e dos recursos mais importantes da linguagem. Ao terminar de ler o livro, você saberá:

- A história e o contexto que explicam por que o TypeScript é útil tendo como base o JavaScript “vanilla” (puro).
- Como a tipagem modela o código.
- Como um verificador de tipos analisa o código.
- Como usar anotações de tipo só no desenvolvimento para fornecer informações para a tipagem.
- Como o TypeScript opera com os IDEs (Integrated Development

Environments, Ambientes de Desenvolvimento Integrado) para fornecer ferramentas de exploração e refatoração de código.

E conseguirá:

- Descrever os benefícios do TypeScript e as características gerais de seu sistema de tipos.
- Adicionar anotações de tipo onde for útil em seu código.
- Representar tipos moderadamente complexos usando as inferências internas e a nova sintaxe do TypeScript.
- Usar o TypeScript como auxiliar do desenvolvimento local na refatoração de código.

## **Por que escrevi este livro**

O TypeScript é uma linguagem muito popular tanto no setor empresarial quanto no ambiente open source:

- As pesquisas State of the Octoverse do GitHub realizadas em 2021 e 2020 o consideraram a quarta linguagem mais importante da plataforma, tendo sua posição subido do sétimo lugar em 2019 e 2018 e do décimo lugar em 2017.
- A Pesquisa de Desenvolvedores de 2021 do StackOverflow o considerou a terceira linguagem mais apreciada do mundo (72,73% dos usuários).
- A Pesquisa Estado do JS (State of JS Survey) de 2020 mostrou que o TypeScript apresenta valores consistentemente altos relacionados à satisfação e ao uso tanto como uma ferramenta de build quanto como uma variante do JavaScript.

Para os desenvolvedores front-end, o TypeScript tem um bom suporte em todas as principais bibliotecas e frameworks de UI, o que inclui o Angular, que recomenda o uso de TypeScript, assim como o Gatsby, Next.js, React, Svelte e Vue. Para os desenvolvedores back-end, o TypeScript gera JavaScript que é executado nativamente no Node.js; o Deno, um runtime semelhante do criador do Node, enfatiza suportar diretamente arquivos TypeScript.

No entanto, apesar de todos esses projetos populares suportarem, fiquei um pouco desapontado pela falta de bom conteúdo introdutório online quando aprendi a linguagem. Muitas das fontes de documentação online não conseguiam explicar de forma satisfatória o que é um “sistema de tipos” ou como usá-lo. Geralmente elas apenas presumiam que o leitor já tivesse um bom conhecimento anterior tanto de JavaScript quanto de linguagens fortemente tipadas ou usavam exemplos de código somente superficiais.

Foi uma pena não haver disponível um livro da O'Reilly com uma capa com um belo animal introduzindo o TypeScript anos atrás. Embora existam outros livros sobre TypeScript anteriores a este de editoras como a O'Reilly, não consegui encontrar um que se concentrasse nas bases da linguagem da forma como gostaria: explicando por que ela funciona da maneira como conhecemos e como seus recursos básicos operam em conjunto. Um livro que começasse com uma explicação sobre os fundamentos da linguagem antes de apresentar os recursos individualmente. Fico satisfeito por ter conseguido fazer uma introdução clara e abrangente dos fundamentos da linguagem TypeScript para leitores que ainda não estejam familiarizados com seus princípios.

## Como percorrer o livro

*Aprendendo TypeScript* atende a duas finalidades:

- Você pode lê-lo uma vez para entender o TypeScript como um todo.
- Posteriormente, pode voltar a consultá-lo como um guia introdutório prático da linguagem TypeScript.

Este livro passa dos conceitos ao uso prático no decorrer de três seções gerais:

- **Parte I, “Conceitos”:** Inclui como surgiu o JavaScript, o que o TypeScript adicionou a ele, e as bases de um *sistema de tipos* da forma como o TypeScript o criou.
- **Parte II, “Recursos”:** Descreve como o sistema de tipos interage com as principais partes do JavaScript com as quais você trabalhará ao

escrever código TypeScript.

- **Parte III, “Uso”:** Após você conhecer os recursos que compõem a linguagem TypeScript, demonstra como usá-los em situações do mundo real para melhorar a experiência de leitura e edição de seu código.

Incluí uma seção Part IV, “Material complementar” no final para abordar recursos menos usados, mas, mesmo assim, ocasionalmente úteis do TypeScript. Você não precisa conhecê-los bem para se considerar um desenvolvedor TypeScript. No entanto, são conceitos convenientes com os quais provavelmente você se defrontará ao usar a linguagem para projetos do mundo real. Após terminar a aprendizagem nas três primeiras seções, é altamente recomendável que estude a seção de material complementar.

Cada capítulo começa com um haiku<sup>1</sup>, que expressa a essência do conteúdo, e termina com um trocadilho. A comunidade de desenvolvimento web como um todo e a comunidade de TypeScript que ela engloba são famosas por serem joviais e receberem bem os iniciantes. Tentei tornar este livro agradável de ler para aprendizes como eu que não apreciam textos longos e enfadonhos.

## Exemplos e projetos

Ao contrário de outros materiais de introdução ao TypeScript, este livro se concentra propositalmente na apresentação dos recursos da linguagem com exemplos autocontidos, mostrando apenas as novas informações em vez de analisar projetos de tamanho médio ou grande. Prefiro esse método de ensino porque ele dá ênfase à linguagem. O TypeScript é útil em tantos frameworks e plataformas – muitos dos quais passam por atualizações de API regularmente – que não quis abordar nada como específico de um framework ou plataforma.

Dito isso, na aprendizagem de uma linguagem de programação é extremamente útil praticar os conceitos imediatamente após eles serem introduzidos. É altamente recomendável que você faça uma pausa após cada capítulo para praticar com seus conteúdos. Cada capítulo termina com uma sugestão de visita à sua seção em <https://learningtypescript.com> e

de estudo dos exemplos e projetos listados aí.

## Convenções usadas neste livro

As convenções tipográficas a seguir foram usadas no livro:

### *Itálico*

Indica novos termos, URLs, endereços de email, nomes de arquivo e extensões de arquivo.

### Largura constante

Usada para listagens de programa, assim como dentro de parágrafos para indicar elementos de programa, como nome de variáveis ou funções, tipos de dados, instruções e palavras-chave.



Este elemento representa uma dica ou sugestão.



Este elemento representa uma nota geral.



Este elemento indica um aviso ou cuidado.

## Uso dos exemplos de código

Há material complementar (exemplos de código, exercícios etc.) disponível para download em <https://learningtypescript.com>.

Se você tiver alguma dúvida ou problema técnico referente ao uso dos exemplos de código, envie um email para [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com).

Este livro foi escrito para ajudá-lo a realizar seu trabalho. Se o exemplo de código estiver sendo oferecido com o livro, você poderá usá-lo em seus programas e na sua documentação. Não é preciso entrar em contato conosco para obter permissão a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que use vários trechos de código deste livro não requer permissão. Vender ou distribuir exemplos dos livros da O'Reilly requer permissão. Responder a uma pergunta citando este livro e referindo-se a exemplos de código não requer permissão. Incorporar uma quantidade significativa de exemplos

de código do livro à documentação do seu produto requer permissão.

Apreciamos, mas geralmente não exigimos, quando nos é atribuída autoria. Normalmente, a atribuição de autoria inclui o título, o autor, a editora e o ISBN. Por exemplo: “Aprendendo Typescript de Josh Goldberg (O'Reilly). Copyright 2022 Josh Goldberg, 978-1-098-11033-8”.

Se você achar que o uso que está fazendo dos exemplos de código não se enquadra no uso ou na permissão legal mencionado anteriormente, fique à vontade para entrar em contato conosco em [permissions@oreilly.com](mailto:permissions@oreilly.com).

## Como entrar em contato conosco

Envie comentários e dúvidas sobre este livro para:  
[novatec@novatec.com.br](mailto:novatec@novatec.com.br).

Temos uma página da web para este livro, na qual incluímos a lista de erratas, exemplos e qualquer outra informação adicional.

- Página da edição em português  
<https://novatec.com.br/livros/aprendendo-typescript>
- Página da edição original, em inglês  
<https://oreillylearning-typescript>

Para obter mais informações sobre livros da Novatec, acesse nosso site em:

<https://novatec.com.br>

## Agradecimentos

Este livro é resultado de um trabalho em equipe e gostaria de agradecer a todos que o tornaram possível. Em primeiro lugar, à incansável editora-chefe, Rita Fernando, pela incrível paciência e pelo excelente aconselhamento durante a jornada de criação. Também gostaria de estender o agradecimento aos outros membros da equipe da O'Reilly: Kristen Brown, Suzanne Huston, Clare Jensen, Carol Keller, Elizabeth Kelly, Cheryl Lenser, Elizabeth Oliver e Amanda Quinn. Vocês são ótimos!

Fico profundamente grato aos revisores técnicos por seus insights pedagógicos de alta qualidade e pelo domínio do TypeScript: Mike Boyle, Ryan Cavanaugh, Sara Gallagher, Michael Hoffman, Adam Reineke e Dan Vanderkam. Este livro não seria o mesmo sem vocês e espero que eu tenha conseguido capturar a intenção de todas as suas excelentes sugestões!

Agradeço também aos colegas e críticos por fornecerem avaliações que me ajudaram a melhorar a precisão técnica e a qualidade da redação: Robert Blake, Andrew Branch, James Henry, Adam Kaczmarek, Loren Sands-Ramshaw, Nik Stern e Lenz Weber-Tronic. Sugestões são sempre úteis!

Por fim, gostaria de agradecer à minha família por seu amor e apoio no decorrer dos anos. Aos meus pais, Frances e Mark, e ao meu irmão – obrigado por me deixarem passar tempo com o Lego e com livros e videogames. Obrigado também à minha esposa Mariah Goldberg por sua paciência durante meus longos períodos de edição e redação e aos nossos gatos, Luci, Tiny e Jerry, por serem tão fofos e me fazerem companhia.

---

<sup>1</sup> N.T.: O haiku (ou haikai) é uma forma curta de poesia japonesa, composta por 17 sílabas, distribuídas em três versos tradicionais de 5, 7 e 5 sílabas, respectivamente. No original, em inglês, esta regra foi seguida, mas na tradução para o português a regra e a essência se perdem. Por este motivo colocamos o texto original de cada haiku escrito pelo autor como nota de rodapé. O autor também finaliza cada capítulo com um trocadilho, e também incluímos o texto original como nota de rodapé para o leitor entender melhor as nuances do trocadilho.

PARTE I

# **Conceitos**

# CAPÍTULO 1

## Do JavaScript ao TypeScript

*O JavaScript hoje  
Suporta navegadores de décadas passadas  
É a beleza da web<sup>1</sup>*

Antes de falar sobre o TypeScript, precisamos saber de onde ele veio: do JavaScript!

### História do JavaScript

O JavaScript foi projetado em 10 dias por Brendan Eich na Netscape em 1995 para ser acessível e fácil de usar em sites. Desde então os desenvolvedores zombam de suas peculiaridades e deficiências. Abordarei algumas delas na próxima seção.

Na verdade, a linguagem evoluiu muito desde 1995! Seu comitê diretor, o TC39, vem lançando anualmente novas versões do ECMAScript – a especificação de linguagem na qual o JavaScript se baseia – desde 2015, com novos recursos que o alinharam a outras linguagens modernas. O interessante é que, mesmo com novas versões regulares da linguagem, o JavaScript conseguiu manter a compatibilidade com versões anteriores por décadas em vários ambientes, que incluem navegadores, aplicações embutidas e runtimes de servidor.

Atualmente, o JavaScript é uma linguagem extremamente flexível que apresenta muitas vantagens. Temos de admitir que, embora ele tenha suas peculiaridades, também ajudou a impulsionar o enorme crescimento das aplicações web e da internet.

*Mostre-me a linguagem de programação perfeita e lhe mostrarei uma*

*linguagem sem usuários.*

—ANDERS HEJLSBERG, TSConf 2019

## Armadilhas do JavaScript vanilla

Os desenvolvedores costumam se referir ao uso de JavaScript sem extensões da linguagem ou frameworks significativos empregando o termo “vanilla”: querendo dizer que se trata da versão original. Explicarei em breve porque o TypeScript adiciona o que faltava para superar essas dificuldades específicas, mas é útil saber porque elas são graves. Todos esses pontos fracos tornam-se mais evidentes conforme um projeto cresce e tem vida útil maior.

### Liberdade cara

A maior queixa de muitos desenvolvedores relacionada ao JavaScript é, infelizmente, um de seus principais recursos: o JavaScript não apresenta praticamente nenhuma restrição a como estruturamos o código. Essa liberdade torna muito divertido começar um projeto em JavaScript!

No entanto, à medida que o número de arquivos cresce, fica evidente como essa liberdade pode ser prejudicial. Veja o trecho de código a seguir, apresentado fora de contexto a partir de alguma aplicação de desenho fictícia:

```
function paintPainting(painter, painting) {
    return painter
        .prepare()
        .paint(painting, painter.ownMaterials)
        .finish();
}
```

Lendo esse código sem nenhum contexto podemos apenas ter uma vaga ideia de como chamar a função `paintPainting`. Talvez se você examinasse o codebase<sup>2</sup> ao redor conseguiria perceber de que provavelmente `painter` é o que é retornado por alguma função `getPainter`. Poderia até mesmo supor corretamente que `painting` é uma string.

No entanto, mesmo se essas suposições estiverem corretas, alterações

feitas no código posteriormente podem invalidá-las. Talvez `painting` seja alterado de string para algum outro tipo de dado ou um ou mais dos métodos de `painter` podem ser renomeados.

Outras linguagens podem não permitir que você execute o código se o seu compilador determinar que provavelmente ele quebraria. Isso não ocorre com linguagens dinamicamente tipadas – as que executam código sem antes verificar se ele pode quebrar – como o JavaScript.

A liberdade na codificação que torna o JavaScript tão divertido pode acabar sendo um grande incômodo se quisermos segurança ao executar o código.

## Documentação vaga

Não existe nada na especificação da linguagem JavaScript que formalize como descrever quais devem ser os parâmetros e os valores de retorno das funções, as variáveis ou outras estruturas do código. Muitos desenvolvedores têm adotado um padrão chamado JSDoc para descrever funções e variáveis usando comentários de bloco. O padrão JSDoc descreve como podemos escrever comentários de documentação inseridos diretamente acima de estruturas, como as funções e variáveis, formatados de maneira padrão. Aqui está um exemplo, novamente usado fora de contexto:

```
/**  
 * Performs a painter painting a particular painting  
 *  
 * @param {Painting} painter  
 * @param {string} painting  
 * @returns {boolean} Whether the painter painted the painting  
 */  
function paintPainting(painter, painting) { /* ... */ }
```

O JSDoc apresenta problemas importantes que geralmente o tornam difícil de usar em um codebase maior:

- Nada impede que as descrições do JSDoc estejam erradas em relação ao código.
- Mesmo se as descrições estivessem corretas, durante refatorações do

código pode ser difícil encontrar todos os comentários agora inválidos do JSDoc relacionados às alterações.

- Descrever objetos complexos é difícil e verboso, demandando vários comentários autônomos para definir os tipos e seus relacionamentos.

Editar comentários do JSDoc em uma dúzia de arquivos não toma muito tempo, mas em centenas, ou até mesmo milhares de arquivos sendo atualizados constantemente pode ser uma tarefa árdua.

## Ferramentas de desenvolvedor insatisfatórias

Já que o JavaScript não fornece maneiras internas de identificar tipos e o código diverge facilmente dos comentários do JSDoc, pode ser difícil automatizar alterações grandes em um codebase ou entendê-lo melhor. Os desenvolvedores JavaScript costumam ficar surpresos ao ver recursos de linguagens tipadas como C# e Java que permitem que eles executem renomeações de membros de classes ou saltem para o local em que o tipo de um argumento foi declarado.



Você poderia protestar dizendo que IDEs modernos como o VS Code fornecem algumas ferramentas de desenvolvimento como as refatorações automatizadas para o JavaScript. É verdade, mas eles usam o TypeScript ou algo equivalente em segundo plano em muitos de seus recursos JavaScript e essas ferramentas de desenvolvimento não são tão confiáveis ou poderosas na maioria dos códigos JavaScript da maneira como o são em códigos TypeScript bem definidos.

## TypeScript!

O TypeScript foi criado internamente na Microsoft no início dos anos 2010 e lançado e tornado open source em 2012. O coordenador de seu desenvolvimento chama-se Anders Hejlsberg, famoso por também ter liderado o desenvolvimento das populares linguagens C# e Turbo Pascal. Geralmente, o TypeScript é descrito como um “superconjunto (superset) do JavaScript” ou como um “JavaScript tipado”. No entanto, o que é

## TypeScript?

O TypeScript pode ser considerado de quatro formas:

### *Como uma linguagem de programação*

Uma linguagem que inclui toda a sintaxe existente do JavaScript, mais uma nova sintaxe específica do TypeScript para a definição e o uso de tipos.

### *Como um verificador de tipos*

Um programa que acessa um conjunto de arquivos escritos em JavaScript e/ou TypeScript, desenvolve um conhecimento de todas as estruturas (variáveis, funções...) criadas e nos avisa se ele considera que algo foi definido incorretamente.

### *Como um compilador*

Um programa que executa o verificador de tipos (type checker), relata qualquer problema e então retorna o código JavaScript equivalente.

### *Como um serviço de linguagem (language service)*

Um programa que usa o verificador de tipos para informar a editores, como o VS Code, como fornecer funcionalidades úteis para os desenvolvedores.

## Começando a usar o TypeScript Playground

Você já leu o bastante sobre o TypeScript. Agora vai escrever!

O principal site de TypeScript inclui um editor “Playground” em <https://www.typescriptlang.org/play>. Você pode digitar código no editor principal e verá muitas das mesmas sugestões de edição que veria trabalhando com o TypeScript localmente em um IDE (Integrated Development Environment, Ambiente de Desenvolvimento Integrado) completo.

Quase todos os trechos de código deste livro são intencionalmente pequenos e suficientemente autocontidos o bastante para você digitá-los no Playground e se divertir manipulando-os.

## O TypeScript em ação

Veja este trecho de código:

```
const firstName = "Georgia";
const nameLength = firstName.length();
//           ~~~~~
// This expression is not callable
```

O código foi escrito na sintaxe normal do JavaScript – ainda não introduzi a sintaxe específica do TypeScript. Se você executasse o verificador de tipos do TypeScript nesse código, ele usaria seu conhecimento de que a propriedade `length` de uma string é um número – e não uma função – para fornecer o alerta exibido no comentário.

Se você colar esse código no Playground ou em outro editor, o serviço de linguagem solicitará que ele exiba uma pequena linha vermelha ondulada abaixo de `length` para indicar que o TypeScript não ficou satisfeito com seu código. Quando você mover o cursor sobre o código sublinhado, verá o texto do alerta (Figura 1.1).

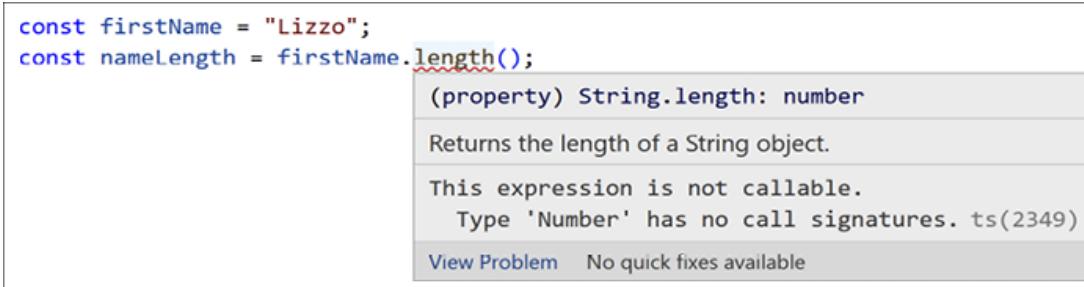


Figura 1.1: O TypeScript relatando um erro sobre a propriedade `length` da string não poder ser executada.

Ser informado sobre esses erros simples em seu editor quando você digitá-los é muito mais cômodo do que esperar até que uma linha de código específica seja executada e ocorra um erro. Se você tentasse executar esse código em JavaScript, ele quebraria!

## Liberdade com restrições

O TypeScript permite especificar quais tipos de valores poderão ser fornecidos a parâmetros e variáveis. Inicialmente alguns desenvolvedores acham restritivo escrever explicitamente no código como determinadas

áreas devem operar.

Eu diria que, na verdade, ficar “restrito” desta forma é algo bom! Ao restringir nosso código a somente ser usado da maneira especificada, o TypeScript assegura que as alterações feitas em uma área do código não prejudiquem outras áreas que a usarem.

Se, digamos, alterássemos o número de parâmetros requeridos por uma função, o TypeScript nos informaria se esquecêssemos de atualizar um local em que ela fosse chamada.

No exemplo a seguir, a função `sayMyName` foi alterada de receber dois parâmetros para receber apenas um, mas a chamada feita a ela com duas strings não foi atualizada e, portanto, gerou um alerta do TypeScript:

```
// Anteriormente: sayMyName(firstName, lastNameName) { ...
function sayMyName(fullName) {
    console.log(`You acting kind of shady, ain't callin' me ${fullName}`);
}

sayMyName("Beyoncé", "Knowles");
// ~~~~~
// Expected 1 argument, but got 2.
```

Esse código seria executado sem travar no JavaScript, mas sua saída seria diferente da esperada (ela não incluiria "Knowles"):

```
You acting kind of shady, ain't callin' me Beyoncé
```

Chamar funções com o número de argumentos errado é exatamente o tipo de liberdade imprudente do JavaScript que o TypeScript restringe.

## Documentação precisa

Vejamos uma versão da função `paintPainting` do exemplo anterior no TypeScript. Embora eu ainda não tenha descrito as particularidades da sintaxe do TypeScript para a documentação de tipos, o trecho de código a seguir demonstra a grande precisão da linguagem para a documentação de códigos:

```
interface Painter {
    finish(): boolean;
    ownMaterials: Material[];
    paint(painting: string, materials: Material[]): boolean;
```

```

}

function paintPainting(painter: Painter, painting: string): boolean { /* ... */ }

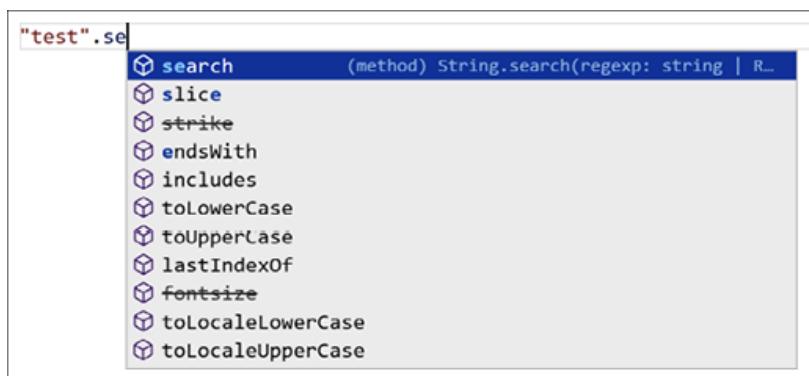
```

Um desenvolvedor TypeScript que lesse esse código pela primeira vez conseguiria entender que `painter` tem pelo menos três propriedades, duas das quais são métodos. Ao incorporar uma sintaxe que descreve as “formas” dos objetos, o TypeScript fornece um excelente e obrigatório sistema para a descrição de sua aparência.

## Ferramentas de desenvolvedor mais robustas

As tipagens do TypeScript permitem que editores como o VS Code recebam informações muito mais detalhadas sobre o código. Eles podem então usar essas informações para gerar sugestões inteligentes enquanto digitamos. Essas sugestões podem ser extremamente úteis para o desenvolvimento.

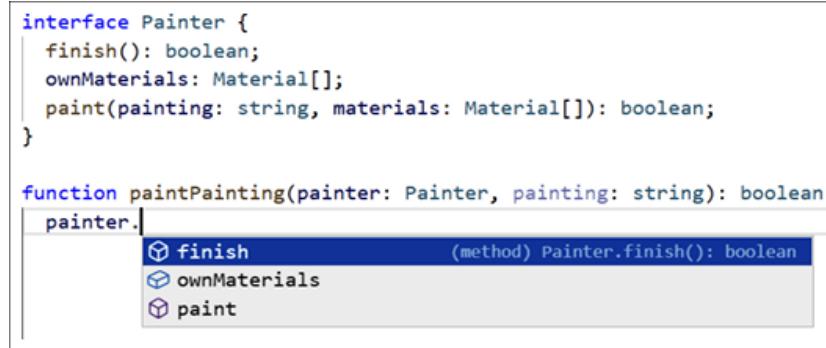
Se você já usou o VS Code para escrever JavaScript, deve ter notado que ele sugere “autopreenchimento” quando digitamos código com tipos de objetos internos como as strings. Se, por exemplo, você começasse a digitar uma propriedade de algo que sabidamente fosse uma string, o TypeScript faria sugestões com todas as propriedades das strings (Figura 1.2).



*Figura 1.2: O TypeScript fornecendo sugestões de autocompletar em JavaScript para uma string.*

Quando você adicionar o verificador de tipos do TypeScript para conseguir entender os códigos, ele fornecerá essas sugestões úteis até para

um código existente que você tiver escrito. Na digitação de `painter`, na função `paintPainting`, o TypeScript usaria seu conhecimento de que o parâmetro `painter` é de tipo `Painter` e de que esse tipo tem os membros a seguir (Figura 1.3).



```
interface Painter {
    finish(): boolean;
    ownMaterials: Material[];
    paint(painting: string, materials: Material[]): boolean;
}

function paintPainting(painter: Painter, painting: string): boolean
    painter.|
```

A screenshot of a code editor showing a tooltip for the `painter` parameter in the `paintPainting` function. The tooltip lists three methods from the `Painter` interface: `finish`, `ownMaterials`, and `paint`. The `finish` method is highlighted with a blue background.

Figura 1.3: O TypeScript fornecendo sugestões de autopreenchimento em JavaScript para uma string.

Estilos! Abordarei vários outros recursos úteis dos editores no Capítulo 12, “Uso de recursos do IDE”.

## Compilação da sintaxe

O compilador do TypeScript nos permite inserir uma sintaxe TypeScript, verificar seus tipos e fazer o JavaScript equivalente ser emitido. Por conveniência, o compilador também pode receber a sintaxe JavaScript moderna e compilá-la para seus equivalentes mais antigos do ECMAScript.

Se você colasse este código TypeScript no Playground:

```
const artist = "Augusta Savage";
console.log({ artist });
```

Ele exibiria no lado direito da tela que este seria o equivalente JavaScript emitido pelo compilador (Figura 1.4).

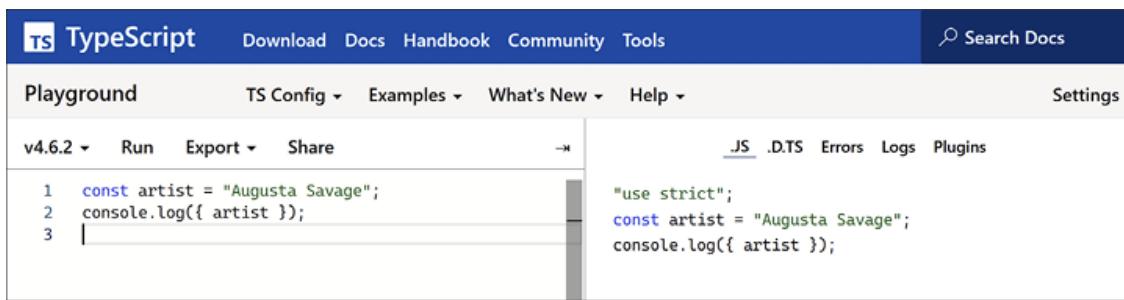


Figura 14: O TypeScript Playground compilando código TypeScript para o JavaScript equivalente.

O TypeScript Playground é uma ótima ferramenta para mostrar como o código-fonte TypeScript se transforma em JavaScript.



Muitos projetos JavaScript usam transpiladores<sup>3</sup> dedicados como o Babel (<https://babeljs.io>) em vez do transpilador próprio do TypeScript para transpilar código-fonte em JavaScript executável. Você pode encontrar uma lista dos starters<sup>4</sup> comuns para projetos em <https://learningtypescript.com/starters>.

## Comece localmente

Você pode executar o TypeScript em seu computador contanto que tenha o Node.js instalado. Para instalar a última versão do TypeScript globalmente, execute o comando a seguir:

```
npm i -g typescript
```

Agora você poderá executar o TypeScript na linha de comando usando o comando `tsc` (TypeScript Compiler, Compilador Typescript). Faça o teste com a flag `--version` para verificar se ele está instalado corretamente:

```
tsc --version
```

O comando deve exibir algo como `Version X.Y.Z` – qualquer que fosse a versão atual quando você instalou o TypeScript:

```
$ tsc --version
Version 4.7.2
```

## Execução local

Agora que o TypeScript está instalado, você definirá uma pasta

localmente para executá-lo nos códigos. Crie uma pasta em algum local em seu computador e execute este comando para gerar um novo arquivo de configuração `tsconfig.json`:

```
tsc --init
```

Um arquivo `tsconfig.json` declara as configurações que o TypeScript usará ao analisar o código. A maioria das opções desse arquivo não será relevante para você neste livro (existem muitos casos extremos incomuns em programação que a linguagem precisa solucionar!). Abordarei as opções no Capítulo 13, “Opções de configuração”. O importante é que agora você pode executar `tsc` para solicitar ao TypeScript que compile todos os arquivos da pasta e ele recorrerá ao arquivo `tsconfig.json` para acessar as opções de configuração.

Tente adicionar um arquivo chamado `index.ts` com o conteúdo a seguir:

```
console.blub("Nothing is worth more than laughter.");
```

Em seguida, execute o `tsc` e forneça para ele o nome desse arquivo, `index.ts`:

```
tsc index.ts
```

Você deve ver uma mensagem de erro semelhante a esta:

```
index.ts:1:9 - error TS2339: Property 'blub' does not exist on type 'Console'.
```

```
1 console.blub("Nothing is worth more than laughter.");
```

```
~~~~~
```

```
Found 1 error.
```

Realmente não existe um método `blub` para o `console`. No que eu estava pensando?

Antes de corrigir o código para satisfazer ao TypeScript, observe que o `tsc` criou um arquivo `index.js` para você com conteúdos que incluem `console.blub`.



Esse é um conceito importante: ainda que tenha ocorrido um *type error (erro de tipo)* em nosso código, a *sintaxe* é totalmente válida. O compilador TypeScript produzirá o JavaScript a partir de um arquivo de entrada independentemente de qualquer erro de tipo.

Corrija o código de `index.ts` para que ele chame `console.log` e execute o `tsc` novamente. Não deve haver alertas em seu terminal e agora o arquivo `index.js` conterá o código da saída atualizado:

```
console.log("Nothing is worth more than laughter.");
```

 É altamente recomendável que você pratique usando os trechos de código deste livro enquanto você lê, no Playground ou em um editor com suporte ao TypeScript, ou seja, que execute o serviço de linguagem do TypeScript. Pequenos exercícios autocontidos, assim como projetos maiores, também estão disponíveis para ajudá-lo a praticar o que aprendeu em <https://learningtypescript.com>.

## Recursos do editor

Outro benefício da criação de um arquivo `tsconfig.json` é que quando os editores forem abertos com uma pasta específica, agora eles a reconhecerão como um projeto do TypeScript. Por exemplo, se você abrir o VS Code com uma pasta, as configurações que ele usará para analisar o código TypeScript respeitarão o que quer que esteja no arquivo `tsconfig.json` dessa pasta.

Como exercício, volte aos trechos de código deste capítulo e digite-os em seu editor. Você deve ver menus suspensos sugerindo autocompletamento para os nomes ao digitá-los, principalmente para propriedades como `log` do `console`.

Muito interessante: você está usando o serviço de linguagem do TypeScript para ajudá-lo a escrever código! Está no caminho certo para tornar-se um desenvolvedor TypeScript!

 O VS Code vem com um ótimo suporte ao TypeScript e ele próprio foi criado em TypeScript. Você não *precisa* usá-lo para trabalhar com o TypeScript – praticamente todos os editores modernos têm um excelente suporte ao TypeScript, seja interno ou disponível por meio de plugins – mas recomendo que o use pelo menos para fazer testes com a linguagem enquanto lê este livro. Se você usar um editor diferente, também recomendo ativar o seu suporte ao TypeScript.

Abordarei os recursos dos editores com mais detalhes no Capítulo 12, “Usando recursos do IDE”.

## 0 que o TypeScript não é

Agora que você viu quão maravilhoso o TypeScript é, preciso alertá-lo sobre algumas limitações. Todas as ferramentas se destacam em algumas áreas e têm limitações em outras.

### Um remédio para códigos ruins

O TypeScript ajuda a estruturar o JavaScript, mas exceto por reforçar a segurança de tipo (type safety), não adota nenhuma posição sobre como deve ser a estrutura.

Isso é bom!

O TypeScript é uma linguagem que foi projetada para que todos consigam usá-la e não para ser um framework opinativo com um público específico. Você pode escrever código usando os padrões de arquitetura que está acostumado a empregar com JavaScript e o TypeScript os suportará.

Se alguém tentar lhe dizer que o TypeScript nos força a usar classes, torna difícil escrever códigos bons ou fizer qualquer outra reclamação quanto ao estilo do código, olhe para essa pessoa com seriedade e diga-lhe para pegar uma cópia de *Aprendendo TypeScript*. O TypeScript não tem nenhuma opinião em relação ao estilo do código, como devemos usar classes ou funções, nem dá preferência a algum framework específico – Angular, React etc. – sobre outros.

### Extensões para JavaScript (em grande parte)

Os objetivos de design do TypeScript declaram explicitamente que ele deve:

- Estar alinhado com as propostas atuais e futuras do ECMAScript.
- Preservar o comportamento de runtime de todos os códigos JavaScript.

O TypeScript não tenta alterar como o JavaScript funciona. Seus criadores queriam evitar adicionar novos recursos de codificação que adicionassem ou entrassem em conflito com o JavaScript. Essa tarefa é responsabilidade do TC39, o comitê técnico que trabalha no ECMAScript.

Existem alguns recursos mais antigos do TypeScript que foram adicionados anos atrás para refletir casos de uso comuns em código JavaScript. Quase todos esses recursos são relativamente incomuns ou deixaram de ser usados e serão abordados apenas brevemente no Capítulo 14, “Extensões da sintaxe”. Recomendo evitá-los na maioria dos casos.



A partir de 2022, o TC39 começou a examinar a inclusão de uma sintaxe de anotações de tipo (type annotations) para o JavaScript. As últimas propostas preconizam que elas atuem como uma espécie de comentário que não afete o código no runtime e só sejam usadas para sistemas específicos do tempo de desenvolvimento como o TypeScript. Só depois de muitos anos os comentários de tipo ou algo equivalente serão adicionados ao JavaScript, logo, eles não serão mencionados em outro local deste livro.

## Mais lento do que o JavaScript

Você pode encontrar na internet alguns desenvolvedores inflexíveis se queixando de que o TypeScript é mais lento do que o JavaScript no runtime. Essa reclamação é imprecisa e enganosa. As únicas alterações que o TypeScript faz no código é quando solicitamos que ele o compile para versões mais antigas do JavaScript a fim de suportar ambientes de runtime (runtime environments) anteriores como o do Internet Explorer 11. Muitos frameworks de produção não usam o compilador TypeScript; em vez dele usam uma ferramenta separada para a transpilação (a parte da compilação que converte código-fonte de uma linguagem de programação para outra) e só usam o TypeScript para a verificação de tipos.

No entanto, o TypeScript realmente aumenta um pouco o tempo de criação do código. O código TypeScript deve ser compilado para

JavaScript antes que a maioria dos ambientes, como os navegadores e o Node.js, o execute. Geralmente grande parte dos pipelines de build é definida de modo que o impacto sobre o desempenho seja mínimo e os recursos mais lentos do TypeScript como a análise do código em busca de possíveis erros são executados separadamente da geração dos arquivos de código executável das aplicações.



Até mesmo projetos que aparentemente permitem a execução do código TypeScript de maneira direta, como o ts-node e o Deno, o convertem internamente para JavaScript antes de executá-lo.

## Término da evolução

A web está longe de chegar ao término de sua evolução e, portanto, o mesmo ocorre com o TypeScript. A linguagem TypeScript está sempre recebendo correções de bugs e acréscimos de recursos para atender às necessidades em constante mudança da comunidade web. Os princípios básicos do TypeScript que você aprenderá neste livro não mudarão muito, mas as mensagens de erro, os recursos sofisticados e as integrações com os editores melhorarão com o tempo.

Na verdade, embora esta edição do livro tenha sido publicada com o TypeScript 4.7.2 como a versão mais recente, podemos ter certeza de que quando você começou a lê-la já havia uma versão mais nova. Algumas das mensagens de erro do TypeScript exibidas no livro podem até mesmo já estar desatualizadas!

## Resumo

Neste capítulo, você recebeu informações sobre o contexto de alguns dos principais pontos fracos do JavaScript, sobre onde o TypeScript é usado e sobre como começar a usá-lo:

- Uma breve história do JavaScript.
- Armadilhas do JavaScript: liberdade cara, documentação vaga e ferramentas de desenvolvedor insatisfatórias.
- O que é TypeScript: uma linguagem de programação, um verificador

de tipos, um compilador e um serviço de linguagem.

- Vantagens do TypeScript: liberdade com restrições, documentação precisa e ferramentas de desenvolvedor mais robustas.
- Você pode começar a escrever código TypeScript no TypeScript Playground e localmente em seu computador.
- O que o TypeScript não é: um remédio para códigos ruins, extensões para JavaScript (em grande parte), mais lento do que o JavaScript ou o término da evolução.



Agora que você terminou de ler este capítulo, pratique o que aprendeu em <https://learningtypescript.com/from-javascript-to-typescript>.

*O que acontecerá se você detectar erros ao executar o compilador  
TypeScript?  
É melhor capturá-los!<sup>5</sup>*

---

<sup>1</sup> N.T.: Original: *JavaScript today / Supports browsers decades past / Beauty of the web*

<sup>2</sup> N.T.: Codebase (ou code base) é o corpo completo do código-fonte de um programa ou aplicação.

<sup>3</sup> N.T.: Transpilador é uma ferramenta que lê o código-fonte escrito em uma linguagem e produz um código equivalente em outra linguagem, com o mesmo nível de abstração.

<sup>4</sup> N.T.: Starters são repositórios na forma de um modelo que permitem criar uma solução pré-configurada.

<sup>5</sup> N.T.: Original: *What happens if you spot errors running the TypeScript compiler? You'd better go catch them!*

# CAPÍTULO 2

## O sistema de tipos

*O poder do JavaScript  
Vem da flexibilidade  
Cuidado com isso!<sup>1</sup>*

Falei brevemente no Capítulo 1, “Do JavaScript ao TypeScript”, sobre a existência de um “verificador de tipos” no TypeScript que examina o código, entende seu funcionamento e nos informa onde podemos ter errado. No entanto, como um verificador de tipos funciona na prática?

### O que o tipo fornece?

Um “tipo” é uma descrição de qual pode ser a *forma* de um valor do código JavaScript. Com o termo “forma” eu me refiro a que propriedades e métodos existem em um valor, e como o operador interno `typeof` a descreveria.

Por exemplo, se criarmos uma variável com o valor inicial "Aretha":

```
let singer = "Aretha";
```

O TypeScript poderá inferir, ou entender, que a variável `singer` é de *tipo string*.

Os tipos mais básicos do TypeScript correspondem aos sete tipos básicos de primitivos do JavaScript:

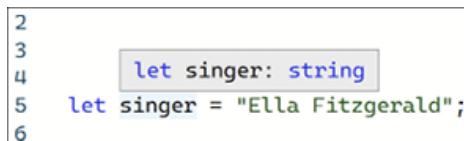
- `null`
- `undefined`
- `boolean // true OU false`
- `string // "", "Hi!", "abc123", ...`
- `number // 0, 2.1, -4, ...`

- `bigint` // `0n`, `2n`, `-4n`, ...
- `symbol` // `Symbol()`, `Symbol("hi")`, ...

Para cada um destes valores, o TypeScript deduzirá que o seu tipo equivale a um dos sete tipos primitivos básicos:

- `null`; // `null`
- `undefined`; // `undefined`
- `true`; // `boolean`
- `"Louise"`; // `string`
- `1337`; // `number`
- `1337n`; // `bigint`
- `Symbol("Franklin")`; // `symbol`

Se você se esquecer do nome de um primitivo, poderá digitar `let` e a variável com o valor do primitivo no *TypeScript Playground* (<https://www.typescriptlang.org/>) ou em um IDE e passar o ponteiro do mouse sobre o nome da variável. O popover<sup>2</sup> resultante incluirá o nome do primitivo, como neste screenshot que exibe a passagem do cursor sobre uma variável `string` (Figura 2.1).



*Figura 2.1: O TypeScript exibindo o tipo `string` de uma variável nas informações da passagem do ponteiro do mouse sobre ela.*

O TypeScript também é esperto o bastante para conseguir inferir o tipo de uma variável cujo valor inicial é calculado. Neste exemplo, ele sabe que a expressão ternária sempre resulta em uma `string`, logo, a variável `bestSong` é um `string`:

```
// Inferred type: string
let bestSong = Math.random() > 0.5
  ? "Chain of Fools"
  : "Respect";
```

De volta ao *TypeScript Playground*, ou ao seu IDE, tente passar o cursor

sobre a variável `bestSong`. Você deve ver uma caixa de informação ou uma mensagem dizendo que o TypeScript inferiu que a variável `bestSong` é de tipo `string` (Figura 2.2).

```
let bestSong: string
let bestSong = Math.random() > 0.5
  ? "Chain of Fools"
  : "Respect";
```

Figura 2.2: O TypeScript relatando uma variável declarada com `let` como sendo de seu tipo literal `string` a partir da expressão ternária.



Não se esqueça das diferenças entre objetos e primitivos em JavaScript: classes como `Boolean` e `Number` encapsulam seus equivalentes primitivos. Geralmente, a prática recomendada no TypeScript é usar os nomes em letras minúsculas, como em `boolean` e `number`, respectivamente.

## Sistemas de tipos

Um *sistema de tipos* é o conjunto de regras que permite que uma linguagem de programação saiba quais tipos as estruturas de um programa podem ter.

Basicamente, o sistema de tipos do TypeScript funciona da seguinte forma:

- Lendo o código e detectando todos os tipos e valores que existem.
- Para cada valor, vendo que tipo sua declaração inicial indica que ele pode conter.
- Para cada valor, vendo todas as maneiras de como ele é usado posteriormente no código.
- Alertando ao usuário se o uso de um valor não for correspondente ao do seu tipo.

Percorreremos esse processo de inferência de tipos detalhadamente.

Considere o trecho de código a seguir no qual o TypeScript está emitindo uma mensagem de erro de tipo relacionada a uma propriedade que está sendo chamada erroneamente como uma função:

```
let firstName = "Whitney";
firstName.length();
// ~~~~~
// This expression is not callable.
// Type 'Number' has no call signatures
```

O TypeScript chegou a esse alerta da seguinte forma:

1. Lendo o código e sabendo que há uma variável chamada `firstName`.
2. Concluindo que `firstName` é do tipo `string` porque seu valor inicial é uma `string`, "Whitney".
3. Vendo que o código está tentando acessar uma propriedade `.length` de `firstName` e chamá-lo como uma função.
4. Alertando que a propriedade `.length` de uma string é um número e não uma função (*não pode ser chamado como uma função*).

Entender o sistema de tipos do TypeScript é uma habilidade importante para a compreensão de código TypeScript. Os trechos de código deste capítulo e do restante do livro exibirão tipos cada vez mais complexos que o TypeScript conseguirá inferir a partir do código.

## Tipos de erros

Na criação de código TypeScript, os dois tipos de “erros” encontrados com mais frequência são:

### *De sintaxe*

Impedem que o TypeScript seja convertido para JavaScript.

### *De tipo*

Algo incompatível é detectado pelo verificador de tipos.

As diferenças entre os dois são importantes.

## Erros de sintaxe

Os erros de sintaxe ocorrem quando o TypeScript detecta uma sintaxe incorreta que ele não consegue entender como código. Esses erros impedem que o TypeScript consiga gerar apropriadamente a saída JavaScript a partir do arquivo. Dependendo da ferramenta e das configurações que você estiver usando para converter seu código

TypeScript, talvez consiga obter algum tipo de saída JavaScript (com as configurações padrão do `tsc` conseguirá). No entanto, se conseguir, provavelmente ela não terá o resultado que você esperava.

Esta entrada TypeScript tem um erro de sintaxe de um `let` inesperado:

```
let let wat;  
//      ~~~  
// Error: ',' expected.
```

Dependendo da versão do compilador TypeScript, a saída JavaScript compilada pode ser semelhante a algo como:

```
let let, wat;
```

 Embora o TypeScript faça o possível para exibir o código JavaScript independentemente de erros de sintaxe, o código da saída pode não ser o esperado. É melhor corrigir os erros de sintaxe antes de tentar executar a saída JavaScript.

## Erros de tipo

Os erros de tipo (Type Error) ocorrem quando a sintaxe é válida, mas o verificador de tipos do TypeScript detecta um erro nas tipagens do programa. Esses erros não impedem que a sintaxe TypeScript seja convertida para JavaScript. No entanto, eles costumam indicar que algo travará ou se comportará inesperadamente se for permitido que o código seja executado.

Você viu isso no Capítulo 1, “Do JavaScript ao TypeScript”, com o exemplo de `console.blub`, em que o código era sintaticamente válido, mas o TypeScript conseguiu detectar que possivelmente ele quebraria quando executado:

```
console.blub("Nothing is worth more than laughter.");  
//      ~~~~  
// Error: Property 'blub' does not exist on type 'Console'.
```

Ainda que o TypeScript possa exibir o código JavaScript apesar da presença de erros de tipo, geralmente estes são um sinal de que a saída JavaScript não poderá ser executada da maneira esperada. É melhor lê-los e considerar corrigir qualquer problema relatado antes de executar o

JavaScript.

 Alguns projetos são configurados para bloquear códigos em execução durante o desenvolvimento até que todos os erros de tipo – e não apenas de sintaxe – do TypeScript sejam corrigidos. Muitos desenvolvedores, entre eles eu, acham isso tedioso e desnecessário. A maioria dos projetos tem uma maneira de não ser bloqueada, como com o arquivo `tsconfig.json` e as opções de configuração abordadas no Capítulo 3, “Opções de configuração”.

## Capacidade de atribuição

O TypeScript lê os valores iniciais das variáveis para determinar qual tipo elas podem ser. Se posteriormente ele detectar a atribuição de um novo valor à variável, verificará se o tipo do novo valor é o mesmo do da variável.

O TypeScript age de maneira correta quando atribui posteriormente um valor diferente, porém do mesmo tipo a uma variável. Se inicialmente uma variável tivesse, digamos, um valor `string`, seria aceitável atribuir a ela posteriormente outro valor do tipo `string`:

```
let firstName = "Carole";
firstName = "Joan";
```

Se o TypeScript detectar a atribuição de um tipo diferente, ele exibirá uma mensagem de erro de tipo. Não poderíamos, por exemplo, declarar inicialmente uma variável com um valor `string` e depois fornecer um `boolean`:

```
let lastName = "King";
lastName = true;
// Error: Type 'boolean' is not assignable to type 'string'.
```

A verificação que o TypeScript faz para saber se um valor pode ser fornecido a uma chamada de função ou a uma variável chama-se *capacidade de atribuição (assignability)*: ela verifica se esse valor é *atribuível* ao tipo esperado para o qual ele é passado. Esse será um termo importante em capítulos posteriores quando compararmos objetos mais complexos.

## Entendendo os erros de capacidade de atribuição

Os erros no formato “Type... is not assignable to type...” (O tipo... não é atribuível ao tipo...) serão um dos mais comuns que você verá ao escrever código TypeScript.

O primeiro tipo mencionado nessa mensagem de erro é o valor que o código está tentando atribuir a um destinatário. O segundo tipo mencionado é o destinatário para o qual está sendo atribuído o primeiro tipo. Por exemplo, quando escrevemos `lastName = true` no trecho de código anterior, estávamos tentando *atribuir* o valor `true` – tipo `boolean` – à variável de destino `lastName` – tipo `string`.

Você verá problemas de capacidade de atribuição cada vez mais complexos ao avançar na leitura deste livro. Lembre-se de lê-los com atenção para entender as diferenças relatadas entre o tipo real e o esperado. Isso facilitará trabalhar com o TypeScript quando ele fizer alertas sobre erros de sintaxe.

## Anotações de tipo

Uma variável pode não ter um valor inicial para o TypeScript ler. Ele não tentará descobrir o tipo inicial da variável a partir de usos posteriores. Por padrão considerará a variável como implicitamente do tipo `any`: indicando que ela pode ser qualquer coisa.

Variáveis cujo tipo inicial não pode ser inferido passam pelo que é chamado de *any modificável* (*evolving any*): em vez de impor algum tipo específico, o TypeScript muda o que conhece sobre o tipo da variável sempre que um novo valor é atribuído.

Aqui, a atribuição da variável `rocker` de tipo `any` modificável primeiro fornece uma `string`, o que significa que ela tem métodos de `string` como `toUpperCase`, mas depois isso evolui para `number`:

```
let rocker; // Type: any

rocker = "Joan Jett"; // Type: string
rocker.toUpperCase(); // Ok
```

```
rocker = 19.58; // Type: number
rocker.toPrecision(1); // Ok

rocker.toUpperCase();
// ~~~~~
// Error: 'toUpperCase' does not exist on type 'number'.
```

O TypeScript conseguiu detectar que estávamos chamando o método `toUpperCase()` em uma variável que modificou para o tipo `number`. No entanto, não informou se foi intencional a variável ter evoluído de `string` para `number`.

Permitir que as variáveis sejam do tipo `any` modificável – e usar o tipo `any` em geral – invalida parcialmente a finalidade da verificação de tipos do TypeScript! O TypeScript funciona melhor quando ele sabe que tipos os valores devem ter. Grande parte da verificação de tipos do TypeScript não pode ser aplicada a valores de tipo `any` porque eles não têm tipos conhecidos para serem verificados. O Capítulo 13, “Opções de configuração”, abordará como configurar alertas implícitos do TypeScript para `any`.

O TypeScript fornece uma sintaxe para a declaração do tipo de uma variável na qual não precisamos atribuir a ela um valor inicial, o que é chamado de *anotação de tipo (type annotation)*. Uma anotação de tipo é inserida após o nome de uma variável e inclui dois pontos seguidos do nome de um tipo.

A anotação de tipo a seguir indica que a variável `rocker` deve ser de tipo `string`:

```
let rocker: string;
rocker = "Joan Jett";
```

Essas anotações de tipo só existem para o TypeScript – elas não afetam o código de runtime e não são uma sintaxe JavaScript válida. Se você executar o `tsc` para compilar o código-fonte TypeScript para JavaScript, elas serão excluídas. Por exemplo, o caso anterior seria compilado para algo parecido com o JavaScript a seguir:

```
// output .js file
let rocker;
rocker = "Joan Jett";
```

A atribuição de um valor cujo tipo não seja correspondente ao tipo anotado da variável causará um erro de tipo.

Este trecho de código atribui um número a uma variável `rocker`, declarada anteriormente como de tipo `string`, causando um erro de tipo:

```
let rocker: string;  
rocker = 19.58;  
// Error: Type 'number' is not assignable to type 'string'.
```

Você verá nos próximos capítulos que as anotações de tipo ajudam o TypeScript a entender melhor o código, permitindo que ele forneça recursos mais adequados durante o desenvolvimento. O TypeScript contém vários elementos novos na sintaxe, como as anotações de tipo que só existem no sistema de tipos.



Nada que só exista no sistema de tipos é copiado para o JavaScript gerado. Os tipos do TypeScript não o afetarão.

## Anotações de tipo desnecessárias

As anotações de tipo permitem fornecer informações para o TypeScript que ele não conseguiria obter por conta própria. Também podemos usá-las em variáveis de tipos imediatamente inferíveis, mas não estaremos informando ao TypeScript nada que ele já não saiba.

A anotação de tipo `: string` a seguir é redundante porque o TypeScript pode inferir que `firstName` é de tipo `string`:

```
let firstName: string = "Tina";  
// ~~~~~ Does not change the type system...
```

Se você adicionar uma anotação de tipo a uma variável que tenha um valor inicial, o TypeScript verificará se o tipo coincide com o do valor da variável.

A variável `firstName` a seguir foi declarada como de tipo `string`, mas seu inicializador é o número `42`, o que o TypeScript vê como uma incompatibilidade:

```
let firstName: string = 42;  
// ~~~~~  
// Error: Type 'number' is not assignable to type 'string'.
```

Muitos desenvolvedores – inclusive eu – preferem não adicionar anotações de tipo em variáveis para as quais as anotações não alterariam nada. Escrever anotações de tipo manualmente pode ser difícil – principalmente quando elas mudam e para os tipos complexos que mostrarei posteriormente neste livro.

Pode ser útil incluir anotações de tipo explícitas para as variáveis com o objetivo de documentar o código de maneira mais clara e/ou para deixar o TypeScript protegido contra alterações acidentais no tipo da variável. Veremos em capítulos posteriores que as anotações de tipo explícitas podem passar explicitamente para o TypeScript informações que ele não conseguiria inferir normalmente.

## Formas dos tipos

O TypeScript faz mais do que apenas verificar se os valores atribuídos às variáveis têm os tipos originais. Ele também sabe quais propriedades devem existir para os objetos. Se você tentar acessar uma propriedade de uma variável, o TypeScript verificará se ela realmente existe para o tipo dessa variável.

Suponhamos que declarássemos uma variável `rapper` de tipo `string`. Posteriormente, quando usarmos essa variável, somente serão permitidas as operações que o TypeScript souber que funcionam com strings:

```
let rapper = "Queen Latifah";
rapper.length; // ok
```

Operações que o TypeScript não considerar que funcionam com strings não serão permitidas:

```
rapper.push('!');
// ~~~~
// Property 'push' does not exist on type 'string'.
```

Os tipos também podem ter formas mais complexas, principalmente os objetos. No trecho de código a seguir, o TypeScript sabe que o objeto `birthNames` não tem uma chave `middleName` e faz um alerta:

```
let cher = {
  firstName: "Cherilyn",
  lastName: "Sarkisian",
```

```
};

cher.middleName;
// ~~~~~
// Property 'middleName' does not exist on type
// '{ firstName: string; lastName: string; }'.
```

O conhecimento que o TypeScript tem das formas dos objetos permite que ele relate problemas referentes ao seu uso, em vez de apenas problemas de capacidade de atribuição. O Capítulo 4, “Objetos”, descreverá mais recursos poderosos do TypeScript relacionados aos objetos e seus tipos.

## Módulos

A linguagem de programação JavaScript não tinha uma especificação de como os arquivos poderão compartilhar código uns com os outros até pouco tempo. O ECMAScript 2015 adicionou os “ECMAScript modules”, ou ESM, para padronizar a sintaxe de importação (`import`) e exportação (`export`) entre arquivos.

Como referência, este arquivo módulo (module file) importa um valor (`value`) de um arquivo irmão `./values` e exporta uma variável `doubled`:

```
import { value } from "./values";

export const doubled = value * 2;
```

Para seguir a especificação ECMAScript, neste livro usarei a seguinte nomenclatura:

### *Módulo*

Um arquivo com uma instrução inicial `export` ou `import`.

### *Script*

Qualquer arquivo que não seja um módulo.

O TypeScript consegue trabalhar tanto com os arquivos módulos modernos quanto com arquivos mais antigos. Qualquer elemento que for declarado em um arquivo módulo só estará disponível nesse arquivo, a não ser que uma instrução `export` explícita o exporte. A declaração de uma variável em um módulo com o mesmo nome de uma variável declarada em outro arquivo não será considerada um conflito de nomeação (a

menos que um arquivo importe a variável do outro arquivo).

Os arquivos `a.ts` e `b.ts` a seguir são módulos que exportam sem qualquer problema uma variável de mesmo nome, `shared`. O arquivo `c.ts` causa um erro de tipo porque gera um conflito de nomeação entre uma variável `shared` importada e o valor que ele armazena:

```
// a.ts
export const shared = "Cher";
// b.ts
export const shared = "Cher";
// c.ts
import { shared } from "./a";
//      ~~~~~
// Error: Import declaration conflicts with local declaration of 'shared'.

export const shared = "Cher";
//      ~~~~~
// Error: Individual declarations in merged declaration
// 'shared' must be all exported or all local.
```

Se um arquivo for um script, o TypeScript o considerará de escopo global, o que significa que todos os scripts terão acesso ao seu conteúdo. Ou seja, variáveis declaradas em um arquivo script (script file) não podem ter o mesmo nome de variáveis declaradas em outros arquivos scripts.

Os arquivos `a.ts` e `b.ts` a seguir são considerados scripts porque não têm instruções `export` ou `import`. Isso significa que suas variáveis de mesmo nome entram em conflito como se tivessem sido declaradas no mesmo arquivo:

```
// a.ts
const shared = "Cher";
//      ~~~~~
// Cannot redeclare block-scoped variable 'shared'.
// b.ts
const shared = "Cher";
//      ~~~~~
// Cannot redeclare block-scoped variable 'shared'.
```

Se você encontrar erros “Cannot redeclare...” em um arquivo TypeScript, talvez isso tenha ocorrido por estar faltando adicionar uma instrução `export` ou `import`. De acordo com a especificação ECMAScript, se você

precisar que um arquivo seja um módulo sem uma instrução `export` ou `import`, pode adicionar `export {};` em algum local dele para forçá-lo a ser um módulo:

```
// a.ts and b.ts
const shared = "Cher"; // Ok

export {};
```



O TypeScript não reconhece os tipos de importações e exportações em arquivos TypeScript escritos com o uso de sistemas de módulo mais antigos como o CommonJS. Geralmente ele considera os valores retornados de funções `require` no estilo do CommonJS como de tipo `any`.

## Resumo

Neste capítulo, você viu como o sistema de tipos do TypeScript funciona:

- O que é um “tipo” e quais são os tipos primitivos reconhecidos pelo TypeScript.
- O que é um “sistema de tipos” e como o sistema de tipos do TypeScript entende o código.
- O que são os erros de tipo em comparação com os erros de sintaxe.
- Os tipos de variável inferidos e a capacidade de atribuição de variáveis.
- As anotações de tipo para declararmos os tipos de variáveis explicitamente e evitarmos os tipos `any` modificável.
- A verificação de propriedades de objetos nas formas dos tipos.
- O escopo da declaração de arquivos módulos do ECMAScript em comparação com os arquivos scripts.



Agora que você terminou de ler este capítulo, pratique o que aprendeu em <https://learningtypescript.com/the-type-system>.

*Por que o número e a string se separaram?*

*Um não era o tipo do outro.<sup>3</sup>*

---

1 N.T.: Original: *JavaScript's power / Comes from flexibility / Be careful with that!*

2 N.T.: Um popover é uma janela ou caixa de diálogo temporária que é acionada quando um usuário clica ou passa o cursor sobre um botão, ícone ou outra área definida.

3 N.T.: Original: *Why did the number and string break up? They weren't each other's types.*

# CAPÍTULO 3

## Uniões e literais

*Nada é constante.*

*Os valores podem mudar com o tempo  
(bem, exceto as constantes)<sup>1</sup>*

O Capítulo 2, “O sistema de tipos”, abordou o conceito de “sistema de tipos” e como ele lê os valores para conhecer os tipos das variáveis. Agora eu gostaria de introduzir dois conceitos importantes com os quais o TypeScript trabalha para fazer inferências com base nesses valores:

### *Uniões*

Expansão do tipo permitido de um valor para que use dois ou mais tipos possíveis.

### *Estreitamento*

Redução do tipo permitido de um valor para que ele *não* use um ou mais tipos possíveis.

Atuando em conjunto, as uniões e o estreitamento (narrowing) são conceitos poderosos que permitem que o TypeScript faça inferências embasadas sobre o código que muitas linguagens convencionais não conseguem fazer.

### Tipos união (union)

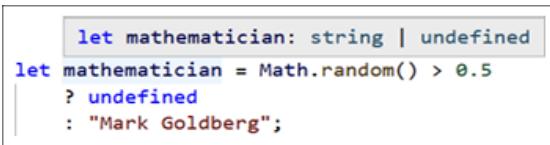
Veja esta variável `mathematician`:

```
let mathematician = Math.random() > 0.5
  ? undefined
  : "Mark Goldberg";
```

Qual é o tipo de `mathematician`?

Ele não é apenas `undefined` nem somente `string`, ainda que os dois sejam tipos possíveis. `mathematician` pode ser *ou* `undefined` ou `string`. Um tipo que se enquadre na classificação “OU” (either or) é chamado de tipo *união* (*union type*). Os tipos união são um conceito maravilhoso que nos permite manipular casos de códigos em que não sabemos exatamente qual é o tipo de um valor, mas sabemos que há duas ou mais opções para ele.

O TypeScript representa os tipos união usando o operador `|` (pipe) entre os valores, ou *constituíntes*, possíveis. O tipo da variável `mathematician` é considerado como `string | undefined`. Se passarmos o cursor sobre a variável exibiremos seu tipo como `string | undefined` (Figura 3.1).



```
let mathematician: string | undefined
let mathematician = Math.random() > 0.5
? undefined
: "Mark Goldberg";
```

A screenshot of a code editor showing a tooltip for the variable 'mathematician'. The tooltip displays the type annotation 'string | undefined'. The code below the tooltip shows a conditional assignment where 'mathematician' is assigned either 'undefined' or the string 'Mark Goldberg' based on a random value from Math.random().

Figura 3.1: O TypeScript relatando que a variável `mathematician` é de tipo `string | undefined`.

## Declaração de tipos união

Os tipos união são um exemplo de uma situação na qual pode ser útil fornecer uma anotação de tipo explícita para uma variável mesmo que ela tenha um valor inicial. Neste exemplo, inicialmente a variável `thinker` é de tipo `null`, mas ela também pode conter uma `string`. Se fornecermos para ela uma anotação de tipo explícita `string | null`, o TypeScript permitirá que a variável receba valores de tipo `string`:

```
let thinker: string | null = null;

if (Math.random() > 0.5) {
    thinker = "Susanne Langer"; // Ok
}
```

As declarações de tipo união podem ser inseridas em qualquer local onde você poderia declarar um tipo com uma anotação.



A ordem de uma declaração de tipo união não é importante. Você pode escrever `boolean | number` ou `number | boolean` e o TypeScript tratará

as duas formas da mesma maneira.

## Propriedades da união

Quando é sabido que um valor é de um tipo união, o TypeScript só permite acessar propriedades que existam em todos os tipos possíveis da união. Ele exibirá uma mensagem de erro de verificação de tipo se você tentar acessar um tipo que não seja um dos possíveis.

No trecho de código a seguir, `physicist` é de tipo `number | string`. Embora `.toString()` exista nos dois tipos e, portanto, pode ser usada, `.toUpperCase()` e `.toFixed()` não podem porque `.toUpperCase()` não está presente no tipo `number` e `.toFixed()` não faz parte do tipo `string`:

```
let physicist = Math.random() > 0.5
  ? "Marie Curie"
  : 84;

physicist.toString(); // Ok

physicist.toUpperCase();
// ~~~~~
// Error: Property 'toUpperCase' does not exist on type 'string | number'.
// Property 'toUpperCase' does not exist on type 'number'.

physicist.toFixed();
// ~~~
// Error: Property 'toFixed' does not exist on type 'string | number'.
// Property 'toFixed' does not exist on type 'string'.
```

Restringir o acesso às propriedades que não existem em todos os tipos união é uma medida de segurança. Se o TypeScript não souber se um objeto é de um tipo que contém uma propriedade, ele considerará arriscado tentar usar essa propriedade. Ela pode não existir!

Para usar uma propriedade de um valor de tipo união que só exista em um subconjunto dos tipos possíveis, seu código precisará indicar para o TypeScript que o valor nesse local do código é de um desses tipos mais específicos: um processo chamado *estreitamento*.

## Estreitamento (narrowing)

O estreitamento ocorre quando o TypeScript infere pelo código que um valor é de um tipo mais específico do que aquele com o qual ele foi definido, declarado ou anteriormente inferido. Quando o TypeScript sabe que o tipo de um valor é mais restrito do que o que se conhecia anteriormente, ele permite tratar o valor como sendo do tipo mais específico. Uma verificação lógica que pode ser usada para o estreitamento de tipos chama-se *type guard*.

Abordaremos dois dos type guards mais comuns que o TypeScript pode usar para inferir o estreitamento de tipo a partir do código.

## Estreitamento por atribuição

Se você atribuir diretamente um valor a uma variável, o TypeScript estreitará o tipo da variável para o tipo desse valor.

Aqui, inicialmente a variável `admiral` é declarada como `number | string`, mas depois de receber o valor "Grace Hopper", o TypeScript sabe que ela deve ser um `string`:

```
let admiral: number | string;

admiral = "Grace Hopper";

admiral.toUpperCase(); // Ok: string

admiral.toFixed();
// ~~~~~
// Error: Property 'toFixed' does not exist on type 'string'.
```

O estreitamento por atribuição é usado quando uma variável recebe uma anotação de tipo união explícita e um valor inicial. O TypeScript entenderá que, embora posteriormente a variável possa receber um valor de qualquer um dos tipos união, ela começará apenas com o tipo de seu valor inicial.

No trecho de código a seguir, `inventor` é declarada como de tipo `number | string`, mas o TypeScript sabe que ela será imediatamente estreitada para uma `string` por causa de seu valor inicial:

```
let inventor: number | string = "Hedy Lamarr";
```

```
inventor.toUpperCase(); // Ok: string

inventor.toFixed();
//      ~~~~~
// Error: Property 'toFixed' does not exist on type 'string'.
```

## Verificações condicionais

Uma maneira comum de fazer o TypeScript estreitar o valor de uma variável é escrevendo uma instrução `if` que verifique se a variável é igual a um valor conhecido. O TypeScript é suficientemente inteligente para entender que dentro do corpo dessa instrução `if` a variável precisa ter o mesmo tipo do valor conhecido:

```
// Type of scientist: number | string
let scientist = Math.random() > 0.5
    ? "Rosalind Franklin"
    : 51;

if (scientist === "Rosalind Franklin") {
    // Type of scientist: string
    scientist.toUpperCase(); // Ok
}

// Type of scientist: number | string
scientist.toUpperCase();
//      ~~~~~
// Error: Property 'toUpperCase' does not exist on type 'string | number'.
// Property 'toUpperCase' does not exist on type 'number'.
```

O estreitamento com a lógica condicional mostra a lógica de verificação de tipos do TypeScript refletindo bons padrões de codificação JavaScript. Se uma variável puder ter um entre vários tipos, vamos querer garantir que seu tipo seja aquele do qual precisamos. O TypeScript está nos forçando a agir com prudência com o nosso código. Obrigado, TypeScript!

## Verificações com `typeof`

Além da verificação de valor direta, o TypeScript também reconhece o operador `typeof` no estreitamento de tipos de variáveis.

De forma semelhante ao exemplo de `scientist`, verificar se `typeof researcher` é igual a "string" indica para o TypeScript que o tipo de `researcher` deve ser `string`:

```
let researcher = Math.random() > 0.5
  ? "Rosalind Franklin"
  : 51;

if (typeof researcher === "string") {
  researcher.toUpperCase(); // Ok: string
}
```

As negações lógicas do operador `!` e das instruções `else` também funcionam:

```
if (!(typeof researcher === "string")) {
  researcher.toFixed(); // Ok: number
} else {
  researcher.toUpperCase(); // Ok: string
}
```

Esses trechos de código podem ser sobrepostos por uma instrução ternária, que também é suportada para o estreitamento de tipos:

```
typeof researcher === "string"
  ? researcher.toUpperCase() // Ok: string
  : researcher.toFixed(); // Ok: number
```

Independentemente da forma como forem escritas, as verificações com `typeof` são uma maneira prática e muito usada para estreitar tipos.

O verificador de tipos do TypeScript reconhece várias formas de estreitamento que veremos em capítulos posteriores.

## Tipos literais

Agora que mostrei os tipos união e o estreitamento para o trabalho com valores que podem ter dois ou mais tipos, gostaria de tomar a direção oposta introduzindo os *tipos literais*: versões mais específicas dos tipos primitivos.

Veja esta variável `philosopher`:

```
const philosopher = "Hypatia";
```

Qual é o tipo de `philosopher`?

Inicialmente, você poderia dizer `string` – e estaria certo. `philosopher` é realmente uma `string`.

No entanto, `philosopher` não é qualquer `string`. Ela tem o valor específico "Hypatia". Logo, o tipo da variável `philosopher` é tecnicamente o tipo mais específico "Hypatia".

Esse é o conceito de um *tipo literal*: o tipo de um valor que é sabidamente um valor específico de um primitivo, em vez de qualquer um dos valores desse primitivo. O tipo primitivo `string` representa o conjunto de todas as strings possíveis existentes; o tipo literal "Hypatia" representa apenas essa string específica.

Se você declarar uma variável como `const` e fornecer diretamente para ela um valor literal, o TypeScript inferirá que a variável tem esse valor literal como tipo. É por isso que, quando passamos o ponteiro do mouse sobre uma variável `const` com um valor literal inicial em um IDE como o VS Code, ele exibe o tipo da variável como sendo esse literal (Figura 3.2) em vez do primitivo mais genérico (Figura 3.3).

```
const mathematician: "Mark Goldberg"  
const mathematician = "Mark Goldberg";
```

Figura 3.2: O TypeScript relatando uma variável `const` como sendo especificamente do seu tipo literal.

```
let mathematician: string  
let mathematician = "Mark Goldberg";
```

Figura 3.3: O TypeScript relatando uma variável `let` como sendo genericamente do seu tipo primitivo.

Podemos considerar cada tipo *primitivo* como uma *união* de todos os valores *literais* correspondentes a ele possíveis. Em outras palavras, um tipo primitivo é o conjunto de todos os valores literais possíveis desse tipo. Exceto pelos tipos `boolean`, `null` e `undefined`, todos os outros primitivos, como `number` e `string`, têm um número infinito de tipos literais. Os tipos mais comuns que você encontrará em códigos TypeScript típicos serão apenas estes:

- `boolean`: apenas `true` | `false`
- `null` e `undefined`: os dois têm um único valor literal, eles próprios
- `number`: `0` | `1` | `2` | ... | `0.1` | `0.2` | ...
- `string`: `" "` | `"a"` | `"b"` | `"c"` | ... | `"aa"` | `"ab"` | `"ac"` | ...

As anotações de tipos união podem combinar literais e primitivos. A representação de um tempo de vida, por exemplo, poderia ser feita por qualquer tipo `number` ou por um entre dois casos extremos conhecidos:

```
let lifespan: number | "ongoing" | "uncertain";

lifespan = 89; // Ok
lifespan = "ongoing"; // Ok

lifespan = true;
// Error: Type 'true' is not assignable to
// type 'number | "ongoing" | "uncertain"'
```

## Capacidade de atribuição de literais

Você viu que tipos primitivos diferentes como `number` e `string` não são atribuíveis uns aos outros. Da mesma forma, tipos literais diferentes pertencentes ao mesmo tipo primitivo – por exemplo, `0` e `1` – não são atribuíveis entre si.

Neste exemplo, a variável `specificallyAda` é declarada como sendo do tipo literal `"Ada"`, logo, embora o valor `"Ada"` possa ser fornecido para ela, os tipos `"Byron"` e `string` não são atribuíveis:

```
let specificallyAda: "Ada";

specificallyAda = "Ada"; // Ok

specificallyAda = "Byron";
// Error: Type '"Byron"' is not assignable to type '"Ada"'.

let someString = ""; // Type: string

specificallyAda = someString;
// Error: Type 'string' is not assignable to type '"Ada"'.
```

No entanto, os tipos literais podem ser atribuídos aos seus tipos

primitivos correspondentes. Qualquer string literal específica continua sendo uma `string`.

Neste exemplo de código, o valor ":)", que é de tipo ":)", está sendo atribuído à variável `someString` anteriormente inferida como sendo de tipo `string`:

```
someString = ":)";
```

Quem poderia esperar que uma simples atribuição de variável seria tão teoricamente intensa?

## Verificação estrita de nulos

O poder das uniões com estreitamento usando literais é visível principalmente no trabalho com valores potencialmente indefinidos, uma área dos sistemas de tipos que o TypeScript chama de *verificação estrita de nulos* (*strict null checking*). O TypeScript fez parte do surgimento das linguagens de programação modernas que utilizam a verificação estrita de nulos para corrigir o temido “erro de um bilhão de dólares”.

### O erro de um bilhão de dólares

*Eu o chamo de erro de um bilhão de dólares. Foi a invenção da referência nula em 1965... Ela levou a inúmeras falhas, vulnerabilidades e travamentos de sistemas, o que pode ter causado um bilhão de dólares em problemas e prejuízos nos últimos 40 anos.*

TONY HOARE, 2009

O “erro de um bilhão de dólares” é um termo fácil de se lembrar e usado no setor empresarial para muitos sistemas de tipos que permitem que valores nulos sejam usados em locais que demandem um tipo diferente. Em linguagens sem a verificação estrita de nulos, um código como o deste exemplo, que atribui `null` a um `string`, é permitido:

```
const firstName: string = null;
```

Se você já trabalhou com uma linguagem tipada, como C++ ou Java, que cause o erro de um bilhão de dólares, pode ficar surpreso ao saber que algumas linguagens não permitem que algo assim ocorra. Se você nunca

trabalhou com uma linguagem que tenha a verificação estrita de nulos, achará estranho que haja linguagens que permitam que o erro ocorra!

O compilador TypeScript contém várias opções que permitem alterar como ele é executado. O Capítulo 13, “Opções de configuração” abordará as opções com detalhes. Uma das opções selecionáveis mais úteis, `strictNullChecks`, define se a verificação estrita de nulas será ativada. De modo geral, desativar `strictNullChecks` adiciona `| null | undefined` a todos os tipos do código, permitindo que todas as variáveis recebam `null` ou `undefined`.

Com a opção `strictNullChecks` configurada com `false`, o código a seguir é considerado totalmente type safe. No entanto, isso está errado; a variável `nameMaybe` pode ser `undefined` quando a função `.toLowerCase` for acessada a partir dela:

```
let nameMaybe = Math.random() > 0.5
  ? "Tony Hoare"
  : undefined;

nameMaybe.toLowerCase();
// Potential runtime error: Cannot read property 'toLowerCase' of undefined.
```

Se a verificação estrita de nulos estiver ativada, o TypeScript detectará o possível travamento no trecho de código:

```
let nameMaybe = Math.random() > 0.5
  ? "Tony Hoare"
  : undefined;

nameMaybe.toLowerCase();
// Error: Object is possibly 'undefined'.
```

Sem a verificação estrita de nulos, será mais difícil saber se o código está protegido contra erros causados por valores acidentalmente `null` ou `undefined`.

Geralmente, a prática recomendada para o TypeScript é a ativação da verificação estrita de nulos. Isso ajuda a evitar travamentos e a eliminar o erro de um bilhão de dólares.

## Estreitamento por verdades

Você deve se lembrar de que *verdadeiro*, ou ser *verdade*, no JavaScript é quando um valor é considerado `true` ao ser avaliado em um contexto `Boolean`, como com um operador `&&` ou uma instrução `if`. Todos os valores em JavaScript são verdades, exceto os definidos como *falsos*: `false`, `0`, `-0`, `NaN`, `""`, `null`, `undefined` e `NaN`.<sup>2</sup>

O TypeScript também pode estreitar o tipo de uma variável com uma verificação de verdade se apenas parte de seus valores puder ser verdadeiro. No trecho de código a seguir, a variável `geneticist` é de tipo `string | undefined`, e já que `undefined` é sempre falso, o TypeScript pode inferir que ela deve ser de tipo `string` dentro do corpo da instrução `if`:

```
let geneticist = Math.random() > 0.5
    ? "Barbara McClintock"
    : undefined;

if (geneticist) {
    geneticist.toUpperCase(); // Ok: string
}

geneticist.toUpperCase();
// Error: Object is possibly 'undefined'.
```

Os operadores lógicos que executam a verificação de verdade também funcionam, a saber, `&&` e `?:`:

```
geneticist && geneticist.toUpperCase(); // Ok: string | undefined
geneticist?.toUpperCase(); // Ok: string | undefined
```

Infelizmente, a verificação de verdadeiros não funciona no sentido inverso. Se tudo que soubermos sobre um valor `string | undefined` é que ele é falso, isso não nos informará se ele é uma string vazia ou se é `undefined`.

Aqui, a variável `biologist` é de tipo `false | string`, e embora possa ser estreitada para apenas `string` no corpo da instrução `if`, o corpo da instrução `else` sabe que ela também pode ser uma string se for igual a `""`:

```
let biologist = Math.random() > 0.5 && "Rachel Carson";

if (biologist) {
    biologist; // Type: string
} else {
    biologist; // Type: false | string
```

}

## Variáveis sem valores iniciais

Variáveis declaradas sem valor inicial usam como padrão `undefined` em JavaScript. Isso gera um caso extremo no sistema de tipos: E se você declarar uma variável como de um tipo que não inclua `undefined` e tentar usá-la antes de atribuir um valor?

O TypeScript é suficientemente astuto para saber que a variável será `undefined` até um valor ser atribuído a ela. Ele exibirá uma mensagem de erro especializada se você tentar usar essa variável, por exemplo, acessando uma de suas propriedades, antes de atribuir um valor:

```
let mathematician: string;

mathematician?.length;
// Error: Variable 'mathematician' is used before being assigned.

mathematician = "Mark Goldberg";
mathematician.length; // Ok
```

É bom ressaltar que essa mensagem não seria aplicável se o tipo da variável incluísse `undefined`. A inclusão de `| undefined` no tipo de uma variável indica para o TypeScript que ela não precisa ser definida antes do uso, já que `undefined` é um tipo válido para o seu valor.

O trecho de código anterior não emitiria nenhuma mensagem de erro se o tipo de `mathematician` fosse `string | undefined`:

```
let mathematician: string | undefined;

mathematician?.length; // Ok

mathematician = "Mark Goldberg";
mathematician.length; // Ok
```

## Aliases de tipos

A maioria dos tipos união que você verá nos códigos provavelmente só terá dois ou três constituintes. No entanto, em algumas situações você pode achar útil usar tipos união mais longos cuja digitação repetida pode

ser tediosa.

Cada uma das variáveis a seguir pode ser de um entre quatro tipos:

```
let rawDataFirst: boolean | number | string | null | undefined;
let rawDataSecond: boolean | number | string | null | undefined;
let rawDataThird: boolean | number | string | null | undefined;
```

O TypeScript inclui *aliases de tipos* para a atribuição de nomes mais fáceis a tipos reutilizados. Um alias de tipo começa com a palavra-chave `type`, um novo nome, = e depois qualquer tipo. Por convenção, os aliases de tipo recebem nomes em PascalCase:

```
type MyName = ...;
```

Os aliases de tipo atuam como uma operação copiar e colar no sistema de tipos. Quando o TypeScript vê um alias de tipo, ele age como se tivéssemos digitado o tipo real que o alias está referenciando. As anotações de tipo das variáveis anteriores poderiam ser reescritas para usar um alias para o tipo união longo:

```
type RawData = boolean | number | string | null | undefined;
```

```
let rawDataFirst: RawData;
let rawDataSecond: RawData;
let rawDataThird: RawData;
```

Ficou muito mais fácil de ler!

O alias de tipo é um recurso prático para ser usado no TypeScript sempre que seus tipos começarem a ficar complexos. Por enquanto, isso inclui apenas tipos união longos; posteriormente incluirá tipos de array, função e objeto.

## Os aliases de tipo não são JavaScript

Os aliases de tipo, como as anotações de tipo, não são compilados para a saída JavaScript. Eles existem apenas no sistema de tipos do TypeScript.

O trecho de código anterior seria compilado para algo semelhante a esse JavaScript:

```
let rawDataFirst;
let rawDataSecond;
let rawDataThird;
```

Já que os aliases de tipo só existem no sistema de tipos, você não pode referenciá-los em código de runtime. O TypeScript informará com um erro de tipo se você estiver tentando acessar algo que não existirá no runtime:

```
type SomeType = string | undefined;

console.log(SomeType);
//           ~~~~~
// Error: 'SomeType' only refers to a type, but is being used as a value here.
```

Os aliases de tipo só existem como uma estrutura do tempo de desenvolvimento.

## Combinando aliases de tipo

Os aliases de tipo podem referenciar outros aliases de tipo. Pode ser útil os aliases de tipo referenciarem uns aos outros, como quando um alias de tipo é uma união de tipos que inclui (e é um superconjunto de) os tipos união de outro alias de tipo.

Este tipo `IdMaybe` é uma união dos tipos de `Id` com `undefined` e `null`:

```
type Id = number | string;

// Equivalent to: number | string | undefined | null
type IdMaybe = Id | undefined | null;
```

Os aliases de tipo não precisam ser declarados em ordem de uso. Um alias de tipo declarado anteriormente pode referenciar um alias declarado posteriormente no arquivo.

O trecho de código anterior poderia ser reescrito de modo que `IdMaybe` viesse antes de `Id`:

```
type IdMaybe = Id | undefined | null; // Ok
type Id = number | string;
```

## Resumo

Neste capítulo, você estudou os tipos união e os tipos literais no TypeScript e como seu sistema de tipos pode inferir tipos mais específicos (mais restritos) a partir de como o código foi estruturado:

- Como os tipos união podem representar valores que sejam de um entre dois ou mais tipos.
- A indicação explícita dos tipos união com anotações de tipo.
- Como o estreitamento de tipos reduz os tipos possíveis de um valor.
- A diferença entre variáveis `const` com tipos literais e variáveis `let` com tipos primitivos.
- O “erro de um bilhão de dólares” e como o TypeScript manipula a verificação estrita de nulos.
- O uso explícito de `| undefined` para representar valores que podem não existir.
- O uso implícito de `| undefined` para variáveis não atribuídas.
- O uso de aliases de tipo para evitar a digitação repetida de uniões de tipos longas.



Agora que você terminou de ler este capítulo, pratique o que aprendeu em <https://learningtypescript.com/unions-and-literals>.

*Por que as variáveis `const` são tão sérias?  
Elas se consideram muito literais.<sup>3</sup>*

---

<sup>1</sup> N.T.: Original: *Nothing is constant / Values may change over time / (well, except constants)*

<sup>2</sup> O objeto descontinuado `document.all` dos navegadores também é definido como falso em uma antiga peculiaridade de compatibilidade de navegador legado. Para os fins deste livro – e para nosso alívio como desenvolvedores – não precisamos nos preocupar com `document.all`.

<sup>3</sup> N.T.: Original: *Why are const variables so serious? They take themselves too literally.*

## CAPÍTULO 4

# Objetos

*Objetos literais*

*Um conjunto de chaves e valores*

*Cada um com seu próprio tipo<sup>1</sup>*

O Capítulo 3, “Unões e literais”, forneceu detalhes sobre os tipos união e literais: o trabalho com primitivos, como `boolean`, e com seus valores literais, como `true`. Esses primitivos são só uma amostra das formas de objetos complexas que normalmente o código JavaScript usa. Não seria possível usar o TypeScript se ele não conseguisse representar esses objetos. Este capítulo abordará como descrever formas de objetos complexas e como o TypeScript verifica sua capacidade de atribuição.

## Tipos objeto

Quando você criar um objeto literal com a sintaxe `{...}`, o TypeScript o considerará como um novo tipo de objeto, ou um novo tipo, de acordo com suas propriedades. Esse tipo objeto terá os mesmos nomes de propriedade e tipos primitivos dos valores do objeto. O acesso às propriedades do valor poderá ser realizado com `value.member` ou com a sintaxe equivalente `value['member']`.

O TypeScript sabe que o tipo da variável `poet` a seguir é o de um objeto com duas propriedades: `born`, de tipo `number`, e `name`, de tipo `string`. Seria permitido acessar essas propriedades, mas tentar acessar outro nome de propriedade causaria um erro de tipo por esse nome não existir:

```
const poet = {  
    born: 1935,  
    name: "Mary Oliver",
```

```
};

poet['born']; // Type: number
poet.name; // Type: string

poet.end;
// ~~~
// Error: Property 'end' does not exist on
// type '{ name: string; start: number; }'.
```

Os tipos de objeto são um conceito básico que define como o TypeScript entende o código JavaScript. Todo valor diferente de `null` e `undefined` tem um conjunto de propriedades pertencente à forma do seu tipo e, portanto, o TypeScript precisa conhecer o tipo de objeto de cada valor para confirmá-lo.

## Declaração de tipos de objeto

É ótimo poder inferir os tipos diretamente a partir de objetos existentes, mas uma hora você vai querer declarar um tipo de objeto explicitamente. Será preciso uma maneira de descrever uma forma de objeto separadamente dos objetos que a satisfazem.

Os tipos de objeto podem ser descritos com o uso de uma sintaxe semelhante a dos objetos literais, mas com tipos em vez de valores para os campos. É a mesma sintaxe que o TypeScript exibe em mensagens de erro sobre a capacidade de atribuição de tipos.

Esta variável `poetLater` tem o mesmo tipo do exemplo anterior com `name: string` e `born: number`:

```
let poetLater: {
  born: number;
  name: string;
};

// Ok
poetLater = {
  born: 1935,
  name: "Mary Oliver",
};

poetLater = "Sappho";
```

```
// Error: Type 'string' is not assignable to
// type '{ born: number; name: string; }'
```

## Apelidos de tipos

Escrever repetidamente tipos objeto como `{ born: number; name: string; }` cansaria rapidamente. É mais comum usar aliases de tipo para a atribuição de um nome a cada forma de tipo.

O trecho de código anterior poderia ser reescrito com `type Poet`, o que traria o benefício adicional de tornar a mensagem de erro de capacidade de atribuição do TypeScript mais direta e legível:

```
type Poet = {
    born: number;
    name: string;
};

let poetLater: Poet;

// Ok
poetLater = {
    born: 1935,
    name: "Sara Teasdale",
};

poetLater = "Emily Dickinson";
// Error: Type 'string' is not assignable to 'Poet'.
```



A maioria dos projetos TypeScript prefere usar a palavra-chave `interface` para descrever tipos de objeto, um recurso que não abordarei antes do Capítulo 7, “Interfaces”. Os apelidos de tipos (Aliased object types) e as interfaces são quase idênticos: tudo o que é apresentado neste capítulo também é aplicável às interfaces.

Estou abordando esses tipos de objeto agora porque saber como o TypeScript interpreta objetos literais é uma parte importante da aprendizagem do sistema de tipos da linguagem. Esses conceitos continuarão sendo importantes quando passarmos para os recursos na próxima seção deste livro.

## Tipagem estrutural

O sistema de tipos do TypeScript é *estruturalmente tipado*, o que significa que qualquer valor que atenda a um tipo poderá ser usado como valor dele. Em outras palavras, quando você declarar que um parâmetro ou variável é de um tipo objeto específico, estará dizendo ao TypeScript que, independentemente do objeto usado, ele precisará ter essas propriedades.

Os apelidos de tipos `WithFirstName` e `WithLastName` a seguir declaram apenas uma propriedade de tipo `string`. A variável `hasBoth` contém os dois membros – ainda que não tenha sido declarada explicitamente assim – logo, pode ser fornecida para variáveis declaradas como sendo de um dos dois apelidos:

```
type WithFirstName = {
    firstName: string;
};

type WithLastName = {
    lastName: string;
};

const hasBoth = {
    firstName: "Lucille",
    lastName: "Clifton",
};

// Ok: `hasBoth` contains a `firstName` property of type `string`
let withFirstName: WithFirstName = hasBoth;

// Ok: `hasBoth` contains a `lastName` property of type `string`
let withLastName: WithLastName = hasBoth;
```

Tipagem estrutural não é o mesmo que duck typing<sup>2</sup>, termo que vem da frase “se anda como um pato (duck) e faz quack como um pato, provavelmente é um pato”.

- A tipagem estrutural ocorre quando existe um sistema estático verificando o tipo – no caso do TypeScript, o verificador de tipos.
- O duck typing ocorre quando nada verifica os tipos dos objetos até eles serem usados no runtime.

Resumindo: o *JavaScript* é *duck typed* (*usa o duck typing*) enquanto o *TypeScript* é estruturalmente tipado.

## Verificação de uso

Ao fornecer um valor para um local anotado com um tipo de objeto, o *TypeScript* verificará se o valor é atribuível a esse tipo de objeto. Para começar, o valor deve ter as propriedades requeridas do tipo de objeto. Se alguma propriedade obrigatória estiver faltando no objeto, o *TypeScript* emitirá uma mensagem de erro de tipo.

O apelido de tipo `FirstAndLastNames` a seguir requer que existam as propriedades `first` e `last`. Um objeto contendo ambas poderá ser usado em uma variável declarada como de tipo `FirstAndLastNames`, mas um objeto sem elas não poderá:

```
type FirstAndLastNames = {
    first: string;
    last: string;
};

// Ok
const hasBoth: FirstAndLastNames = {
    first: "Sarojini",
    last: "Naidu",
};

const hasOnlyOne: FirstAndLastNames = {
    first: "Sappho"
};
// Property 'last' is missing in type '{ first: string; }'
// but required in type 'FirstAndLastNames'.
```

Tipos que não sejam compatíveis entre as duas propriedades também não são permitidos. Os tipos de objeto especificam tanto os nomes das propriedades obrigatórias quanto os tipos que essas propriedades devem ter. Se uma propriedade do objeto não for correspondente, o *TypeScript* relatará um erro de tipo.

O tipo `TimeRange` a seguir espera que a propriedade `start` seja de tipo `Date`. O objeto `hasStartString` está causando um erro de tipo porque seu membro

`start` é de tipo `string`:

```
type TimeRange = {
  start: Date;
};

const hasStartString: TimeRange = {
  start: "1879-02-13",
  // Error: Type 'string' is not assignable to type 'Date'.
};
```

## Verificação de propriedades em excesso

O TypeScript relatará um erro de tipo se uma variável for declarada com um tipo de objeto e seu valor inicial tiver mais campos do que seu tipo descreve. Logo, declarar uma variável como tipo de objeto é uma maneira de fazer o verificador de tipos assegurar que ela só tenha os campos esperados nesse tipo.

A variável `poetMatch` a seguir tem os campos exatos descritos no apelido de tipo `Poet`, enquanto `extraProperty` causa um erro de tipo por ter uma propriedade adicional:

```
type Poet = {
  born: number;
  name: string;
}

// Ok: all fields match what's expected in Poet
const poetMatch: Poet = {
  born: 1928,
  name: "Maya Angelou"
};

const extraProperty: Poet = {
  activity: "walking",
  born: 1935,
  name: "Mary Oliver",
};
// Error: Type '{ activity: string; born: number; name: string; }'
// is not assignable to type 'Poet'.
// Object literal may only specify known properties,
// and 'activity' does not exist in type 'Poet'.
```

É bom ressaltar que as verificações de propriedades em excesso só são acionadas para objetos literais que estejam sendo criados em locais declarados como sendo de um tipo de objeto. O fornecimento de um objeto literal existente é ignorado pelas verificações de propriedades em excesso.

A variável `extraPropertyButOk` a seguir não gera um erro de tipo referente ao tipo `Poet` do exemplo anterior porque seu valor inicial corresponde estruturalmente a `Poet`:

```
const existingObject = {  
    activity: "walking",  
    born: 1935,  
    name: "Mary Oliver",  
};  
  
const extraPropertyButOk: Poet = existingObject; // ok
```

As verificações de propriedades em excesso serão acionadas quando um novo objeto for criado em um local que demande que ele seja de um tipo de objeto específico, o que, como você verá em capítulos posteriores, inclui membros de arrays, campos de classes e parâmetros de funções. Remover as propriedades em excesso é outra maneira de o TypeScript ajudar a assegurar que seu código esteja limpo e faça o esperado. Geralmente, as propriedades em excesso não declaradas em seus tipos de objeto são nomes digitados incorretamente ou código não usado.

## Tipos de objeto aninhados

Assim como os objetos JavaScript podem ser aninhados como propriedades de outros objetos, os tipos de objeto do TypeScript devem poder representar tipos de objeto aninhados no sistema de tipos. A sintaxe é igual a que já vimos, mas com um tipo de objeto `{ ... }` em vez do nome de um primitivo.

O tipo `Poem` é declarado para ser um objeto cuja propriedade `author` deve ter `firstName: string` e `lastName: string`. A variável `poemMatch` é atribuível a `Poem` porque contém essa estrutura, enquanto `poemMismatch` não é atribuível porque sua propriedade `author` inclui `name` em vez de `firstName` e `lastName`:

```
type Poem = {
```

```

author: {
    firstName: string;
    lastName: string;
},
name: string;
};

// Ok
const poemMatch: Poem = {
    author: {
        firstName: "Sylvia",
        lastName: "Plath",
    },
    name: "Lady Lazarus",
};

const poemMismatch: Poem = {
    author: {
        name: "Sylvia Plath",
    },
    // Error: Type '{ name: string; }' is not assignable
    // to type '{ firstName: string; lastName: string; }'.
    // Object literal may only specify known properties, and 'name'
    // does not exist in type '{ firstName: string; lastName: string; }'.
    name: "Tulips",
};

```

Outra maneira de escrever o tipo `Poem` é inserir a forma da propriedade `author` em seu próprio apelido de tipo, `Author`. Inserir tipos aninhados em seus próprios apelidos de tipo também ajuda o TypeScript a fornecer mensagens de erro de tipo mais informativas. Nesse caso, ela exibiria '`Author`' em vez de '`{ firstName: string; lastName: string; }`':

```

type Author = {
    firstName: string;
    lastName: string;
};

type Poem = {
    author: Author;
    name: string;
};

```

```
const poemMismatch: Poem = {
  author: {
    name: "Sylvia Plath",
  },
  // Error: Type '{ name: string; }' is not assignable to type 'Author'.
  //   Object literal may only specify known properties,
  //   and 'name' does not exist in type 'Author'.
  name: "Tulips",
};
```



É uma boa ideia mover os tipos de objeto aninhados para um tipo com um nome próprio como nesse exemplo para o código e as mensagens de erro do TypeScript ficarem mais legíveis.

Você verá em capítulos posteriores que os membros dos tipos de objeto podem ser de outros tipos como arrays e funções.

## Propriedades opcionais

As propriedades de um tipo de objeto não precisam ser todas obrigatórias. Pode incluir um `?`  antes de `:` na anotação de tipo de uma propriedade para indicar que ela é opcional.

O tipo `Book` exige apenas a propriedade `pages` e permite opcionalmente o uso de `author`. Os objetos que o adotarem poderão fornecer ou não `author`, contanto que forneçam `pages`:

```
type Book = {
  author?: string;
  pages: number;
};

// Ok
const ok: Book = {
  author: "Rita Dove",
  pages: 80,
};

const missing: Book = {
  author: "Rita Dove",
};
// Error: Property 'pages' is missing in type
// '{ author: string; }' but required in type 'Book'.
```

Lembre-se de que existe uma diferença entre propriedades opcionais e propriedades cujo tipo inclua `undefined` em uma união de tipos. Uma propriedade declarada como opcional com `?` pode não existir. Uma propriedade declarada como obrigatória e `| undefined` deve existir, mesmo se o valor for `undefined`.

A propriedade `editor` do tipo `Writers` a seguir pode ser ignorada na declaração de variáveis porque ela tem um `?` na sua declaração. A propriedade `author` não tem um `?`, logo, precisa estar presente, mesmo se o seu valor for apenas `undefined`:

```
type Writers = {
    author: string | undefined;
    editor?: string;
};

// Ok: author is provided as undefined
const hasRequired: Writers = {
    author: undefined,
};
const missingRequired: Writers = {};
// ~~~~~
// Error: Property 'author' is missing in type
// '{}' but required in type 'Writers'.
```

O Capítulo 7, “Interfaces” abordará mais informações sobre outros tipos de propriedades, enquanto o Capítulo 13, “Opções de Configuração” descreverá as configurações de rigidez (strictness) do TypeScript para propriedades opcionais.

## Uniões de tipos de objeto

É aceitável no código TypeScript descrevermos um tipo que possa ser de um ou mais tipos de objeto diferentes contendo algumas propriedades distintas. Além disso, o código pode ter de executar um estreitamento entre esses tipos de objeto de acordo com o valor de uma propriedade.

### Uniões de tipos de objeto inferidas

Se uma variável receber um valor inicial que possa ser de um entre vários

tipos objeto, o TypeScript inferirá seu tipo como uma união de tipos objeto. Esse tipo união terá um constituinte para cada uma das formas de objeto possíveis. Todas as propriedades possíveis do tipo estarão presentes em todos esses constituintes, mas serão consideradas de tipos opcionais, com `?`, em qualquer tipo que não tiver um valor inicial para elas.

O valor desta variável `poem` tem sempre uma propriedade `name` de tipo `string` e pode ou não ter as propriedades `pages` e `rhymes`:

```
const poem = Math.random() > 0.5
  ? { name: "The Double Image", pages: 7 }
  : { name: "Her Kind", rhymes: true };
// Type:
// {
//   name: string;
//   pages: number;
//   rhymes?: undefined;
// }
// |
// {
//   name: string;
//   pages?: undefined;
//   rhymes: boolean;
// }

poem.name; // string
poem.pages; // number | undefined
poem.rhymes; // boolean | undefined
```

## Uniões de tipos objeto explícitas

Alternativamente, você pode ser mais explícito com seus tipos de objeto sendo explícito ao declarar sua união. Isso requer escrever um pouco mais de código, porém apresenta a vantagem de dar mais controle sobre os tipos objeto. Especificamente, se o tipo de um valor for uma união de tipos de objeto, o sistema de tipos do TypeScript só permitirá o acesso a propriedades que existirem em todos os tipos da união.

Esta versão da variável `poem` anterior é explicitamente tipada para ser um tipo união que sempre terá a propriedade `name` com `pages` ou `rhymes`. O acesso à propriedade `name` é permitido porque ela sempre existirá, mas não

é garantido que `pages` e `rhymes` existirão:

```
type PoemWithPages = {
    name: string;
    pages: number;
};

type PoemWithRhymes = {
    name: string;
    rhymes: boolean;
};

type Poem = PoemWithPages | PoemWithRhymes;

const poem: Poem = Math.random() > 0.5
    ? { name: "The Double Image", pages: 7 }
    : { name: "Her Kind", rhymes: true };

poem.name; // Ok

poem.pages;
// ~~~~~
// Property 'pages' does not exist on type 'Poem'.
// Property 'pages' does not exist on type 'PoemWithRhymes'.

poem.rhymes;
// ~~~~~
// Property 'rhymes' does not exist on type 'Poem'.
// Property 'rhymes' does not exist on type 'PoemWithPages'.
```

Restringir o acesso a membros potencialmente não existentes dos objetos pode ser algo bom para a segurança do código. Se um valor puder ser de um entre vários tipos, não haverá garantias de que as propriedades que não existirem em todos os tipos existirão no objeto.

Assim como as uniões de tipos literais e/ou primitivos devem passar pelo estreitamento de tipos para o acesso a propriedades que não existam como constituintes de todos os tipos, você terá de estreitar essas uniões de tipos de objeto.

## Estreitamento de tipos de objeto

Se o verificador de tipos detectar que uma área do código só poderá ser

executada se um valor de um tipo união tiver uma propriedade específica, ele restringirá o tipo do valor apenas aos constituintes que contiverem essa propriedade. Em outras palavras, o estreitamento de tipos do TypeScript será aplicado a objetos se você verificar sua forma no código.

Continuando com o exemplo da variável `poem` explicitamente tipada, verifique se `"pages"` `in` `poem` age como um type guard do TypeScript para indicar que ela é de tipo `PoemWithPages`. Se `poem` não for de tipo `PoemWithPages`, deve ser `PoemWithRhymes`:

```
if ("pages" in poem) {  
    poem.pages; // Ok: poem is narrowed to PoemWithPages  
} else {  
    poem.rhymes; // Ok: poem is narrowed to PoemWithRhymes  
}
```

É bom ressaltar que o TypeScript não permitirá a verificação de existência verdadeira como em `if (poem.pages)`. A tentativa de acessar uma propriedade de objeto que pode não existir será considerada um erro de tipo, mesmo se usada de maneira que pareça se comportar como um type guard:

```
if (poem.pages) { /* ... */ }  
//      ~~~~~  
// Property 'pages' does not exist on type 'PoemWithPages | PoemWithRhymes'.  
// Property 'pages' does not exist on type 'PoemWithRhymes'.
```

## Uniões discriminadas

Outro formato popular de objeto de tipo união em JavaScript e TypeScript é aquele em que uma propriedade do objeto indica que forma ele tem. Essa espécie de forma de tipo chama-se *união discriminada* e a propriedade cujo valor indica o tipo do objeto é o *discriminante*. O TypeScript pode executar o estreitamento de tipos para um código que garanta os tipos em propriedades discriminantes.

Por exemplo, este tipo `Poem` descreve um objeto que pode ser de um novo tipo `PoemWithPages` ou de um novo tipo `PoemWithRhymes` e a propriedade `type` indica qual é o tipo. Se `poem.type` for `"pages"`, o TypeScript poderá inferir que o tipo de `poem` deve ser `PoemWithPages`. Sem esse estreitamento de tipo, nenhuma propriedade é garantida de existir no valor:

```

type PoemWithPages = {
    name: string;
    pages: number;
    type: 'pages';
};

type PoemWithRhymes = {
    name: string;
    rhymes: boolean;
    type: 'rhymes';
};

type Poem = PoemWithPages | PoemWithRhymes;

const poem: Poem = Math.random() > 0.5
    ? { name: "The Double Image", pages: 7, type: "pages" }
    : { name: "Her Kind", rhymes: true, type: "rhymes" };

if (poem.type === "pages") {
    console.log(`It's got pages: ${poem.pages}`); // Ok
} else {
    console.log(`It rhymes: ${poem.rhymes}`);
}

poem.type; // Type: 'pages' | 'rhymes'

poem.pages;
// ~~~~
// Error: Property 'pages' does not exist on type 'Poem'.
// Property 'pages' does not exist on type 'PoemWithRhymes'.

```

As uniões discriminadas são meu recurso favorito do TypeScript porque elas combinam lindamente um padrão comum e elegante do JavaScript com o estreitamento de tipos do TypeScript. O Capítulo 10, “Tipos genéricos” e os projetos associados mostrarão mais sobre o uso de uniões discriminadas para operações de dados de tipo genérico.

## Tipos interseção

Os tipos união do TypeScript, caracterizados pelo operador `|`, representam o tipo de um valor que pode ser de um entre dois ou mais tipos diferentes. Assim como o operador de runtime `|` do JavaScript age

como contrapartida de seu operador `&`, o TypeScript permite representar um tipo que ao mesmo tempo são vários: um *tipo interseção*, caracterizado pelo operador `&`. Normalmente, os tipos interseção são usados com os apelidos de tipo para criar um novo tipo que combine vários tipos de objeto existentes.

Os tipos `Artwork` e `Writing` a seguir estão sendo usados para formar um tipo misto `WrittenArt` que tem as propriedades `genre`, `name` e `pages`:

```
type Artwork = {
    genre: string;
    name: string;
};

type Writing = {
    pages: number;
    name: string;
};

type WrittenArt = Artwork & Writing;
// Equivalent to:
// {
//     genre: string;
//     name: string;
//     pages: number;
// }
```

Os tipos interseção podem ser combinados com os tipos união, o que pode ser útil para descrever uniões discriminadas em um único tipo.

Este tipo `ShortPoem` tem sempre uma propriedade `author` e é uma união discriminada em uma propriedade `type`:

```
type ShortPoem = { author: string } & (
    | { kigo: string; type: "haiku"; }
    | { meter: number; type: "villanelle"; }
);

// ok
const morningGlory: ShortPoem = {
    author: "Fukuda Chiyo-ni",
    kigo: "Morning Glory",
    type: "haiku",
};
```

```

const oneArt: ShortPoem = {
    author: "Elizabeth Bishop",
    type: "villanelle",
};
// Error: Type '{ author: string; type: "villanelle"; }'
// is not assignable to type 'ShortPoem'.
// Type '{ author: string; type: "villanelle"; }' is not assignable to
// type '{ author: string; } & { meter: number; type: "villanelle"; }'.
// Property 'meter' is missing in type '{ author: string; type:
// "villanelle"; }'
// but required in type '{ meter: number; type: "villanelle"; }'.

```

## Perigos dos tipos interseção

Os tipos interseção são um conceito útil, mas é fácil eles serem usados de tal maneira que você mesmo ou o compilador TypeScript possa se confundir. Recomendo tentar manter o código o mais simples possível ao usá-los.

## Mensagens de erro de capacidade de atribuição longas

As mensagens de erro de capacidade de atribuição do TypeScript ficam muito mais difíceis de ler quando criamos tipos interseção complexos, como um que fosse combinado com um tipo união. Esse será um tema comum no sistema de tipos do TypeScript (e nas linguagens de programação tipadas em geral): quanto mais complexo você for, mais difícil será entender as mensagens do verificador de tipos.

No caso do tipo `ShortPoem` do trecho de código anterior, seria muito mais legível dividir o tipo em uma série de apelidos de tipos para permitir que o TypeScript exiba esses nomes:

```

type ShortPoemBase = { author: string };
type Haiku = ShortPoemBase & { kigo: string; type: "haiku" };
type Villanelle = ShortPoemBase & { meter: number; type: "villanelle" };
type ShortPoem = Haiku | Villanelle;

const oneArt: ShortPoem = {
    author: "Elizabeth Bishop",
    type: "villanelle",
};

```

```
// Type '{ author: string; type: "villanelle"; }'  
// is not assignable to type 'ShortPoem'.  
// Type '{ author: string; type: "villanelle"; }'  
// is not assignable to type 'Villanelle'.  
// Property 'meter' is missing in type  
// '{ author: string; type: "villanelle"; }'  
// but required in type '{ meter: number; type: "villanelle"; }'.
```

## never

Também é fácil usar incorretamente os tipos interseção e criar um tipo impossível com eles. Tipos primitivos não podem ser reunidos como constituintes de um tipo interseção porque é impossível um valor ser de vários tipos primitivos ao mesmo tempo. Tentar reunir dois tipos primitivos com `&` resultará no tipo `never`, representado pela palavra-chave `never`:

```
type NotPossible = number & string;  
// Type: never
```

A palavra-chave e o tipo `never` são o que as linguagens de programação chamam de *bottom type*, ou tipo vazio. Um tipo vazio é aquele que não pode ter valores possíveis e não pode ser alcançado. Nenhum tipo pode ser fornecido para um local cujo tipo seja um tipo vazio:

```
let notNumber: NotPossible = 0;  
// ~~~~~  
// Error: Type 'number' is not assignable to type 'never'.  
  
let notString: never = "";  
// ~~~~~  
// Error: Type 'string' is not assignable to type 'never'.
```

Raramente, a maioria dos projetos TypeScript – se é que em algum momento isso ocorre – usa o tipo `never`. Ele surge de vez em quando para representar estados impossíveis no código. Quase sempre, entretanto, é provável que seja um erro pelo uso incorreto de tipos interseção. Vou abordá-lo com mais detalhes no Capítulo 15, “Operações com tipos”.

## Resumo

Neste capítulo, você expandiu seu conhecimento do sistema de tipos do

TypeScript para que possa trabalhar com objetos:

- Como o TypeScript interpreta tipos a partir de tipos objetos literais.
- Descrição de tipos objetos literais, incluindo as propriedades aninhadas e opcionais.
- Declarar, inferir e estreitar tipos com uniões de tipos objetos literais.
- Uniões discriminadas e discriminantes.
- Combinação de tipos objeto com tipos interseção.



Agora que você terminou de ler este capítulo, pratique o que aprendeu em <https://learningtypescript.com/objects>.

*Como um advogado declara seu tipo TypeScript?  
“Eu objecto!”<sup>3</sup>*

---

<sup>1</sup> N.T.: Original: *Object literals / A set of keys and values / Each with their own type*

<sup>2</sup> N.T.: Duck typing é um estilo de codificação de linguagens dinamicamente tipadas onde o tipo de uma variável não importa, contanto que seu comportamento seja o desejado.

<sup>3</sup> N.T.: Original: *How does a lawyer declare their TypeScript type? “I object!”*

**PARTE II**

## **Recursos**

# CAPÍTULO 5

# Funções

*Argumentos de funções*

*Em uma extremidade, na outra*

*Como tipo de retorno<sup>1</sup>*

No Capítulo 2, “O sistema de tipos”, você viu como usar anotações de tipo para anotar valores de variáveis. Agora, verá como fazer o mesmo com parâmetros de funções e tipos de retorno – e por que isso pode ser útil.

## Parâmetros de funções

Vejamos a função `sing` a seguir que recebe o parâmetro `song` e envia para o console:

```
function sing(song) {  
    console.log(`Singing: ${song}!`);  
}
```

Que tipo de valor o desenvolvedor que escreveu a função `sing` deseja que seja fornecido para o parâmetro `song`?

Seria uma `string`? Seria um objeto com uma sobrescrita do método `toString()`? Esse código tem um bug? *Quem sabe?!*

Sem informações de tipo explícitas declaradas, podemos nunca saber – o TypeScript o considerará de tipo `any`, ou seja, o tipo do parâmetro poderá ser qualquer coisa.

Assim como nas variáveis, o TypeScript permite declarar o tipo de parâmetros de funções com uma anotação de tipo. Podemos usar `: string` para informar ao TypeScript que o parâmetro `song` é de tipo `string`:

```
function sing(song: string) {  
    console.log(`Singing: ${song}!`);
```

```
}
```

Melhorou muito: agora sabemos que tipo `song` deve ter!

É bom ressaltar que você não precisa adicionar anotações de tipo apropriadas aos parâmetros de funções para que seu código tenha uma sintaxe TypeScript válida. O TypeScript pode alertá-lo com mensagens de erro de tipo, mas o JavaScript gerado será executado. Mesmo não tendo uma declaração de tipo no parâmetro `song`, o trecho de código anterior será convertido de TypeScript para JavaScript. O Capítulo 13, “Opções de Configuração” abordará como configurar os alertas do TypeScript sobre parâmetros que sejam implicitamente de tipo `any` como ocorre com `song`.

## Parâmetros obrigatórios

Ao contrário do JavaScript, que permite que as funções sejam chamadas com qualquer número de argumentos, o TypeScript presume que todos os parâmetros declarados em uma função sejam obrigatórios. Se uma função for chamada com um número errado de argumentos, ele emitirá um alerta com uma mensagem de erro de tipo. A contagem de argumentos do TypeScript entrará em cena se uma função for chamada com argumentos insuficientes ou em excesso.

Esta função `singTwo` requer dois parâmetros, logo, não é permitido passar um nem passar três argumentos:

```
function singTwo(first: string, second: string) {
  console.log(` ${first} / ${second}`);
}

// Logs: "Ball and Chain / undefined"
singTwo("Ball and Chain");
//           ~~~~~
// Error: Expected 2 arguments, but got 1.

// Logs: "I Will Survive / Higher Love"
singTwo("I Will Survive", "Higher Love"); // Ok

// Logs: "Go Your Own Way / The Chain"
singTwo("Go Your Own Way", "The Chain", "Dreams");
//           ~~~~~
// Error: Expected 2 arguments, but got 3.
```

Impor que os parâmetros obrigatórios sejam fornecidos para uma função ajuda a reforçar a segurança de tipo (type safety) assegurando que todos os valores de argumentos esperados existam dentro da função. Não assegurar que esses valores existam pode resultar em um comportamento inesperado no código, como a função `singTwo` anterior logando `undefined` ou ignorando um argumento.



O *parâmetro* é a declaração de uma função do que ela espera receber como argumento. O *argumento* é um valor fornecido a um parâmetro em uma chamada de função. No exemplo anterior, `first` e `second` são parâmetros, enquanto strings como "`Dreams`" são argumentos.

## Parâmetros opcionais

Você deve se lembrar de que em JavaScript, quando um parâmetro de função não é fornecido, o valor de seu argumento dentro da função por padrão é `undefined`. Existem situações em que não é necessário fornecer parâmetros de funções e o planejado para a função é que ela use o valor `undefined`. Não queremos que o TypeScript relate erros de tipo pelo não fornecimento de argumentos para esses parâmetros opcionais. O TypeScript permite anotar um parâmetro como opcional com a inclusão de um `?` antes de `:` em sua anotação de tipo – como nas propriedades opcionais de tipos de objetos.

Os parâmetros opcionais não precisam ser fornecidos para as chamadas de função. Logo, seus tipos sempre têm `| undefined` adicionado como tipo união.

Na função `announceSong` a seguir, o parâmetro `singer` é marcado como opcional. Seu tipo é `string | undefined`, e ele não precisa ser fornecido pelos chamadores da função. Se `singer` for fornecido, pode ser um valor `string` ou `undefined`:

```
function announceSong(song: string, singer?: string) {
  console.log(`Song: ${song}`);

  if (singer) {
    console.log(`Singer: ${singer}`);
  }
}
```

```
}

announceSong("Greensleeves"); // Ok
announceSong("Greensleeves", undefined); // Ok
announceSong("Chandelier", "Sia"); // Ok
```

Esses parâmetros opcionais podem sempre ser implicitamente `undefined`. No código anterior, inicialmente, `singer` é de tipo `string | undefined` e depois é estreitado apenas para `string` pela instrução `if`.

Parâmetros opcionais não são o mesmo que parâmetros com tipos união que incluem `| undefined`. Parâmetros que não são marcados como opcionais com um `?` devem ser sempre fornecidos, mesmo se o valor for explicitamente `undefined`.

O parâmetro `singer` desta função `announceSongBy` deve ser fornecido explicitamente. Ele pode ser um valor `string` ou `undefined`:

```
function announceSongBy(song: string, singer: string | undefined) { /* ... */ }

announceSongBy("Greensleeves");
// Error: Expected 2 arguments, but got 1.

announceSongBy("Greensleeves", undefined); // Ok
announceSongBy("Chandelier", "Sia"); // Ok
```

Qualquer parâmetro opcional de uma função deve ficar na última posição. Inserir um parâmetro opcional antes de um parâmetro obrigatório acionaria um erro de sintaxe do TypeScript:

```
function announceSinger(singer?: string, song: string) {}
//           ~~~~
// Error: A required parameter cannot follow an optional parameter.
```

## Parâmetros padrão

Os parâmetros opcionais do JavaScript podem receber um valor padrão com um `=` e um valor em sua declaração. Já que um valor é fornecido por padrão para esses parâmetros opcionais, seu tipo no TypeScript não terá implicitamente a união `| undefined` adicionada dentro da função. Mesmo assim, o TypeScript permitirá que a função seja chamada com argumentos ausentes ou `undefined` para esses parâmetros.

A inferência de tipos do TypeScript funciona para valores padrão de

parâmetros de função de maneira semelhante a como funciona para valores iniciais de variáveis. Se um parâmetro tiver um valor padrão sem ter uma anotação de tipo, o TypeScript inferirá seu tipo de acordo com esse valor padrão.

Na função `rateSong` a seguir, `rating` é inferido como de tipo `number`, mas é um tipo opcional `number | undefined` que chama a função:

```
function rateSong(song: string, rating = 0) {
  console.log(`#${song} gets ${rating}/5 stars!`);
}

rateSong("Photograph"); // Ok
rateSong("Set Fire to the Rain", 5); // Ok
rateSong("Set Fire to the Rain", undefined); // Ok

rateSong("At Last!", "100");
// ~~~~~
// Error: Argument of type '"100"' is not assignable
// to parameter of type 'number | undefined'.
```

## Parâmetros rest

Algumas funções do JavaScript são criadas para serem chamadas com qualquer número de argumentos. O operador spread ...<sup>2</sup> pode ser inserido no último parâmetro em uma declaração de função para indicar que qualquer argumento “rest”<sup>3</sup> passado para a função a partir desse parâmetro deve ser armazenado no mesmo array.

O TypeScript permite declarar os tipos desses parâmetros rest da mesma forma que os parâmetros comuns são declarados, exceto por ser preciso adicionar uma sintaxe [] ao final para indicar que trata-se de um array de argumentos.

Aqui, `singAllTheSongs` pode receber zero ou mais argumentos de tipo `string` para seu parâmetro rest `songs`:

```
function singAllTheSongs(singer: string, ...songs: string[]) {
  for (const song of songs) {
    console.log(`#${song}, by ${singer}`);
  }
}
```

```
singAllTheSongs("Alicia Keys"); // Ok
singAllTheSongs("Lady Gaga", "Bad Romance", "Just Dance", "Poker Face"); // Ok

singAllTheSongs("Ella Fitzgerald", 2000);
//                                ~~~~
// Error: Argument of type 'number' is not
// assignable to parameter of type 'string'.
```

Abordarei o trabalho com arrays no TypeScript no Capítulo 6, “Arrays”.

## Tipos de retorno

O TypeScript é astuto: se ele conhecer todos os valores que podem ser retornados por uma função, saberá que tipo ela retorna. Neste exemplo, o TypeScript detecta que `singSongs` retorna um tipo `number`:

```
// Type: (songs: string[]) => number
function singSongs(songs: string[]) {
    for (const song of songs) {
        console.log(`#${song}`);
    }

    return songs.length;
}
```

Se uma função tiver várias instruções `return` com diferentes valores, o TypeScript inferirá o tipo de retorno como sendo de uma união de todos os tipos retornados possíveis.

Esta função `getSongAt` seria inferida como retornando `string | undefined` porque os dois valores que ela pode retornar são de tipo `string` e `undefined`, respectivamente:

```
// Type: (songs: string[], index: number) => string | undefined
function getSongAt(songs: string[], index: number) {
    return index < songs.length
        ? songs[index]
        : undefined;
}
```

## Tipos de retorno explícitos

Como com as variáveis, geralmente recomendo que as pessoas não se importem em declarar explicitamente os tipos de retorno de funções com

anotações de tipo. No entanto, existem alguns casos em que isso pode ser útil, especificamente para funções:

- Você poderia querer impor que funções que retornem muitos valores sempre retornem o mesmo tipo de valor.
- O TypeScript se recusará a tentar inferir tipos de retorno de funções recursivas.
- Pode acelerar a verificação de tipos do TypeScript em projetos muito grandes – isto é, os que tiverem centenas de arquivos TypeScript ou mais.

As anotações de tipo de retorno das declarações de funções são inseridas após o `)` que vem depois da lista de parâmetros.

Em uma declaração de função, ela se encontra imediatamente antes de `:`:

```
function singSongsRecursive(songs: string[], count = 0): number {  
    return songs.length ? singSongsRecursive(songs.slice(1), count + 1) : count;  
}
```

Em arrow functions<sup>4</sup> (também conhecidas como lambdas), ela fica antes de `=>`:

```
const singSongsRecursive = (songs: string[], count = 0): number =>  
    songs.length ? singSongsRecursive(songs.slice(1), count + 1) : count;
```

Se uma instrução `return` de uma função retornar um valor não atribuível ao tipo de retorno da função, o TypeScript exibirá um alerta de capacidade de atribuição.

Aqui, a função `getSongRecordingDate` foi declarada explicitamente como retornando `Date | undefined`, mas uma de suas instruções `return` fornece incorretamente uma `string`:

```
function getSongRecordingDate(song: string): Date | undefined {  
    switch (song) {  
        case "Strange Fruit":  
            return new Date('April 20, 1939'); // Ok  
  
        case "Greensleeves":  
            return "unknown";  
            // Error: Type 'string' is not assignable to type 'Date'.  
  
        default:    }
```

```
        return undefined; // Ok
    }
}
```

## Tipos função

O JavaScript permite passar funções como valores. Isso significa que precisamos de uma maneira de declarar um tipo de um parâmetro ou variável destinado a conter uma função.

A sintaxe do tipo função é semelhante à de uma arrow function, mas com um tipo em vez do corpo.

O tipo desta variável `nothingInGivesString` descreve uma função sem parâmetros e retornando um valor `string`:

```
let nothingInGivesString: () => string;
```

O tipo da variável `inputAndOutput` descreve uma função com um parâmetro `string[]`, um parâmetro opcional `count` e um valor de retorno `number`:

```
let inputAndOutput: (songs: string[], count?: number) => number;
```

Geralmente, os tipos função são usados para descrever parâmetros de callback (parâmetros que são chamados como funções).

Por exemplo, o trecho de código de `runOnSongs` a seguir declara o tipo de seu parâmetro `getSongAt` como uma função que recebe `index: number` e retorna uma `string`. Passar `getSongAt` atende a esse tipo, mas `logSong` não é apropriada porque recebe uma `string` como seu parâmetro em vez de `number`:

```
const songs = ["Juice", "Shake It Off", "What's Up"];

function runOnSongs(getSongAt: (index: number) => string) {
  for (let i = 0; i < songs.length; i += 1) {
    console.log(getSongAt(i));
  }
}

function getSongAt(index: number) {
  return `${songs[index]}`;
}

runOnSongs(getSongAt); // Ok
```

```
function logSong(song: string) {
  return `${song}`;
}

runOnSongs(logSong);
//           ~~~~~
// Error: Argument of type '(song: string) => string' is not
// assignable to parameter of type '(index: number) => string'.
//   Types of parameters 'song' and 'index' are incompatible.
//   Type 'number' is not assignable to type 'string'.
```

A mensagem de erro de `runOnSongs(logSong)` é um exemplo de erro de capacidade de atribuição que inclui alguns níveis de detalhes. Ao alertar que dois tipos função não são atribuíveis um ao outro, normalmente o TypeScript fornece três níveis de detalhes, com níveis de especificidade cada vez maiores:

1. O primeiro nível de indentação exibe os dois tipos função.
2. O nível seguinte de indentação especifica que parte é incompatível.
3. O último nível de indentação é o alerta preciso de capacidade de atribuição da parte incompatível.

No trecho de código anterior, esses níveis são:

1. `logSong`: `(song: string) => string` é o tipo fornecido que está sendo atribuído ao destinatário `getSongAt: (index: number) => string`
2. O parâmetro `song` de `logSong` está sendo atribuído ao parâmetro `index` de `getSongAt`
3. O tipo `number` de `song` não é atribuível ao tipo `string` de `index`

 Inicialmente, as mensagens de erro de várias linhas do TypeScript podem parecer complexas. Lê-las linha a linha e entender o que cada parte quer transmitir ajuda muito a compreender o erro.

## Parênteses dos tipos função

Os tipos função podem ser inseridos em qualquer local em que outro tipo seria usado. Isso inclui os tipos união.

Nos tipos união, os parênteses podem ser usados para indicar que parte

de uma anotação é o retorno da função ou o tipo união mais provável:

```
// Type is a function that returns a union: string | undefined
let returnsStringOrUndefined: () => string | undefined;

// Type is either undefined or a function that returns a string
let maybeReturnsString: (() => string) | undefined;
```

Capítulos posteriores que introduzirão mais sintaxes de tipo mostraráo outros locais em que os tipos função devem ser inseridos em parênteses.

## Inferências do tipo do parâmetro

Seria complexo se precisássemos declarar os tipos dos parâmetros para cada função que escrevêssemos, incluindo funções inline usadas como parâmetros. Felizmente, o TypeScript pode inferir os tipos dos parâmetros de uma função fornecida para um local com um tipo declarado.

Esta variável `singer` é sabidamente uma função que recebe um parâmetro de tipo `string`, logo, obviamente o parâmetro `song` posteriormente atribuído a `singer` é uma `string`:

```
let singer: (song: string) => string;

singer = function (song) {
    // Type of song: string
    return `Singing: ${song.toUpperCase()}!`; // Ok
};
```

Funções passadas como argumentos para parâmetros com tipos função também terão os tipos dos seus parâmetros inferidos.

Por exemplo, aqui o TypeScript inferiu que os parâmetros `song` e `index` são respectivamente de tipo `string` e `number`:

```
const songs = ["Call Me", "Jolene", "The Chain"];

// song: string
// index: number
songs.forEach((song, index) => {
    console.log(`${song} is at index ${index}`);
});
```

## Apelidos de tipos função

Você deve se lembrar dos apelidos de tipo do Capítulo 3, “Unões e literais”. Eles também podem ser usados para tipos função.

Este tipo `StringToNumber` é o apelido (alias) de uma função que recebe uma `string` e retorna um tipo `number`, o que significa que ele pode ser usado posteriormente para descrever os tipos das variáveis:

```
type StringToNumber = (input: string) => number;

let stringToNumber: StringToNumber;

stringToNumber = (input) => input.length; // Ok

stringToNumber = (input) => input.toUpperCase();
// ~~~~~
// Error: Type 'string' is not assignable to type 'number'.
```

De forma semelhante, até mesmo os parâmetros das funções podem ser tipados com apelidos que referenciem um tipo função.

A função `usesNumberToString` a seguir possui um único parâmetro, tendo ele próprio um apelido de tipo função `NumberToString`:

```
type NumberToString = (input: number) => string;

function usesNumberToString(numberToString: NumberToString) {
  console.log(`The string is: ${numberToString(1234)}`);
}

usesNumberToString((input) => `${input}! Hooray!`); // Ok

usesNumberToString((input) => input * 2);
// ~~~~~
// Error: Type 'number' is not assignable to type 'string'.
```

Os apelidos de tipo são particularmente úteis para tipos função. Eles podem economizar muito espaço horizontal gasto com a digitação repetida de parâmetros e/ou tipos de retorno.

## Mais tipos de retorno

Agora examinaremos mais dois tipos de retorno: `void` e `never`.

## Retorno void

Algumas funções não são projetadas para retornar nenhum valor. Elas não têm instruções `return` ou só têm instruções `return` que não retornam um valor. O TypeScript nos permite usar a palavra-chave `void` para referenciar o tipo de retorno dessas funções que não retornam nada.

Funções cujo tipo de retorno é `void` não podem retornar um valor. A função `logSong` é declarada como retornando `void`, logo, não pode retornar um valor:

```
function logSong(song: string | undefined): void {
  if (!song) {
    return; // Ok
  }

  console.log(` ${song}`);

  return true;
// Error: Type 'boolean' is not assignable to type 'void'.
}
```

`void` pode ser útil como tipo de retorno na declaração de um tipo função. Quando usado na declaração de um tipo função, `void` indica que qualquer valor retornado pela função será ignorado.

Por exemplo, a variável `songLogger` representa uma função que recebe `song: string` e não retorna um valor:

```
let songLogger: (song: string) => void;

songLogger = (song) => {
  console.log(` ${songs}`);
};

songLogger("Heart of Glass"); // Ok
```

É bom ressaltar que, embora por padrão todas as funções JavaScript retornem `undefined`, quando nenhum valor real é retornado, `void` não é o mesmo que `undefined`. `void` indica que o tipo de retorno de uma função será ignorado, enquanto `undefined` é um valor literal a ser retornado. Tentar atribuir um valor de tipo `void` a um valor cujo tipo inclua `undefined` é um erro de tipo:

```

function returnsVoid() {
    return;
}

let lazyValue: string | undefined;

lazyValue = returnsVoid();
// Error: Type 'void' is not assignable to type 'string | undefined'.

```

A diferença entre os tipos de retorno `undefined` e `void` é útil principalmente para que seja ignorado qualquer valor retornado por uma função passada para um local cujo tipo tenha sido declarado como retornando `void`. Por exemplo, o método interno `forEach` dos arrays recebe um callback que retorna `void`. As funções fornecidas para `forEach` podem retornar qualquer valor. Na função `saveRecords` a seguir, `records.push(record)` retorna um tipo `number` (o valor retornado pela função `.push()` de um array), mesmo assim ele pode ser o valor retornado pela arrow function passada para `newRecords.forEach`:

```

const records: string[] = [];

function saveRecords(newRecords: string[]) {
    newRecords.forEach(record => records.push(record));
}

saveRecords(['21', 'Come On Over', 'The Bodyguard'])

```

O tipo `void` não é JavaScript. É uma palavra-chave do TypeScript usada para declarar tipos de retorno de funções. Lembre-se, é uma indicação de que o valor retornado por uma função não deve ser usado, e não um valor que possa ser ele próprio retornado.

## Retorno never

Além de não retornar um valor, algumas funções não são projetadas nem mesmo para retornar. Funções que nunca retornam são aquelas que sempre lançam um erro ou executam um loop infinito (esperamos que intencionalmente!).

Quando uma função é projetada para nunca retornar, a inclusão de uma anotação de tipo explícita : `never` indica que qualquer código que vier

após uma chamada a essa função não será executado. A função `fail` a seguir apenas lança um erro, logo, pode ajudar a análise de controle de fluxo do TypeScript no estreitamento do tipo de `param` para `string`:

```
function fail(message: string): never {
    throw new Error(`Invariant failure: ${message}.`);
}

function workWithUnsafeParam(param: unknown) {
    if (typeof param !== "string") {
        fail(`param should be a string, not ${typeof param}`);
    }
    // Here, param is known to be type string
    param.toUpperCase(); // Ok
}
```



`never` não é o mesmo que `void`. `void` é para uma função que não retorna nada. `never` é para uma função que nunca retorna.

## Sobrecargas de funções

Algumas funções JavaScript podem ser chamadas com conjuntos de parâmetros muito diferentes que não podem ser representados apenas por parâmetros opcionais e/ou rest. Essas funções podem ser descritas com uma sintaxe do TypeScript chamada *assinaturas de sobrecarga (overload signatures)*: declarar diferentes versões do nome, dos parâmetros e dos tipos de retorno da função várias vezes antes da *assinatura de implementação (implementation signature)* final e do corpo da função.

Para determinar se uma mensagem de erro de sintaxe deve ser emitida para uma chamada a uma função sobre carregada, o TypeScript examinará apenas as assinaturas de sobrecarga da função. A assinatura de implementação só é usada pela lógica interna da função.

A função `createDate` a seguir deve ser chamada com um único parâmetro, `timestamp`, ou com três parâmetros — `month`, `day` e `year`. É permitida uma chamada com qualquer uma dessas quantidades de argumentos, mas chamar com dois argumentos causaria um erro de tipo porque nenhuma assinatura de sobrecarga permite dois argumentos. Neste exemplo, as

duas primeiras linhas são assinaturas de sobrecarga e a terceira linha é a assinatura de implementação:

```
function createDate(timestamp: number): Date;
function createDate(month: number, day: number, year: number): Date;
function createDate(monthOrTimestamp: number, day?: number, year?: number) {
    return day === undefined || year === undefined
        ? new Date(monthOrTimestamp)
        : new Date(year, monthOrTimestamp, day);
}

createDate(554356800); // Ok
createDate(7, 27, 1987); // Ok

createDate(4, 1);
// Error: No overload expects 2 arguments, but overloads
// do exist that expect either 1 or 3 arguments.
```

Como ocorre com outras sintaxes do sistema de tipos, as assinaturas de sobrecarga são removidas na compilação do TypeScript para a saída JavaScript.

A função do trecho de código anterior seria compilada para um JavaScript semelhante a este:

```
function createDate(monthOrTimestamp, day, year) {
    return day === undefined || year === undefined
        ? new Date(monthOrTimestamp)
        : new Date(year, monthOrTimestamp, day);
}
```

 Geralmente, as sobrecargas de funções são usadas como último recurso para tipos função complexos e difíceis de descrever. Pode ser melhor manter as funções simples e evitar o uso de sobrecargas de função sempre que possível.

## Compatibilidade das assinaturas de chamadas

A assinatura de implementação usada para a implementação de uma função sobrecarregada é a que a implementação da função usa para os tipos dos parâmetros e o tipo de retorno. Logo, o tipo de retorno e cada parâmetro das assinaturas de sobrecarga de uma função devem ser atribuíveis ao mesmo índice em sua assinatura de implementação. Em

outras palavras, a assinatura de implementação tem de ser compatível com todas as assinaturas de sobrecarga.

A assinatura de implementação da função `format` a seguir declara seu primeiro parâmetro como uma `string`. Embora as duas primeiras assinaturas de sobrecarga sejam compatíveis por também serem de tipo `string`, o tipo `() => string` da terceira assinatura de sobrecarga não é compatível:

```
function format(data: string): string; // Ok
function format(data: string, needle: string, haystack: string): string; // Ok

function format(getData: () => string): string;
//           ~~~~~
// This overload signature is not compatible with its implementation signature.

function format(data: string, needle?: string, haystack?: string) {
    return needle && haystack ? data.replace(needle, haystack) : data;
}
```

## Resumo

Neste capítulo, você viu como os parâmetros e os tipos de retorno de uma função podem ser inferidos ou declarados explicitamente no TypeScript:

- Declaração de parâmetros de tipo função com anotações de tipo.
- Declaração de parâmetros opcionais, valores padrão e parâmetros rest para alteração do comportamento do sistema de tipos.
- Declaração de tipos de retorno de funções com anotações de tipo.
- Descrição de funções que não retornam um valor usável com o tipo `void`.
- Descrição de funções que não retornam com o tipo `never`.
- Uso de sobrecargas de funções para a descrição de várias assinaturas de chamadas de funções.



Agora que você terminou de ler este capítulo, pratique o que aprendeu em <https://learningtypescript.com/functions>.

*O que faz com que um projeto TypeScript seja adequado?  
Ele tem de atender à sua função.<sup>5</sup>*

---

1 N.T.: Original: *Function arguments / In one end, out the other / As a return type*

2 N.T.: O operador spread permite definir um número indefinido de parâmetros para uma função, array ou objeto.

3 N.T.: Os parâmetros rest permitem representar um número indefinido de argumentos como em um array.

4 N.T.: Uma arrow function possui uma sintaxe mais curta sendo parecida com as expressões lambda da linguagem C#.

5 N.T.: Original: *What makes a TypeScript project good? It functions well.*

# CAPÍTULO 6

## Arrays

*Arrays e tuplas  
Um flexível e a outra fixa  
Escolha sua aventura<sup>1</sup>*

Os arrays do JavaScript são muito flexíveis e podem conter qualquer combinação de valores:

```
const elements = [true, null, undefined, 42];

elements.push("even", ["more"]);
// Value of elements: [true, null, undefined, 42, "even", ["more"]]
```

Quase sempre, entretanto, cada array JavaScript deve conter apenas um tipo de valor. A inclusão de valores de um tipo diferente pode ser confusa para os leitores, ou pior, causar um erro que gere problemas no programa.

O TypeScript respeita a boa prática de aceitar um único tipo de dado por array, lembrando-se do tipo de dado que estava inicialmente dentro de um array e só permitindo que ele opere com esse tipo de dado.

Neste exemplo, o TypeScript sabe que inicialmente o array `warriors` continha valores de tipo `string`, portanto, embora a inclusão de mais valores de tipo `string` seja permitida, a inserção de algum outro tipo de dado não é:

```
const warriors = ["Artemisia", "Boudica"];

// Ok: "Zenobia" is a string
warriors.push("Zenobia");

warriors.push(true);
//           ~~~~
// Argument of type 'boolean' is not assignable to parameter of type 'string'.
```

Podemos considerar a inferência que o TypeScript faz de um tipo array a partir de seus membros iniciais semelhante a como ele infere os tipos das variáveis a partir de seus valores iniciais. Geralmente, o TypeScript tenta inferir os tipos usados no código de acordo com como os valores são atribuídos, e os arrays não são exceção.

## Tipos array

Como ocorre nas declarações de outras variáveis, as variáveis destinadas a armazenar arrays não precisam ter um valor inicial. Elas podem começar como `undefined` e posteriormente receber um array como valor.

O TypeScript precisa que você o informe que tipos de valores farão parte do array colocando uma anotação de tipo para a variável. A anotação de tipo de um array requer o tipo dos elementos do array seguido de `[]`:

```
let arrayOfNumbers: number[];  
  
arrayOfNumbers = [4, 8, 15, 16, 23, 42];
```



Os tipos array também podem ser escritos com uma sintaxe como `Array<number>` chamada *classe genérica* (*class generics*). A maioria dos desenvolvedores prefere a sintaxe mais simples `number[]`. As classes serão abordadas no Capítulo 8, “Classes”, e os genéricos no Capítulo 9, “Modificadores de tipo”.

## Tipos de array e função

Os tipos de array são um exemplo de contêiner de sintaxe no qual os tipos de função podem precisar de parênteses para declarar o que está no tipo da função ou não. Os parênteses podem ser usados para indicar que parte de uma anotação é o retorno da função e qual indica o tipo array.

O tipo `createStrings` a seguir, que é um tipo função, não é igual a `stringCreators`, que é um tipo array:

```
// Type is a function that returns an array of strings  
let createStrings: () => string[];  
  
// Type is an array of functions that each return a string
```

```
let stringCreators: (() => string)[];
```

## Arrays de tipo união

Você pode usar um tipo união para indicar que cada elemento de um array pode ser de um entre vários tipos.

No uso de tipos array com uniões, parênteses podem ter de ser usados para indicar que parte de uma anotação é o conteúdo do array e qual se refere ao tipo união. O uso de parênteses em tipos união de arrays é importante – os dois tipos a seguir não são iguais:

```
// Type is either a number or an array of strings
let stringOrArrayOfNumbers: string | number[];

// Type is an array of elements that are each either a number or a string
let arrayOfStringOrNumbers: (string | number)[];
```

O TypeScript saberá, a partir da declaração de um array, que ele é de tipo união se houver mais de um tipo de elemento. Em outras palavras, o tipo dos elementos de um array será a união de todos os tipos que os elementos puderem ter.

Aqui, `namesMaybe` é `(string | undefined)[]` porque pode tanto ser `string` quanto um valor `undefined`:

```
// Type is (string | undefined)[]
const namesMaybe = [
  "Aqualtune",
  "Blenda",
  undefined,
];
```

## Arrays any modificável

Se você não incluir uma anotação de tipo em uma variável configurada inicialmente com um array vazio, o TypeScript tratará o array como `any[]` modificável, o que significa que ele poderá receber qualquer conteúdo. Como no caso das variáveis de tipo `any` modificável, não recomendo o uso de arrays de tipo `any[]` modificável. Eles invalidam parcialmente os benefícios do verificador de tipos do TypeScript, permitindo a inclusão de valores potencialmente incorretos.

Inicialmente, este array `values` contém elementos `any`, é modificado para conter elementos `string` e é alterado novamente para incluir elementos `number | string`:

```
// Type: any[]
let values = [];

// Type: string[]
values.push('');

// Type: (number | string)[]
values[0] = 0;
```

Como com as variáveis, permitir que os arrays sejam de tipo `any` modificável – e usar o tipo `any` em geral – invalida parcialmente a finalidade da verificação de tipos do TypeScript. Ele funciona melhor quando sabe que tipos os valores devem ter.

## Arrays multidimensionais

Um array 2D, ou um array composto de arrays, tem dois pares de colchetes:

```
let arrayOfArraysOfNumbers: number[][][];

arrayOfArraysOfNumbers = [
  [1, 2, 3],
  [2, 4, 6],
  [3, 6, 9],
];
```

Um array 3D, ou um array composto de arrays que também são compostos de arrays, tem três pares de colchetes. Os arrays 4D têm quatro pares de colchetes. Os arrays 5D têm cinco pares de colchetes. É fácil adivinhar o que acontece com os arrays 6D e os de mais dimensões.

Esses tipos de arrays multidimensionais não introduzem nenhum conceito novo aos tipos array. Você pode considerar um array 2D como o tipo array original, que tem um par de colchetes no fim, e adiciona outro par de colchetes depois dele.

O array `arrayOfArraysOfNumbers` a seguir é de tipo `number[][][]`, que também pode ser representado com `(number[])[]`:

```
// Type: number[][]  
let arrayOfArraysOfNumbers: (number[][]);
```

## Membros do array

O TypeScript espera que o acesso típico baseado em índice para a recuperação dos membros de um array retorne um elemento do tipo desse array.

Este array `defenders` é de tipo `string[]`, logo, `defender` é uma `string`:

```
const defenders = ["Clarenza", "Dina"];
```

```
// Type: string  
const defender = defenders[0];
```

Os membros de arrays de tipo união também têm o mesmo tipo união.

Aqui, `soldiersOrDates` é de tipo `(string | Date)[]`, portanto, a variável `soldierOrDate` é de tipo `string | Date`:

```
const soldiersOrDates = ["Deborah Sampson", new Date(1782, 6, 3)];
```

```
// Type: Date | string  
const soldierOrDate = soldiersOrDates[0];
```

## Aviso: membros inconsistentes

A tipagem do TypeScript é famosa por ser tecnicamente *inconsistente*: quase sempre ela detecta os tipos corretamente, mas às vezes seu conhecimento dos tipos dos valores pode estar incorreto. Os arrays em particular são uma fonte de inconsistência na tipagem. Por padrão, o TypeScript presume que todos os acessos aos membros de um array retornarão um membro desse array, ainda que em JavaScript o acesso a um elemento com índice maior do que o tamanho do array resulte em `undefined`.

Este código não exibe alertas com as configurações padrão do compilador TypeScript:

```
function withElements(elements: string[]) {  
    console.log(elements[9001].length); // No type error  
}
```

```
withElements(["It's", "over"]);
```

Como leitores, podemos deduzir que ele quebrará no runtime com o alerta “`Cannot read property 'length' of undefined`”, mas intencionalmente o TypeScript não se certificará se os membros recuperados do array existem. Ele considerará `elements[9001]`, que aparece no trecho de código, como sendo de tipo `string`, e não `undefined`.



O TypeScript tem uma flag `--noUncheckedIndexedAccess` que torna as buscas em arrays mais restrita e type safe, mas ela é muito rígida e a maioria dos projetos não a utiliza. Não a abordarei neste livro. O Capítulo 12, “Uso de recursos do IDE” tem links de recursos que explicam todas as opções de configuração do TypeScript com detalhes.

## Spreads e Rests

Lembra-se dos parâmetros rest ... das funções que vimos no Capítulo 5, “Funções”? Os parâmetros rest e o spreading de arrays, os dois usados com o operador ..., são maneiras essenciais de interagir com arrays em JavaScript. O TypeScript entende ambos.

### Spreads

Os arrays podem ser reunidos com o uso do operador spread .... O TypeScript sabe que o array resultante terá valores que serão dos dois arrays da entrada.

Se os arrays da entrada tiverem o mesmo tipo, o array da saída também o terá. Se dois arrays de tipos diferentes forem reunidos para criar um novo array, este será deduzido com sendo um array de tipo união com elementos dos dois tipos originais.

Aqui, sabe-se que o array `conjoined` contém tanto valores que são de tipo `string` quanto valores que são de tipo `number`, logo, seu tipo é inferido como `(string | number)[]`:

```
// Type: string[]
const soldiers = ["Harriet Tubman", "Joan of Arc", "Khutulun"];
```

```
// Type: number[]
const soldierAges = [90, 19, 45];

// Type: (string | number)[]
const conjoined = [...soldiers, ...soldierAges];
```

## Spreading de parâmetros rest

O TypeScript reconhece e executará a verificação de tipos na prática JavaScript de spreading de um array como um parâmetro rest com o operador .... Os arrays usados como argumentos de parâmetros rest devem ter o mesmo tipo array do parâmetro rest.

A função `logWarriors` a seguir só recebe valores `string` para seu parâmetro `...names`. O spreading de um array de tipo `string[]` é permitido, mas de um tipo `number[]` não:

```
function logWarriors(greeting: string, ...names: string[]) {
  for (const name of names) {
    console.log(`#${greeting}, ${name}!`);
  }
}
const warriors = ["Cathay Williams", "Lozen", "Nzinga"];

logWarriors("Hello", ...warriors);

const birthYears = [1844, 1840, 1583];

logWarriors("Born in", ...birthYears);
// ~~~~~
// Error: Argument of type 'number' is not
// assignable to parameter of type 'string'.
```

## Tuplas

Embora teoricamente os arrays do JavaScript possam ter qualquer tamanho, pode ser útil usar um array de tamanho fixo – também conhecido como *tupla*. Os arrays tuplas têm um tipo conhecido específico em cada índice que pode ser mais específico do que um tipo união com todos os membros possíveis do array. A sintaxe para a declaração de um tipo tupla é parecida com a de um array literal, mas com tipos em vez dos

valores dos elementos.

Aqui, o array `yearAndWarrior` é declarado como sendo de tipo tupla com `number` no índice 0 e uma `string` no índice 1:

```
let yearAndWarrior: [number, string];

yearAndWarrior = [530, "Tomyris"]; // Ok

yearAndWarrior = [false, "Tomyris"];
//           ~~~~~
// Error: Type 'boolean' is not assignable to type 'number'.

yearAndWarrior = [530];
// Error: Type '[number]' is not assignable to type '[number, string]'.
// Source has 1 element(s) but target requires 2.
```

Geralmente, as tuplas são usadas em JavaScript junto com a desestruturação de arrays<sup>2</sup> para ser possível atribuir múltiplos valores de uma só vez, como na configuração de duas variáveis com valores iniciais de acordo com uma única condição.

Por exemplo, no caso a seguir o TypeScript reconheceria que `year` será sempre de tipo `number` e `warrior` será sempre uma `string`:

```
// year type: number
// warrior type: string
let [year, warrior] = Math.random() > 0.5
  ? [340, "Archidamia"]
  : [1828, "Rani of Jhansi"];
```

## Capacidade de atribuição da tupla

Os tipos tupla são tratados pelo TypeScript como sendo mais específicos do que os tipos array de tamanho variável. Isso significa que os tipos array de tamanho variável não são atribuíveis aos tipos tupla.

Aqui, embora nós, humanos, possamos ver `pairLoose` como o composto de `[boolean, number]`, o TypeScript infere que ele tem o tipo mais geral `(boolean | number)[]`:

```
// Type: (boolean | number)[]
const pairLoose = [false, 123];

const pairTupleLoose: [boolean, number] = pairLoose;
```

```
// ~~~~~
// Error: Type '(number | boolean)[]' is not
// assignable to type '[boolean, number]'.
// Target requires 2 element(s) but source may have fewer.
```

Se `pairLoose` tivesse sido declarado como `[boolean, number]`, a atribuição de seu valor a `pairTuple` teria sido permitida.

Tuplas de tamanhos diferentes também não são atribuíveis umas às outras, já que o TypeScript precisa saber quantos membros existem na tupla no caso de tipos tupla.

Aqui, `tupleTwoExtra` deve ter exatamente dois membros, logo, embora `tupleThree` comece com os membros corretos, seu terceiro membro a impede de ser atribuível a `tupleTwoExtra`:

```
const tupleThree: [boolean, number, string] = [false, 1583, "Nzinga"];

const tupleTwoExact: [boolean, number] = [tupleThree[0], tupleThree[1]];

const tupleTwoExtra: [boolean, number] = tupleThree;
// ~~~~~
// Error: Type '[boolean, number, string]' is
// not assignable to type '[boolean, number]'.
// Source has 3 element(s) but target allows only 2.
```

## Tuplas como parâmetros rest

Já que as tuplas são consideradas arrays com informações de tipo mais específicas para o tamanho e os tipos dos elementos, elas podem ser particularmente úteis para o armazenamento dos argumentos a serem passados para uma função. O TypeScript pode fornecer uma verificação de tipos precisa para tuplas passadas como parâmetros rest ....

Os parâmetros da função `logPair` a seguir são de tipo `string` e `number`. Tentar passar um valor de tipo `(string | number)[]` como argumento não seria type safe já que o conteúdo pode não coincidir: eles podem ser do mesmo tipo ou um de cada tipo na ordem errada. No entanto, se o TypeScript souber que o valor é uma tupla `[string, number]`, saberá que os valores casarão:

```
function logPair(name: string, value: number) {
  console.log(`#${name} has ${value}`);
}
```

```

const pairArray = ["Amage", 1];

logPair(...pairArray);
// Error: A spread argument must either have a
// tuple type or be passed to a rest parameter.

const pairTupleIncorrect: [number, string] = [1, "Amage"];

logPair(...pairTupleIncorrect);
// Error: Argument of type 'number' is not
// assignable to parameter of type 'string'.

const pairTupleCorrect: [string, number] = ["Amage", 1];

logPair(...pairTupleCorrect); // Ok

```

Se você quiser fazer experiências com suas tuplas de parâmetros rest, pode combiná-las com arrays para armazenar uma lista de argumentos para várias chamadas de função. A seguir, `trios` é um array de tuplas, no qual cada tupla também tem uma tupla como seu segundo membro. É sabido que `trios.forEach(trio => logTrio(...trio))` é seguro porque cada `...trio` tem os mesmos tipos de parâmetro de `logTrio`. `trios.forEach(logTrio)`, mas não é atribuível porque estamos tentando passar o array `[string, [number, boolean]]` inteiro como primeiro parâmetro, que é de tipo `string`:

```

function logTrio(name: string, value: [number, boolean]) {
  console.log(`#${name} has ${value[0]} (${value[1]})`);
}

const trios: [string, [number, boolean]][] = [
  ["Amanitore", [1, true]],
  ["Æthelflæd", [2, false]],
  ["Ann E. Dunwoody", [3, false]]
];

trios.forEach(trio => logTrio(...trio)); // Ok

trios.forEach(logTrio);
//           ~~~~~
// Argument of type '(name: string, value: [number, boolean]) => void'
// is not assignable to parameter of type

```

```
// '(value: [string, [number, boolean]], ...) => void'.
//   Types of parameters 'name' and 'value' are incompatible.
//     Type '[string, [number, boolean]]' is not assignable to type 'string'.
```

## Inferência de tuplas

Geralmente o TypeScript trata os arrays que são criados como arrays de tamanho variável e não tuplas. Quando ele detecta um array sendo usado como valor inicial de uma variável ou como o valor retornado por uma função, presume que seja um array de tamanho flexível em vez de uma tupla de tamanho fixo.

A função `firstCharAndSize` a seguir é inferida como retornando `(string | number)[]` e não `[string, number]`, porque esse é o tipo inferido para o array literal retornado:

```
// Return type: (string | number)[]
function firstCharAndSize(input: string) {
    return [input[0], input.length];
}

// firstChar type: string | number
// size type: string | number
const [firstChar, size] = firstCharAndSize("Gudit");
```

Existem duas maneiras comuns de o TypeScript indicar que um valor deve ser de um tipo tupla mais específico e não de um tipo array geral: os tipos tupla explícitos e as asserções `const`.

## Tipos tupla explícitos

Os tipos tupla podem ser usados em anotações de tipo, como na anotação do tipo de retorno de uma função. Se a função for declarada como retornando um tipo tupla e retornar um array literal, esse array literal será inferido como uma tupla em vez de um array mais geral de tamanho variável.

Esta versão da função `firstCharAndSizeExplicit` declara explicitamente que ela retorna uma tupla com uma `string` e um `number`:

```
// Return type: [string, number]
function firstCharAndSizeExplicit(input: string): [string, number] {
    return [input[0], input.length];
```

```
}

// firstChar type: string
// size type: number
const [firstChar, size] = firstCharAndSizeExplicit("Cathay Williams");
```

## Tuplas declaradas em asserções const

Digitar tipos tupla em anotações de tipo explícitas pode ser incômodo pelas mesmas razões da digitação de qualquer anotação de tipo explícita. É sintaxe adicional para você escrever e atualizar quando o código mudar.

Como alternativa, o TypeScript fornece um operador `as const` conhecido como *asseração const* que pode ser inserido após um valor. As asserções `const` solicitam ao TypeScript para usar a forma mais literal e somente leitura possível do valor ao inferir seu tipo. Se uma for inserida após um array literal, ela indicará que o array deve ser tratado como uma tupla:

```
// Type: (string | number)[]
const unionArray = [1157, "Tomoe"];

// Type: readonly [1157, "Tomoe"]
const readonlyTuple = [1157, "Tomoe"] as const;
```

É bom ressaltar que as asserções `as const` fazem mais do que apenas mudar de arrays de tamanho flexível para tuplas de tamanho fixo: elas também indicam para o TypeScript que a tupla é somente leitura e não pode ser usada em um local que espere ser permitido modificar o valor.

Neste exemplo, `pairMutable` pode ser modificado porque tem um tipo tupla explícito tradicional. No entanto, `as const` torna o valor não atribuível ao array mutável `pairAlsoMutable`, e os membros da constante `pairConst` não podem ser modificados:

```
const pairMutable: [number, string] = [1157, "Tomoe"];
pairMutable[0] = 1247; // Ok

const pairAlsoMutable: [number, string] = [1157, "Tomoe"] as const;
// ~~~~~
// Error: The type 'readonly [1157, "Tomoe"]' is 'readonly'
// and cannot be assigned to the mutable type '[number, string]'.

const pairConst = [1157, "Tomoe"] as const;
```

```
pairConst[0] = 1247;  
//      ~  
// Error: Cannot assign to '0' because it is a read-only property.
```

Na prática, as tuplas somente leitura são convenientes como retorno de funções. Geralmente, os valores fornecidos por funções que retornam uma tupla são sempre desestruturados imediatamente, logo, a tupla ser somente leitura não atrapalha o uso da função.

A função `firstCharAndSizeAsConst` retorna um valor `readonly [string, number]`, mas o código chamador só está interessado em recuperar os valores da tupla:

```
// Return type: readonly [string, number]  
function firstCharAndSizeAsConst(input: string) {  
    return [input[0], input.length] as const;  
}  
  
// firstChar type: string  
// size type: number  
const [firstChar, size] = firstCharAndSizeAsConst("Ching Shih");
```



Os objetos somente leitura e as asserções `as const` serão abordados com mais detalhes no Capítulo 9, “Modificadores de Tipo”.

## Resumo

Neste capítulo, você trabalhou com a declaração de arrays e a recuperação de seus membros:

- Declaração de tipos array com `[ ]`.
- Uso de parênteses para declarar arrays de funções ou tipos união.
- Como o TypeScript entende os elementos do array como o tipo do array.
- Trabalho com spreads e rests ....
- Declaração de tipos tupla para representar arrays de tamanho fixo.
- Uso de anotações de tipo ou asserções `as const` para criar tuplas.



Agora que você terminou de ler este capítulo, pratique o que aprendeu em <https://learningtypescript.com/arrays>.

*Qual é a estrutura de dados favorita de um pirata?  
Arrrrr-ays!<sup>3</sup>*

---

<sup>1</sup> N.T.: Original: *Arrays and tuples / One flexible and one fixed / Choose your adventure*

<sup>2</sup> N.T.: Com a desestruturação podemos extrair dados dos arrays e dos objetos e atribuí-los às variáveis.

<sup>3</sup> N.T.: Original: *What's a pirate's favorite data structure? Arrrrr-ays!*

## CAPÍTULO 7

# Interfaces

*Por que só usar as  
Tediosas formas de tipos internas se  
Podemos criar as nossas?<sup>1</sup>*

Mencionei no Capítulo 4, “Objetos” que, embora os apelidos para tipos de objeto { ... } sejam uma maneira de descrever as formas dos objetos, o TypeScript também inclui um recurso “interface” que muitos desenvolvedores preferem. As interfaces são outra maneira de declarar a forma de um objeto com um nome associado. Em muitos aspectos elas são semelhantes aos apelidos de tipos de objeto, mas são preferidas por suas mensagens de erro mais legíveis, desempenho mais rápido do compilador e melhor interoperabilidade com as classes.

## Apelidos de tipo versus interfaces

Vejamos um exemplo rápido da sintaxe de como um apelido de tipo de objeto descreveria um objeto com `born: number` e `name: string`:

```
type Poet = {  
    born: number;  
    name: string;  
};
```

Aqui está a sintaxe equivalente para uma interface:

```
interface Poet {  
    born: number;  
    name: string;  
}
```

As duas sintaxes são quase idênticas.



Os desenvolvedores TypeScript que preferem o símbolo de ponto e

vírgula geralmente o colocam após apelidos de tipo e não após interfaces. Essa preferência reflete a diferença entre declarar um variável com ; e declarar uma classe ou função sem.

A verificação de capacidade de atribuição e as mensagens de erro do TypeScript também funcionam para interfaces e têm quase a mesma aparência das dos tipos de objeto. As mensagens de erro de capacidade de atribuição a seguir referentes à atribuição à variável `valueLater` seriam quase iguais se `Poet` fosse uma interface ou um apelido de tipo:

```
let valueLater: Poet;

// Ok
valueLater = {
  born: 1935,
  name: 'Sara Teasdale',
};

valueLater = "Emily Dickinson";
// Error: Type 'string' is not assignable to 'Poet'.

valueLater = {
  born: true,
  // Error: Type 'boolean' is not assignable to type 'number'.
  name: 'Sappho'
};
```

No entanto, existem algumas diferenças essenciais entre interfaces e apelidos de tipo:

- Como você verá posteriormente neste capítulo, as interfaces podem se “mesclar” para ficar maiores – um recurso particularmente útil no trabalho com código de terceiros como variáveis globais internas ou pacotes npm.
- Como você verá no Capítulo 8, “Classes”, as interfaces podem ser usadas na verificação do tipo da estrutura de declarações de classe, o que os apelidos de tipo não fazem.
- Geralmente, o verificador de tipos do TypeScript é mais rápido no trabalho com interfaces: elas declaram um tipo nomeado que pode ser armazenado com mais facilidade internamente em cache, em vez de

usar uma cópia dinâmica de um novo objeto literal como fazem os apelidos de tipo.

- Já que as interfaces são consideradas objetos nomeados em vez de um apelido para um objeto literal nomeado, suas mensagens de erro têm mais probabilidades de serem legíveis em casos muito extremos.

Devido às duas últimas razões e para manter a consistência, o restante deste livro e os projetos associados por padrão usarão interfaces em vez de apelidos de formas de objetos. Geralmente recomendo usar interfaces sempre que possível (isto é, até você precisar de recursos como os tipos união dos apelidos de tipo).

## Tipos de propriedades

Pode ser difícil usar objetos do JavaScript no mundo real, o que inclui getters e setters<sup>2</sup>, propriedades que nem sempre existem ou a aceitar algum nome de propriedade arbitrário. O TypeScript fornece um conjunto de ferramentas de tipagem para nos ajudar a modelar esse contrassenso.



Já que as interfaces e os apelidos de tipo se comportam de maneira semelhante, todos os tipos de propriedades introduzidos neste capítulo também podem ser usados com apelidos de tipos de objeto.

### Propriedades opcionais

Como ocorre com os tipos de objeto, nem todas as propriedades das interfaces precisam ser obrigatórias no objeto. Você pode indicar que uma propriedade da interface é opcional incluindo um `?` antes de `:` na sua anotação de tipo.

Esta interface `Book` quer apenas uma propriedade `required` (obrigatória) e opcionalmente permite uma `optional`. Os objetos que a usarem podem fornecer `optional` ou deixá-la de fora contanto que forneçam `required`:

```
interface Book {  
    author?: string;  
    pages: number;  
};
```

```
// Ok
const ok: Book = {
    author: "Rita Dove",
    pages: 80,
};
const missing: Book = {
    pages: 80
};
// Error: Property 'author' is missing in type
// '{ pages: number; }' but required in type 'Book'.
```

A mesma advertência referente à diferença entre propriedades opcionais e propriedades cujo tipo inclua `undefined` em uma união de tipos é aplicável tanto às interfaces quanto aos tipos de objetos. O Capítulo 13, “Opções de configuração”, descreverá as configurações de rigidez do TypeScript relacionadas às propriedades opcionais.

## Propriedades somente leitura

Em certas situações, você pode querer impedir que os usuários de sua interface reatribuam as propriedades dos objetos que a estejam usando. O TypeScript permite adicionar um modificador `readonly` antes do nome de uma propriedade para indicar que uma vez definida, ela não deve ser atribuída com um valor diferente. Essas propriedades `readonly` podem ser lidas normalmente, mas não podem ser reatribuídas a nada novo.

Por exemplo, a propriedade `text` da interface `Page` a seguir retorna uma `string` quando acessada, mas causa um type error ao receber um novo valor:

```
interface Page {
    readonly text: string;
}

function read(page: Page) {
    // Ok: reading the text property doesn't attempt to modify it
    console.log(page.text);

    page.text += "!";
    // ~~~~
    // Error: Cannot assign to 'text'
    // because it is a read-only property.
```

```
}
```

É bom ressaltar que o modificador `readonly` só existe na tipagem e é aplicável apenas ao uso dessa interface. Ele não será aplicável a um objeto a menos que este seja usado em um local que o declare como sendo dessa interface.

Nesta continuação do exemplo de `exclaim`, a propriedade `text` pode ser modificada fora da função porque seu objeto pai não é usado explicitamente como `Text` até estar dentro da função. `pageIsh` pode ser usada como `Page` porque uma propriedade gravável (`writable`) é atribuível a uma propriedade `readonly` (somente leitura) (propriedades mutáveis podem ser lidas, o que é exatamente do que uma propriedade `readonly` precisa):

```
const pageIsh = {
  text: "Hello, world!",
};

// Ok: messengerIsh is an inferred object type with text, not a Page
page.text += "!";

// Ok: read takes in Page, which happens to
// be a more specific version of pageIsh's type
read(messengerIsh);
```

Declarar a variável `pageIsh` com a anotação de tipo explícita : `Page` teria indicado para o TypeScript que sua propriedade `text` é `readonly`. O tipo inferido, entretanto, não foi `readonly`.

As propriedades da interface somente leitura são uma maneira útil de assegurar que áreas de código não modifiquem inesperadamente objetos que elas não deveriam modificar. No entanto, lembre-se de que elas são uma estrutura apenas da tipagem e não existem no código de saída JavaScript compilado. Só protegem contra a modificação durante o desenvolvimento com o verificador de tipos do TypeScript.

## Funções e métodos

É muito comum em JavaScript as propriedades dos objetos serem funções. Logo, o TypeScript permite declarar as propriedades das

interfaces como tendo os tipos de função abordados anteriormente no Capítulo 5, “Funções”.

O TypeScript fornece duas maneiras de declarar propriedades de interface como funções:

- Sintaxe method: declara que uma propriedade da interface é uma função que deve ser chamada como uma propriedade do objeto, como em `member(): void`.
- Sintaxe property: declara que uma propriedade da interface é igual a uma função autocontida, como em `member: () => void`.

Esses dois tipos de declaração são análogos às duas maneiras pelas quais podemos declarar um objeto JavaScript como tendo uma função.

As propriedades `method` e `property` mostradas aqui são funções que podem ser chamadas sem parâmetros e retornam uma `string`:

```
interface HasBothFunctionTypes {  
    property: () => string;  
    method(): string;  
}  
  
const hasBoth: HasBothFunctionTypes = {  
    property: () => "",  
    method() {  
        return "";  
    }  
};  
  
hasBoth.property(); // Ok  
hasBoth.method(); // Ok
```

As duas formas podem receber o modificador opcional `?` para indicar que as propriedades não precisam ser fornecidas:

```
interface OptionalReadonlyFunctions {  
    optionalProperty?: () => string;  
    optionalMethod?(): string;  
}
```

Quase sempre as declarações de método e propriedade podem ser usadas de maneira intercambiável. As principais diferenças entre elas que abordarei neste livro são:

- Os métodos não podem ser declarados como `readonly` (somente leitura); as propriedades podem.
- A mesclagem de interfaces (abordada posteriormente neste capítulo) as trata diferentemente.
- Algumas das operações executadas com tipos abordadas no Capítulo 15, “Operações com tipos”, as tratam diferentemente.

Versões futuras do TypeScript talvez adicionem a opção de adoção de maior rigidez nas diferenças entre os métodos e as funções de propriedade.

Por enquanto, o guia de estilo geral que recomendo é:

- Use uma função de tipo método se souber que a função subjacente pode referenciar `this`, o que é mais comum para instâncias de classes (abordadas no Capítulo 8, “Classes”).
- Caso contrário use uma função de propriedade.

Não se preocupe se confundir as duas ou não entender a diferença. Raramente isso afetará seu código a não ser que você esteja usando intencionalmente o escopo `this` na forma escolhida.

## Assinaturas de chamadas

As interfaces e os tipos de objeto podem declarar *assinaturas de chamadas*, que é uma descrição da tipagem para como um valor pode ser chamado como uma função. Só valores que puderem ser chamados da maneira que a assinatura de chamada declarar serão atribuíveis à interface – isto é, uma função com parâmetros e tipo de retorno atribuíveis. Uma assinatura de chamada é semelhante a um tipo de função, mas com : (dois pontos) em vez de uma seta (`=>`).

Os tipos `FunctionAlias` e `CallSignature` a seguir descrevem os mesmos parâmetros de função e tipo de retorno:

```
type FunctionAlias = (input: string) => number;

interface CallSignature {
  (input: string): number;
}
```

```
// Type: (input: string) => number
const typedFunctionAlias: FunctionAlias = (input) => input.length; // Ok

// Type: (input: string) => number
const typedCallSignature: CallSignature = (input) => input.length; // Ok
```

As assinaturas de chamadas podem ser usadas para descrever funções que adicionalmente tenham alguma propriedade definida pelo usuário. O TypeScript reconhecerá uma propriedade adicionada a uma declaração de função como um acréscimo ao tipo da declaração dessa função.

A declaração da função `keepsTrackOfCalls` a seguir recebeu uma propriedade `count` de tipo `number`, o que a tornou atribuível à interface `FunctionWithCount`. Logo, ela pode ser atribuída ao argumento `hasCallCount` de tipo `FunctionWithCount`. A função do final do código não recebeu `count`:

```
interface FunctionWithCount {
    count: number;
    (): void;
}

let hasCallCount: FunctionWithCount;

function keepsTrackOfCalls() {
    keepsTrackOfCalls.count += 1;
    console.log(`I've been called ${keepsTrackOfCalls.count} times!`);
}

keepsTrackOfCalls.count = 0;

hasCallCount = keepsTrackOfCalls; // Ok

function doesNotHaveCount() {
    console.log("No idea!");
}

hasCallCount = doesNotHaveCount;
// Error: Property 'count' is missing in type
// '() => void' but required in type 'FunctionWithCalls'
```

## Assinaturas de índice

Alguns projetos JavaScript criam objetos destinados a armazenar valores

usando alguma chave `string` arbitrária. Para esses objetos “contêineres”, declarar uma interface com um campo para cada chave seria inviável ou impossível.

O TypeScript fornece uma sintaxe chamada *assinatura de índice* para indicar que os objetos de uma interface podem receber uma chave e retornar o tipo específico dessa chave. Ela é mais usada com chaves `string` porque as pesquisas de propriedades de objetos do JavaScript convertem chaves em `strings` implicitamente. Uma assinatura de índice se parece com uma definição de propriedade comum, mas com um tipo depois da chave e um colchete de array envolvendo-os, como em `{ [i: string]: ... }`.

Esta interface `WordCounts` foi declarada como podendo receber qualquer chave `string` com valor `number`. Os objetos desse tipo não estão restritos a receber uma chave específica – o que importa é que o valor seja de tipo `number`:

```
interface WordCounts {  
    [i: string]: number;  
}  
  
const counts: WordCounts = {};  
  
counts.apple = 0; // Ok  
counts.banana = 1; // Ok  
  
counts.cherry = false;  
// Error: Type 'boolean' is not assignable to type 'number'.
```

As assinaturas de índice são convenientes para a atribuição de valores a um objeto, mas não são totalmente type safe. Elas indicam que um objeto deve retornar um valor independentemente da propriedade que estiver sendo acessada.

A seguir, o valor de `publishDates` retorna seguramente `Frankenstein` como `Date`, mas engana o TypeScript, fazendo-o achar que `Beloved` foi definido ainda que seja `undefined`:

```
interface DatesByName {  
    [i: string]: Date;  
}
```

```

const publishDates: DatesByName = {
  Frankenstein: new Date("1 January 1818"),
};

publishDates.Frankenstein; // Type: Date
console.log(publishDates.Frankenstein.toString()); // Ok

publishDates.Beloved; // Type: Date, but runtime value of undefined!
console.log(publishDates.Beloved.toString()); // Ok in the type system, but...
// Runtime error: Cannot read property 'toString'
// of undefined (reading publishDates.Beloved)

```

Quando possível, se você quiser armazenar pares chave-valor e as chaves não forem conhecidas antecipadamente, pode ser mais seguro usar `Map`. Seu método `.get` sempre retorna um tipo com `| undefined` para indicar que a chave pode não existir. O Capítulo 9, “Modificadores de tipo”, discutirá o trabalho com classes contêiner genéricas como `Map` e `Set`.

## Combinando propriedades e assinaturas de índice

As interfaces podem incluir explicitamente propriedades nomeadas e assinaturas de índice `string` gerais, com uma única restrição: cada tipo de propriedade nomeada deve ser atribuível ao tipo geral da assinatura de índice. Podemos considerar essa combinação como se disséssemos ao TypeScript que as propriedades nomeadas fornecem um tipo mais específico e qualquer outra propriedade usaria o tipo da assinatura de índice.

Aqui, `HistoricalNovels` declara que todas as propriedades são de tipo `number`, e adicionalmente a propriedade `Oroonoko` precisa existir:

```

interface HistoricalNovels {
  Oroonoko: number;
  [i: string]: number;
}

// Ok
const novels: HistoricalNovels = {
  Outlander: 1991,
  Oroonoko: 1688,
};

```

```
const missingOroonoko: HistoricalNovels = {
  Outlander: 1991,
};
// Error: Property 'Oroonoko' is missing in type
// '{ Outlander: number; }' but required in type 'HistoricalNovels'.
```

Um truque comum da tipagem com propriedades e assinaturas de índice combinadas é o uso de um tipo literal para a propriedade nomeada mais específico que o primitivo de uma assinatura de índice. Contanto que o tipo da propriedade nomeada seja atribuível ao tipo da assinatura de índice – o que ocorre no caso de um literal e um primitivo, respectivamente – o TypeScript permitirá que isso seja feito.

A seguir, `ChapterStarts` declara que uma propriedade considerada como `preface` deve ser `0` e todas as outras propriedades terão o tipo mais geral `number`. Isso significa que qualquer objeto que usar `ChapterStarts` precisa ter uma propriedade `preface` igual a `0`:

```
interface ChapterStarts {
  preface: 0;
  [i: string]: number;
}

const correctPreface: ChapterStarts = {
  preface: 0,
  night: 1,
  shopping: 5
};

const wrongPreface: ChapterStarts = {
  preface: 1,
  // Error: Type '1' is not assignable to type '0'.
};
```

## Assinaturas de índice numéricas

Embora o JavaScript converta implicitamente chaves de pesquisa de propriedades de objetos em strings, às vezes ele só permite números como chaves de um objeto. As assinaturas de índice do TypeScript podem usar um tipo `number` em vez de `string`, mas com a mesma restrição das propriedades nomeadas de que seus tipos devem ser atribuíveis ao tipo

`string` da assinatura de índice geral.

A interface `MoreNarrowNumbers` a seguir seria permitida porque `string` é atribuível a `string | undefined`, mas `MoreNarrowStrings` não seria porque `string | undefined` não é atribuível a `string`:

```
// Ok
interface MoreNarrowNumbers {
  [i: number]: string;
  [i: string]: string | undefined;
}

// Ok
const mixesNumbersAndStrings: MoreNarrowNumbers = {
  0: '',
  key1: '',
  key2: undefined,
}

interface MoreNarrowStrings {
  [i: number]: string | undefined;
  // Error: 'number' index type 'string | undefined'
  // is not assignable to 'string' index type 'string'.
  [i: string]: string;
}
```

## Interfaces aninhadas

Assim como os tipos de objeto podem ser aninhados como propriedades de outros tipos de objeto, os tipos de interface também podem ter propriedades que sejam elas próprias tipos de interface (ou tipos de objeto).

Esta interface `Novel` contém uma propriedade `author` que precisa corresponder a um tipo de objeto inline e uma propriedade `setting` que deve atender à interface `Setting`:

```
interface Novel {
  author: {
    name: string;
  };
  setting: Setting;
}
```

```

interface Setting {
    place: string;
    year: number;
}

let myNovel: Novel;

// Ok
myNovel = {
    author: {
        name: 'Jane Austen',
    },
    setting: {
        place: 'England',
        year: 1812,
    }
};

myNovel = {
    author: {
        name: 'Emily Brontë',
    },
    setting: {
        place: 'West Yorkshire',
    },
    // Error: Property 'year' is missing in type
    // '{ place: string; }' but required in type 'Setting'.
};

```

## Extensões de interfaces

Pode ocorrer de acabarmos obtendo várias interfaces semelhantes. Uma interface poderia conter as mesmas propriedades de outra interface, com a inserção de algumas propriedades adicionais.

O TypeScript permite que uma interface *estenda* outra interface, o que a declarará como copiando todas as propriedades da outra interface. Uma interface pode ser marcada como estendendo outra interface com a inclusão da palavra-chave `extends` após seu nome (a interface “derivada”), seguida do nome da interface a ser estendida (a interface “base”). Isso indicará para o TypeScript que todos os objetos que usarem a interface

derivada também devem ter as propriedades da interface base.

No exemplo a seguir, a interface `Novella` estende `Writing` e, portanto, requer que os objetos tenham pelo menos as propriedades `pages` de `Novella` e `title` de `Writing`:

```
interface Writing {
    title: string;
}

interface Novella extends Writing {
    pages: number;
}

// Ok
let myNovella: Novella = {
    pages: 195,
    title: "Ethan Frome",
};

let missingPages: Novella = {
    // ~~~~~
    // Error: Property 'pages' is missing in type
    // '{ title: string; }' but required in type 'Novella'.
    title: "The Awakening",
}

let extraProperty: Novella = {
    // ~~~~~
    // Error: Type '{ genre: string; name: string; strategy: string; }'
    // is not assignable to type 'Novella'.
    // Object literal may only specify known properties,
    // and 'genre' does not exist in type 'Novella'.
    pages: 300,
    strategy: "baseline",
    style: "Naturalism"
};
```

As extensões de interfaces são uma maneira elegante de representar que um tipo de entidade do projeto é um superconjunto (inclui todas as propriedades) de outra entidade. Elas nos permitem evitar digitar o mesmo código repetidamente em várias interfaces para representar esse relacionamento.

## Propriedades sobrescritas

As interfaces derivadas podem *sobrescrever*, ou substituir, as propriedades de sua interface base declarando a propriedade novamente com um tipo diferente. O verificador de tipos do TypeScript impõe que uma propriedade sobrescrita seja atribuível à sua propriedade base. Ele faz isso para assegurar que as instâncias do tipo da interface derivada permaneçam atribuíveis ao tipo da interface base.

A maioria das interfaces derivadas que redeclara propriedades faz isso para tornar essas propriedades um subconjunto mais específico de uma união de tipos ou para dar às propriedades um tipo que estenda o tipo da interface base.

Por exemplo, o tipo de `WithNullableName` passa a ser apropriadamente não nulo em `WithNonNullableName`. No entanto, `WithNumericName` não é permitida como `number | string` e não é atribuível a `string | null`:

```
interface WithNullableName {
    name: string | null;
}

interface WithNonNullableName extends WithNullableName {
    name: string;
}

interface WithNumericName extends WithNullableName {
    name: number | string;
}
// Error: Interface 'WithNumericName' incorrectly
// extends interface 'WithNullableName'.
//   Types of property 'name' are incompatible.
//     Type 'string | number' is not assignable to type 'string | null'.
//       Type 'number' is not assignable to type 'string'.
```

## Extensão de múltiplas interfaces

As interfaces do TypeScript podem ser declaradas como estendendo várias outras interfaces. Qualquer número de nomes de interfaces separados por vírgulas pode ser usado após a palavra-chave `extends` depois do nome da interface derivada. Esta receberá todos os membros das interfaces base.

Aqui, `givesBothAndEither` tem três métodos: um dela mesma, um é de `GivesNumber` e o outro é de `GivesString`:

```
interface GivesNumber {
    giveNumber(): number;
}

interface GivesString {
    giveString(): string;
}

interface GivesBothAndEither extends GivesNumber, GivesString {
    giveEither(): number | string;
}

function useGivesBoth(instance: GivesBothAndEither) {
    instance.giveEither(); // Type: number | string
    instance.giveNumber(); // Type: number
    instance.giveString(); // Type: string
}
```

Ao marcar uma interface como estendendo outras interfaces, você poderá reduzir a duplicação de código e facilitar a reutilização das formas dos objetos em diferentes áreas do código.

## Mesclagem de interfaces

Um dos recursos importantes das interfaces é sua capacidade de *mesclar*-se umas com as outras. Mesclar interfaces significa que, se duas interfaces forem declaradas no mesmo escopo com o mesmo nome, elas serão reunidas em uma interface maior com esse nome e com todos os campos declarados.

Este trecho de código declara uma interface `Merged` com duas propriedades:

`fromFirst` e `fromSecond`:

```
interface Merged {
    fromFirst: string;
}

interface Merged {
    fromSecond: number;
```

```
}

// Equivalent to:
// interface Merged {
//   fromFirst: string;
//   fromSecond: number;
// }
```

A mesclagem de interfaces não é um recurso usado com muita frequência no desenvolvimento diário com o TypeScript. Eu recomendaria evitá-la quando possível, já que pode ser difícil entender um código em que uma interface é declarada em vários locais.

No entanto, a mesclagem de interfaces é particularmente útil para a ampliação de interfaces de pacotes externos ou de interfaces globais internas como `Window`. Por exemplo, com o uso das opções padrão do compilador TypeScript, declarar uma interface `Window` em um arquivo com uma propriedade `myEnvironmentVariable` torna `window.myEnvironmentVariable` disponível:

```
interface Window {
  myEnvironmentVariable: string;
}

window.myEnvironmentVariable; // Type: string
```

Abordarei as definições de tipo com mais detalhes no Capítulo 11, “Arquivos de declaração”, e as opções de tipos globais do TypeScript no Capítulo 13, “Opções de Configuração”.

## Conflitos de nomeação de propriedades

É bom ressaltar que as interfaces mescladas não podem declarar o mesmo nome de uma propriedade várias vezes com tipos diferentes. Se uma propriedade já tiver sido declarada em uma interface, uma interface mesclada posteriormente terá de usar o mesmo tipo.

Nesta interface `MergedProperties`, a propriedade `same` é permitida porque é igual nas duas declarações, mas `different` é um erro por ser de um tipo diferente:

```
interface MergedProperties {
  same: (input: boolean) => string;
```

```

    different: (input: string) => string;
}

interface MergedProperties {
    same: (input: boolean) => string; // Ok

    different: (input: number) => string;
    // Error: Subsequent property declarations must have the same type.
    // Property 'different' must be of type '(input: string) => string',
    // but here has type '(input: number) => string'.
}

```

Contudo, as interfaces mescladas podem definir um método com o mesmo nome e uma assinatura diferente. Isso criará uma sobrescrita de função para o método.

A interface `MergedMethods` a seguir cria um método `different` que tem duas sobrecargas:

```

interface MergedMethods {
    different(input: string): string;
}

interface MergedMethods {
    different(input: number): string; // Ok
}

```

## Resumo

Este capítulo introduziu como os tipos de objeto podem ser descritos pelas interfaces:

- Uso de interfaces em vez de apelidos de tipo para declarar tipos de objeto.
- Vários tipos de propriedades de interfaces: opcionais, somente leitura, de função e de método.
- Uso de assinaturas de índice para propriedades gerais de objetos.
- Reutilização de interfaces com o uso de interfaces aninhadas e herança com `extends`.
- Como interfaces com o mesmo nome podem ser mescladas.

Em seguida, veremos uma sintaxe nativa do JavaScript para definir que

múltiplos objetos tenham as mesmas propriedades: as classes.

 Agora que você terminou de ler este capítulo, pratique o que aprendeu em <https://learningtypescript.com/objects-and-interfaces>.

*Por que as interfaces são boas motoristas?  
Elas são ótimas nas conversões.<sup>3</sup>*

---

1 N.T.: Original: *Why only use the / Boring built-in type shapes when / We can make our own!*

2 N.T.: Para cada instância de variável, um método getter retorna seu valor, enquanto um método setter o define ou atualiza.

3 N.T.: Original: *Why are interfaces good drivers? They're great at merging.*

## CAPÍTULO 8

# Classes

*Alguns desenvolvedores funcionais*

*Tentam nunca usar classes*

*O que é intenso demais para mim<sup>1</sup>*

O universo do JavaScript durante a criação e o lançamento do TypeScript no início dos anos 2010 era muito diferente do atual. Recursos como arrow functions e variáveis `let/const` que posteriormente seriam padronizados no ES2015 ainda eram promessas distantes. Faltavam alguns anos para o primeiro commit<sup>2</sup> do Babel<sup>3</sup>; as ferramentas que o antecederam, como o Traceur, que convertia uma sintaxe JavaScript mais nova para a antiga, ainda não tinha sido popularmente adotado.

A divulgação e o conjunto de recursos iniciais do TypeScript foram personalizados para esse universo. Além da verificação de tipos, foi dada ênfase ao seu transpilador – com as classes como um exemplo frequente. Atualmente, o suporte a classes do TypeScript é apenas um recurso entre muitos para o suporte a todos os recursos da linguagem JavaScript. O TypeScript não encoraja nem desencoraja o uso de classes ou de algum outro padrão popular do JavaScript.

## Métodos de classes

Geralmente, o TypeScript entende métodos da mesma forma que ele entende as funções autocontidas. Os tipos dos parâmetros são `any`, a não ser que eles recebam um tipo ou um valor padrão; chamar o método requer um número de argumentos aceitável; os tipos de retorno podem ser inferidos se a função não for recursiva.

Este trecho de código define uma classe `Greeter` com um método `greet` que

recebe um único parâmetro obrigatório de tipo `string`:

```
class Greeter {  
    greet(name: string) {  
        console.log(`#${name}, do your stuff!`);  
    }  
}  
  
new Greeter().greet("Miss Frizzle"); // Ok  
  
new Greeter().greet();  
// ~~~~~  
// Error: Expected 1 arguments, but got 0.
```

Os construtores de classe são tratados como os métodos típicos das classes no que diz respeito aos seus parâmetros. O TypeScript executa a verificação de tipos para assegurar que um número de argumentos correto com os tipos certos seja fornecido às chamadas de métodos.

O construtor `Greeted` a seguir também espera que seu parâmetro `message: string` seja fornecido:

```
class Greeted {  
    constructor(message: string) {  
        console.log(`As I always say: ${message}!`);  
    }  
}  
  
new Greeted("take chances, make mistakes, get messy");  
  
new Greeted();  
// Error: Expected 1 arguments, but got 0.
```

Abordarei os construtores no contexto das subclasses posteriormente neste capítulo.

## Propriedades de classes

Para que seja feita a leitura ou a escrita em uma propriedade de uma classe no TypeScript, ela deve ser declarada explicitamente na classe. As propriedades das classes são declaradas com o uso da mesma sintaxe das interfaces: seu nome opcionalmente seguido de uma anotação de tipo.

O TypeScript não tenta deduzir quais propriedades podem existir em uma classe a partir de suas atribuições em um construtor.

Neste exemplo, `destination` pode ser atribuída e acessada em instâncias da classe `FieldTrip` porque foi declarada explicitamente como uma `string`. A atribuição de `this.nonexistent` no construtor não é permitida porque a classe não declara uma propriedade `nonexistent`:

```
class FieldTrip {  
    destination: string;  
  
    constructor(destination: string) {  
        this.destination = destination; // Ok  
        console.log(`We're going to ${this.destination}!`);  
  
        this.nonexistent = destination;  
        // ~~~~~~  
        // Error: Property 'nonexistent' does not exist on type 'FieldTrip'.  
    }  
}
```

Declarar as propriedades explicitamente permite que o TypeScript saiba com rapidez o que pode ou não existir nas instâncias das classes. Posteriormente, quando instâncias das classes estiverem sendo usadas, o TypeScript utilizará esse conhecimento para exibir uma mensagem de type error se o código tentar acessar uma propriedade de uma instância cuja existência não seja conhecida, como no caso de `trip.nonexistent` nesta continuação:

```
const trip = new FieldTrip("planetarium");  
  
trip.destination; // Ok  
  
trip.nonexistent;  
// ~~~~~~  
// Error: Property 'nonexistent' does not exist on type 'FieldTrip'.
```

## Propriedades de função

Recapitularemos alguns aspectos básicos do escopo e da sintaxe dos métodos JavaScript, já que eles podem surpreender se você não estiver acostumado. O JavaScript contém duas sintaxes para a declaração de uma

propriedade em uma classe como uma função invocável: de *método* e de *propriedade*.

Já mostrei a abordagem de método na qual são inseridos parênteses após o nome da propriedade, como em `myFunction() {}`. A abordagem de método atribui uma função ao protótipo da classe, assim todas as instâncias da classe usarão a mesma definição de função.

Esta classe `WithMethod` declara um método `myMethod` que todas as instâncias poderão referenciar:

```
class WithMethod {  
    myMethod() {}  
}  
  
new WithMethod().myMethod === new WithMethod().myMethod; // true
```

A outra sintaxe seria a de declarar uma propriedade cujo valor é uma função. Isso criará uma nova função por instância, o que pode ser útil com arrow functions `() =>` cujo escopo `this` deve sempre apontar para a instância da classe (apesar do custo em tempo e memória para a criação de uma nova função por instância).

A classe `WithProperty` a seguir contém uma única propriedade de nome `myProperty` e tipo `() => void` que será recriada para cada instância da classe:

```
class WithProperty {  
    myProperty: () => {}  
}  
  
new WithMethod().myProperty === new WithMethod().myProperty; // false
```

As propriedades de função podem receber parâmetros e tipos de retorno com o uso da mesma sintaxe dos métodos de classe e das funções autocontidas. Afinal, elas são um valor atribuído a um membro da classe e o valor por acaso é uma função.

Esta classe `WithPropertyParams` tem uma propriedade `takesParameters` de tipo `(input: string) => number`:

```
class WithPropertyParams {  
    takesParameters = (input: boolean) => input ? "Yes" : "No";  
}
```

```
const instance = new WithPropertyParameters();

instance.takesParameters(true); // Ok

instance.takesParameters(123);
// ~~~
// Error: Argument of type 'number' is not
// assignable to parameter of type 'boolean'.
```

## Verificação de inicialização

Com as configurações estritas do compilador ativadas, o TypeScript verificará se cada propriedade declarada, cujos tipos não incluem `undefined`, receberam um valor no construtor. Essa verificação de inicialização estrita é útil porque impede que o código não atribua acidentalmente um valor a uma propriedade da classe.

A classe `WithValue` a seguir não atribui um valor à sua propriedade `unused`, o que o TypeScript reconhece como um type error:

```
class WithValue {
    immediate = 0; // Ok
    later: number; // Ok (set in the constructor)
    maybeUndefined: number | undefined; // Ok (allowed to be undefined)

    unused: number;
    // Error: Property 'unused' has no initializer
    // and is not definitely assigned in the constructor.

    constructor() {
        this.later = 1;
    }
}
```

Sem a verificação de inicialização estrita, uma instância da classe poderia conseguir acessar um valor que pode ser `undefined` ainda que a tipagem não o permita.

Este exemplo seria compilado sem problemas se a verificação de inicialização estrita não ocorresse, mas o JavaScript resultante quebraria no runtime:

```
class MissingInitializer {
    property: string;
```

```
}

new MissingInitializer().property.length;
// TypeError: Cannot read property 'length' of undefined
```

O erro de um bilhão de dólares ataca novamente!

A configuração da verificação estrita de inicialização de propriedades com a opção de compilador `strictPropertyInitialization` do TypeScript será abordada no Capítulo 12, “Uso de recursos do IDE”.

## Propriedades definitivamente atribuídas

Embora a verificação de inicialização estrita seja quase sempre útil, você pode encontrar alguns casos nos quais uma propriedade da classe possa ser intencionalmente desatribuída depois do construtor. Se tiver certeza absoluta de que uma propriedade não deve ter a verificação de inicialização estrita aplicada a ela, adicione um `!` após seu nome para desativar a verificação. Isso confirmará para o TypeScript que a propriedade receberá um valor diferente de `undefined` antes de ser usada pela primeira vez.

A classe `ActivitiesQueue` a seguir pode ser reinicializada qualquer número de vezes separadamente de seu construtor, logo, sua propriedade `pending` deve ser marcada com `!:`

```
class ActivitiesQueue {
    pending!: string[]; // Ok

    initialize(pending: string[]) {
        this.pending = pending;
    }

    next() {
        return this.pending.pop();
    }
}

const activities = new ActivitiesQueue();

activities.initialize(['eat', 'sleep', 'learn'])
activities.next();
```



Geralmente, precisar desativar a verificação de inicialização estrita na propriedade de uma classe é sinal de um código que está sendo definido de uma maneira que não se adequa muito bem à verificação de tipos. Em vez de adicionar uma asserção com o operador `!` e reduzir a segurança de tipo (type safety) da propriedade, considere refatorar a classe para que ela não precise mais da asserção.

## Propriedades opcionais

Assim como as interfaces, as classes no TypeScript podem declarar uma propriedade como opcional adicionando um `?` após seu nome na declaração. As propriedades opcionais se comportam de maneira semelhante às propriedades cujos tipos são de uma união que inclua `| undefined`. A verificação de inicialização estrita não considerará relevante se elas não forem definidas explicitamente em seu construtor.

Esta classe `OptionalProperty` marca `property` como opcional, logo, é permitido que ela não seja atribuída ao construtor da classe independentemente da verificação estrita de inicialização de propriedade:

```
class MissingInitializer {  
    property?: string;  
}  
  
new MissingInitializer().property?.length; // Ok  
  
new MissingInitializer().property.length;  
// Error: Object is possibly 'undefined'.
```

## Propriedades somente leitura

Novamente, assim como as interfaces, as classes em TypeScript podem declarar uma propriedade como somente leitura adicionando a palavra-chave `readonly` antes de seu nome na declaração. A palavra-chave `readonly` só existe dentro da tipagem e é removida na compilação para JavaScript.

As propriedades declaradas como `readonly` só podem receber valores iniciais onde forem declaradas ou em um construtor. Códigos de outros locais – incluindo métodos da própria classe – só podem ler as propriedades, sem poder escrever nelas.

Neste exemplo, a propriedade `text` da classe `Quote` recebe um valor no construtor, mas os outros usos causam type errors:

```
class Quote {
    readonly text: string;

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

    emphasize() {
        this.text += "!";
        // ~~~~
        // Error: Cannot assign to 'text' because it is a read-only property.
    }
}

const quote = new Quote(
    "There is a brilliant child locked inside every student."
);

Quote.text = "Ha!";
// Error: Cannot assign to 'text' because it is a read-only property.
```



Os usuários externos de seu código, como os consumidores de algum pacote npm que você tiver publicado, podem não respeitar os modificadores `readonly` — principalmente se estiverem escrevendo JavaScript e não tiverem verificação de tipos. Se você precisar de uma proteção somente leitura realmente eficaz, considere usar campos privados marcados com `#` e/ou as propriedades de função `get()`.

Propriedades declaradas como `readonly` com um primitivo como valor inicial têm uma pequena peculiaridade em comparação com as outras propriedades: se possível elas são inferidas como tendo o tipo *literal* estreitado de seu valor, em vez do *primitivo* mais amplo. O TypeScript se sente à vontade com um estreitamento de tipo inicial mais agressivo porque sabe que o valor não será alterado posteriormente; isso é semelhante às variáveis `const` assumindo tipos mais restritos do que as variáveis `let`.

No exemplo a seguir, inicialmente as duas propriedades da classe são

declaradas como uma string literal, logo, para ampliar uma delas para `string`, uma anotação de tipo é necessária:

```
class RandomQuote {
    readonly explicit: string = "Home is the nicest word there is.";
    readonly implicit = "Home is the nicest word there is.";

    constructor() {
        if (Math.random () > 0.5) {
            this.explicit = "We start learning the minute we're born." // Ok;

            this.implicit = "We start learning the minute we're born.";
            // Error: Type '"We start learning the minute we're born."' is
            // not assignable to type '"Home is the nicest word there is."'.
        }
    }
}

const quote = new RandomQuote();

quote.explicit; // Type: string
quote.implicit; // Type: "Home is the nicest word there is."
```

Ampliar o tipo de uma propriedade explicitamente não costuma ser necessário. Mesmo assim, às vezes pode ser útil no caso de uma lógica condicional nos construtores como a de `RandomQuote`.

## Classes como tipos

As classes são relativamente únicas na tipagem já que uma declaração de classe cria tanto um valor de runtime – a própria classe – como um tipo que pode ser usado em anotações de tipo.

O nome desta classe `Teacher` é usado na anotação de tipo de uma variável `teacher`, informando ao TypeScript que ela só deve receber valores que sejam atribuíveis à classe `Teacher` – como instâncias da própria classe `Teacher`:

```
class Teacher {
    sayHello() {
        console.log("Take chances, make mistakes, get messy!");
    }
}
```

```

}

let teacher: Teacher;

teacher = new Teacher(); // Ok

teacher = "Wahoo!";
// Error: Type 'string' is not assignable to type 'Teacher'.

```

O interessante é que o TypeScript considerará qualquer tipo de objeto que inclua as mesmas propriedades de uma classe como atribuível à classe. Isso ocorre porque para a tipagem estrutural do TypeScript só importam as formas dos objetos e não como eles são declarados.

Aqui, `withSchoolBus` recebe um parâmetro de tipo `SchoolBus`. Isso pode ser atendido por qualquer objeto que tenha uma propriedade `getAbilities` de tipo `() => string[]`, como uma instância da classe `SchoolBus`:

```

class SchoolBus {
    getAbilities() {
        return ["magic", "shapeshifting"];
    }
}

function withSchoolBus(bus: SchoolBus) {
    console.log(bus.getAbilities());
}

withSchoolBus(new SchoolBus()); // Ok

// Ok
withSchoolBus({
    getAbilities: () => ["transmogrification"],
});

withSchoolBus({
    getAbilities: () => 123,
    //           ~~~
    // Error: Type 'number' is not assignable to type 'string[]'.
});

```

 Na maioria dos códigos do mundo real, os desenvolvedores não passam valores de objeto em locais que demandem tipos de classe.

Esse comportamento de verificação estrutural pode parecer inesperado, mas não ocorre com muita frequência.

## Classes e Interfaces

No Capítulo 7, “Interfaces”, mostrei como as interfaces permitem que os desenvolvedores TypeScript definam expectativas para as formas dos objetos no código. O TypeScript permite que uma classe declare suas instâncias como usando uma interface com a inclusão da palavra-chave `implements` após o nome da classe, seguida do nome de uma interface. Isso indica para o TypeScript que as instâncias da classe devem ser atribuíveis a cada uma dessas interfaces. Qualquer incompatibilidade seria considerada um type error pelo verificador de tipos.

Neste exemplo, a classe `Student` implementa corretamente a interface `Learner` incluindo sua propriedade `name` e seu método `study`, mas `Slacker` não tem `study` e, portanto, resulta em um type error:

```
interface Learner {
    name: string;
    study(hours: number): void;
}

class Student implements Learner {
    name: string;

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

    study(hours: number) {
        for (let i = 0; i < hours; i+= 1) {
            console.log("...studying...");
        }
    }
}

class Slacker implements Learner {
    // ~~~~~
    // Error: Class 'Slacker' incorrectly implements interface 'Learner'.
    // Property 'study' is missing in type 'Slacker'
```

```
// but required in type 'Learner'.
name = "Rocky";
}
```



Interfaces destinadas a serem implementadas por classes são um motivo comum para o uso da sintaxe de método na declaração da propriedade de uma interface como uma função – como utilizado pela interface `Learner`.

Marcar uma classe como implementando uma interface não altera nada na maneira como a classe é usada. Se a classe for compatível com a interface, o verificador de tipos do TypeScript permitirá que suas instâncias sejam usadas em locais em que uma instância da interface seria requerida. O TypeScript não inferirá nem mesmo os tipos dos métodos ou propriedades da classe a partir da interface: se tivéssemos adicionado um método `study(hours) {}` ao exemplo de `Slacker`, o TypeScript consideraria o parâmetro `hours` um `any` implícito, a não ser que fornecêssemos uma anotação de tipo para ele.

Esta versão da classe `Student` causa type errors de `any` implícito porque ela não fornece anotações de tipo em suas propriedades:

```
class Student implements Learner {
    name;
    // Error: Member 'name' implicitly has an 'any' type.

    study(hours) {
        // Error: Parameter 'hours' implicitly has an 'any' type.
    }
}
```

A implementação de uma interface é puramente uma verificação de segurança. Ela não copia nenhuma propriedade da interface na definição da classe para você. Implementar uma interface indica nossa intenção para o verificador de tipos e gera type errors na definição da classe em vez de posteriormente, quando instâncias da classe forem usadas. É semelhante em termos de finalidade a adicionar uma anotação de tipo a uma variável ainda que ela tenha um valor inicial.

## Implementação de múltiplas interfaces

As classes no TypeScript podem ser declaradas como implementando múltiplas interfaces. A lista de interfaces implementadas para uma classe pode ter qualquer número de nomes de interfaces com vírgulas separando-os.

Neste exemplo, as duas classes precisam ter pelo menos uma propriedade `grades` para implementar `Graded` e uma propriedade `report` para implementar `Reporter`. A classe `Empty` tem dois type errors por não implementar as duas interfaces apropriadamente:

```
interface Graded {
    grades: number[];
}

interface Reporter {
    report: () => string;
}

class ReportCard implements Graded, Reporter {
    grades: number[];

    constructor(grades: number[]) {
        this.grades = grades;
    }

    report() {
        return this.grades.join(", ");
    }
}

class Empty implements Graded, Reporter { }
// ~~~~~
// Error: Class 'Empty' incorrectly implements interface 'Graded'.
//   Property 'grades' is missing in type 'Empty'
//   but required in type 'Graded'.
// ~~~~~
// Error: Class 'Empty' incorrectly implements interface 'Reporter'.
//   Property 'report' is missing in type 'Empty'
//   but required in type 'Reporter'.
```

Na prática, podem existir algumas interfaces cujas definições tornem impossível uma classe implementar mais de uma interface. Tentar declarar

uma classe como implementando duas interfaces conflitantes resultará em pelo menos um type error na classe.

As interfaces `AgeIsANumber` e `AgeIsNotANumber` a seguir declaram tipos muito diferentes para uma propriedade `age`. Tanto a classe `AsNumber` quanto a classe `NotAsNumber` não as implementam apropriadamente:

```
interface AgeIsANumber {
    age: number;
}

interface AgeIsNotANumber {
    age: () => string;
}

class AsNumber implements AgeIsANumber, AgeIsNotANumber {
    age = 0;
    // ~~~
    // Error: Property 'age' in type 'AsNumber' is not assignable
    // to the same property in base type 'AgeIsNotANumber'.
    // Type 'number' is not assignable to type '() => string'.
}
class NotAsNumber implements AgeIsANumber, AgeIsNotANumber {
    age() { return ""; }
    // ~~~
    // Error: Property 'age' in type 'NotAsNumber' is not assignable
    // to the same property in base type 'AgeIsANumber'.
    // Type '() => string' is not assignable to type 'number'.
}
```

Casos em que duas interfaces descrevem formas de objetos muito diferentes geralmente indicam que não devemos tentar implementá-las com a mesma classe.

## Extensão de uma classe

O TypeScript adiciona a verificação de tipos ao conceito JavaScript de uma classe estendendo, ou criando subclasses, de outra classe. Para começar, todos os métodos ou propriedades declaradas em uma classe base estarão disponíveis na subclasse, também conhecida como classe derivada.

No exemplo a seguir, `Teacher` declara um método `teach` que pode ser usado por instâncias da subclasse `StudentTeacher`:

```
class Teacher {  
    teach() {  
        console.log("The surest test of discipline is its absence.");  
    }  
}  
  
class StudentTeacher extends Teacher {  
    learn() {  
        console.log("I cannot afford the luxury of a closed mind.");  
    }  
}  
  
const teacher = new StudentTeacher();  
teacher.teach(); // Ok (defined on base)  
teacher.learn(); // Ok (defined on subclass)  
  
teacher.other();  
// ~~~~~  
// Error: Property 'other' does not exist on type 'StudentTeacher'.
```

## Capacidade de atribuição na extensão

As subclasses herdam propriedades de sua classe base de maneira semelhante a como as interfaces derivadas estendem as interfaces base. As instâncias das subclasses têm todas as propriedades de sua classe base e, portanto, podem ser usadas onde quer que uma instância da classe base seja necessária. Se uma classe base não tiver todas as propriedades que uma subclasse tem, ela não poderá ser usada quando a subclasse mais específica for necessária.

As instâncias da classe `Lesson` a seguir não podem ser usadas onde instâncias de sua classe derivada, `OnlineLesson`, são requeridas, mas as instâncias da classe derivada podem ser usadas para atender à classe base ou à subclasse:

```
class Lesson {  
    subject: string;  
  
    constructor(subject: string) {
```

```

        this.subject = subject;
    }
}

class OnlineLesson extends Lesson {
    url: string;

    constructor(subject: string, url: string) {
        super(subject);
        this.url = url;
    }
}

let lesson: Lesson;
lesson = new Lesson("coding"); // Ok
lesson = new OnlineLesson("coding", "oreilly.com"); // Ok

let online: OnlineLesson;
online = new OnlineLesson("coding", "oreilly.com"); // Ok

online = new Lesson("coding");
// Error: Property 'url' is missing in type
// 'Lesson' but required in type 'OnlineLesson'.

```

De acordo com a tipagem estrutural do TypeScript, se todas as propriedades de uma subclasse já existirem em sua classe base com o mesmo tipo, instâncias da classe base poderão ser usadas no lugar da subclasse.

Neste exemplo, `LabeledPastGrades` adiciona apenas uma propriedade opcional a `PastGrades`, logo, as instâncias da classe base podem ser usadas no lugar da subclasse:

```

class PastGrades {
    grades: number[] = [];
}

class LabeledPastGrades extends PastGrades {
    label?: string;
}

let subClass: LabeledPastGrades;

```

```
subClass = new LabeledPastGrades(); // Ok
subClass = new PastGrades(); // Ok
```



Na maioria dos códigos do mundo real, geralmente as subclasses adicionam novas informações de tipo além das de sua classe base. Esse comportamento de verificação estrutural pode parecer inesperado, mas não ocorre com muita frequência.

## Construtores sobreescritos

Assim como no JavaScript vanilla, no TypeScript as subclasses não precisam definir seu próprio construtor. Subclasses sem um construtor próprio usam implicitamente o construtor de sua classe base.

Em JavaScript, se uma subclasse declarar seu próprio construtor, ela deve chamar o construtor de sua classe base com a palavra-chave `super`. Os construtores das subclasses podem declarar qualquer parâmetro, independentemente do que sua classe base demandar. O verificador de tipos do TypeScript assegurará que a chamada ao construtor da classe base use os parâmetros corretos.

Neste exemplo, o construtor de `PassingAnnouncer` chama corretamente o construtor da classe base com um argumento `number`, enquanto `FailingAnnouncer` gera um type error por não fazer essa chamada:

```
class GradeAnnouncer {
    message: string;

    constructor(grade: number) {
        this.message = grade >= 65 ? "Maybe next time..." : "You pass!";
    }
}

class PassingAnnouncer extends GradeAnnouncer {
    constructor() {
        super(100);
    }
}

class FailingAnnouncer extends GradeAnnouncer {
    constructor() { }
// ~~~~~
```

```
// Error: Constructors for subclasses must contain a 'super' call.  
}
```

De acordo com as regras do JavaScript, o construtor de uma subclasse deve chamar o construtor da classe base antes de acessar `this` ou `super`. O TypeScript relatará um type error se detectar `this` ou `super` sendo acessado antes de `super()`.

A classe `ContinuedGradesTally` a seguir referencia erroneamente `this.grades` em seu construtor antes de chamar `super()`:

```
class GradesTally {  
    grades: number[] = [];  
  
    addGrades(...grades: number[]) {  
        this.grades.push(...grades);  
        return this.grades.length;  
    }  
}  
  
class ContinuedGradesTally extends GradesTally {  
    constructor(previousGrades: number[]) {  
        this.grades = [...previousGrades];  
        // Error: 'super' must be called before accessing  
        // 'this' in the constructor of a subclass.  
  
        super();  
  
        console.log("Starting with length", this.grades.length); // ok  
    }  
}
```

## Métodos sobrescritos

As subclasses podem redeclarar novos métodos com os mesmos nomes dos métodos da classe base, contanto que o método da subclasse seja atribuível ao método da classe base. Lembre-se, já que as subclasses podem ser usadas onde quer que a classe original seja usada, os tipos dos novos métodos devem poder ser usados em substituição aos dos métodos originais.

Neste exemplo, o método `countGrades` de `FailureCounter` é permitido porque seu primeiro parâmetro e o tipo de retorno são iguais aos do método

`countGrades` da classe base `GradeCounter`. O método `countGrades` de `AnyFailureChecker` causa um type error por ter o tipo de retorno errado:

```
class GradeCounter {
    countGrades(grades: string[], letter: string) {
        return grades.filter(grade => grade === letter).length;
    }
}

class FailureCounter extends GradeCounter {
    countGrades(grades: string[]) {
        return super.countGrades(grades, "F");
    }
}

class AnyFailureChecker extends GradeCounter {
    countGrades(grades: string[]) {
        // Property 'countGrades' in type 'AnyFailureChecker' is not
        // assignable to the same property in base type 'GradeCounter'.
        // Type '(grades: string[]) => boolean' is not assignable
        // to type '(grades: string[], letter: string) => number'.
        //     Type 'boolean' is not assignable to type 'number'.
        return super.countGrades(grades, "F") !== 0;
    }
}

const counter: GradeCounter = new AnyFailureChecker();

// Expected type: number
// Actual type: boolean
const count = counter.countGrades(["A", "C", "F"]);
```

## Propriedades sobrescritas

As subclasses também podem redeclarar explicitamente propriedades de sua classe base com o mesmo nome, contanto que o novo tipo seja atribuível ao tipo da classe base. Assim como ocorre com os métodos sobrescritos, as subclasses devem ser estruturalmente compatíveis com as classes base.

A maioria das subclasses que redeclara propriedades o faz para torná-las um subconjunto mais específico de uma união de tipos ou para convertê-

las para um tipo que estenda o tipo da propriedade da classe base.

No exemplo a seguir, a classe base `Assignment` declara a propriedade `grade` como de tipo `number | undefined`, enquanto a subclasse `GradedAssignment` a declara como um tipo `number` que sempre deve existir:

```
class Assignment {
    grade?: number;
}

class GradedAssignment extends Assignment {
    grade: number;

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

Não é possível expandir o conjunto de valores permitido para o tipo de união de uma propriedade, porque se isso fosse feito a propriedade da subclasse não seria mais atribuível ao tipo da propriedade da classe base.

Neste exemplo, a propriedade `value` de `VagueGrade` tenta adicionar `| string` ao tipo `number` da classe base `NumericGrade`, causando um type error:

```
class NumericGrade {
    value = 0;
}

class VagueGrade extends NumericGrade {
    value = Math.random() > 0.5 ? 1 : "...";
    // Error: Property 'value' in type 'NumberOrString' is not
    // assignable to the same property in base type 'JustNumber'.
    //   Type 'string | number' is not assignable to type 'number'.
    //     Type 'string' is not assignable to type 'number'.
}
const instance: NumericGrade = new VagueGrade();

// Expected type: number
// Actual type: number | string
instance.value;
```

## Classes abstratas

Às vezes pode ser útil criar uma classe base que não declare a implementação de alguns métodos e espere que uma subclasse os forneça. Podemos marcar uma classe como abstrata adicionando a palavra-chave `abstract` do TypeScript na frente do nome da classe e de qualquer método que deva ser abstrato. Essas declarações de métodos abstratos não fornecem um corpo na classe base abstrata; em vez disso, eles são declarados da mesma forma que uma interface seria.

Neste exemplo, a classe `School` e seu método `getStudentTypes` são marcados como `abstract`. Logo, o que se espera é que suas subclasses – `Preschool` e `Absence` – implementem `getStudentTypes`:

```
abstract class School {
    readonly name: string;

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

    abstract getStudentTypes(): string[];
}

class Preschool extends School {
    getStudentTypes() {
        return ["preschooler"];
    }
}

class Absence extends School { }
// ~~~~~
// Error: Nonabstract class 'Absence' does not implement
// inherited abstract member 'getStudentTypes' from class 'School'.
```

Uma classe abstrata não pode ser instanciada diretamente, já que ela não tem definições de alguns métodos que sua implementação presume que existam. Só classes não abstratas (“concretas”) podem ser instanciadas.

Continuando com o exemplo de `School`, tentar chamar `new School` resultaria em um type error do TypeScript:

```
let school: School;

school = new Preschool("Sunnyside Daycare"); // Ok
```

```
school = new School("somewhere else");
// Error: Cannot create an instance of an abstract class.
```

Geralmente, as classes abstratas são usadas em frameworks nos quais é esperado que os consumidores forneçam os detalhes de uma classe. A classe pode ser usada como uma anotação de tipo para indicar que os valores devem usá-la – como no exemplo anterior de `school: School` – mas a criação de novas instâncias deve ser feita com subclasses.

## Visibilidade dos membros

O JavaScript inclui a possibilidade de iniciarmos o nome do membro de uma classe com `#` para marcá-lo como membro de classe “privado”. Os membros de classe privados só podem ser acessados pelas instâncias desta classe. Os runtimes do JavaScript impõem essa privacidade lançando um erro se uma área de código fora da classe tentar acessar a propriedade ou o método privado.

O suporte a classes do TypeScript precede a privacidade real do JavaScript que usa `#`, e embora o TypeScript suporte membros de classe privados, ele também permite um conjunto de definições de privacidade um pouco mais nuançado em classes e propriedades que só existe na tipagem. As visibilidades de membros do TypeScript são definidas com a inclusão de uma das palavras-chave a seguir antes do nome na declaração de um membro da classe:

`public` (*padrão*)

Pode ser acessado por qualquer pessoa, em qualquer local.

`protected`

Só pode ser acessado pela classe e suas subclasses.

`private`

Só pode ser acessado pela classe.

Essas palavras-chave só existem dentro da tipagem. Elas são removidas junto com todas as outras sintaxes da tipagem quando o código é compilado para JavaScript.

Aqui, `Base` declara dois membros `public`, um `protected`, um `private` e um com privacidade real com o uso de `#truePrivate`. `Subclass` pode acessar os membros `public` e `protected`, mas não o `private` ou `#truePrivate`:

```
class Base {
    isPublicImplicit = 0;
    public isPublicExplicit = 1;
    protected isProtected = 2;
    private isPrivate = 3;
    #truePrivate = 4;
}

class Subclass extends Base {
    examples() {
        this.isPublicImplicit; // Ok
        this.isPublicExplicit; // Ok
        this.isProtected; // Ok

        this.isPrivate;
        // Error: Property 'isPrivate' is private
        // and only accessible within class 'Base'.

        this.#truePrivate;
        // Property '#truePrivate' is not accessible outside
        // class 'Base' because it has a private identifier.
    }
}

new Subclass().isPublicImplicit; // Ok
new Subclass().isPublicExplicit; // Ok

new Subclass().isProtected;
// ~~~~~
// Error: Property 'isProtected' is protected
// and only accessible within class 'Base' and its subclasses.

new Subclass().isPrivate;
// ~~~~~
// Error: Property 'isPrivate' is private
// and only accessible within class 'Base'.
```

A principal diferença entre as visibilidades de membros do TypeScript e as declarações de privacidade real do JavaScript é que as do TypeScript só

existem na tipagem, enquanto as do JavaScript também existem no runtime. Um membro de classe do TypeScript declarado como `protected` ou `private` será compilado para o código JavaScript como se tivesse sido declarado de maneira explícita ou implícita como `public`. Assim como ocorre com as interfaces e as anotações de tipo, as palavras-chave de visibilidade são removidas quando é gerada a saída JavaScript. Só campos declarados como privados com o uso de `#` são realmente privados no runtime JavaScript.

Os modificadores de visibilidade podem ser marcados como `readonly`. Para declarar um membro tanto como `readonly` quanto com uma visibilidade explícita, a visibilidade vem antes.

Esta classe `TwoKeywords` declara sua propriedade `name` tanto como `private` quanto como `readonly`:

```
class TwoKeywords {
    private readonly name: string;

    constructor() {
        this.name = "Anne Sullivan"; // Ok
    }

    log() {
        console.log(this.name); // Ok
    }
}

const two = new TwoKeywords();

two.name = "Savitribai Phule";
// ~~~~
// Error: Property 'name' is private and
// only accessible within class 'TwoKeywords'.
// ~~~~
// Error: Cannot assign to 'name'
// because it is a read-only property.
```

É bom ressaltar que não é permitido combinar a palavra-chave antiga de visibilidade de membros do TypeScript com os novos campos privados `#` do JavaScript. Os campos privados são sempre privados por padrão, logo,

não há necessidade de marcá-los adicionalmente com a palavra-chave `private`.

## Modificadores de campo static

O JavaScript permite declarar membros na própria classe – em vez de em suas instâncias – com o uso da palavra-chave `static`. O TypeScript suporta o uso da palavra-chave `static` individualmente e/ou com `readonly` e/ou também com uma das palavras-chave de visibilidade. Quando combinadas, a palavra-chave de visibilidade vem antes, depois vem `static` e, então, `readonly`.

A classe `HasStatic` a seguir combina as três palavras-chave para tornar suas propriedades estáticas `prompt` e `answer` tanto `readonly` quanto `protected`:

```
class Question {
    protected static readonly answer: "bash";
    protected static readonly prompt =
        "What's an ogre's favorite programming language?";

    guess(getAnswer: (prompt: string) => string) {
        const answer = getAnswer(Question.prompt);

        // Ok
        if (answer === Question.answer) {
            console.log("You got it!");
        } else {
            console.log("Try again...")
        }
    }

    Question.answer;
    // ~~~~~
    // Error: Property 'answer' is protected and only
    // accessible within class 'HasStatic' and its subclasses.
```

O uso de modificadores de indicação de somente leitura e/ou de visibilidade para campos de classe estáticos é útil para evitarmos que esses campos sejam acessados ou modificados fora de sua classe.

## Resumo

Este capítulo introduziu muitos recursos e sintaxes da tipagem relacionados às classes:

- Declaração e uso de métodos e propriedades de classes.
- Marcação de propriedades como `readonly` e/ou opcionais.
- Uso de nomes de classes como tipos em anotações de tipo.
- Implementação de interfaces para a imposição de formas de instâncias de classes.
- Extensão das classes, junto com as regras de capacidade de atribuição e de sobrescrita para as subclasses.
- Marcação de classes e métodos como abstratos.
- Inclusão de modificadores da tipagem em campos de classe.



Agora que você terminou de ler este capítulo, pratique o que aprendeu em <https://learningtypescript.com/classes>.

*Por que os desenvolvedores da programação orientada a objetos sempre usam ternos?*

*Porque eles têm classe.<sup>4</sup>*

---

<sup>1</sup> N.T.: Original: *Some functional devs / Try to never use classes / Too intense for me*

<sup>2</sup> N.T.: Um commit adiciona as alterações mais recentes do código-fonte.

<sup>3</sup> N.T.: O Babel é um transpilador que permite, ao desenvolvedor, utilizar novas funcionalidades da linguagem, mesmo que a runtime não suporte, pois irá compilar para uma versão compatível mais antiga.

<sup>4</sup> N.T.: Original: *Why do object-oriented programming developers always wear suits? Because they've got class.*

# CAPÍTULO 9

# Modificadores de tipo

*Tipos de tipos partindo de tipos.  
“São tartarugas até lá embaixo”<sup>1</sup>,  
Anders gosta de dizer.<sup>2</sup>*

A esta altura você já leu tudo sobre como a tipagem do TypeScript funciona com as estruturas existentes do JavaScript como os arrays, as classes e os objetos. Neste capítulo e no Capítulo 10, “Genéricos”, eu me aprofundarei um pouco mais na tipagem propriamente dita e mostrarei recursos específicos para a criação de tipos mais precisos assim como de tipos baseados em outros tipos.

## Tipos universais

Mencionei o conceito de um *tipo vazio* no Capítulo 4, “Objetos”, para descrever um tipo que não pode ter valores possíveis e não pode ser alcançado. Deduz-se que o oposto também poderia existir na teoria dos tipos. E existe!

Um *top type*, ou tipo universal, é um tipo que pode representar qualquer valor possível em um sistema. Valores de todos os outros tipos podem ser fornecidos para um local cujo tipo seja universal. Em outras palavras, todos os tipos são atribuíveis a um tipo universal.

### **any, novamente**

O tipo `any` pode agir como um tipo universal, já que qualquer tipo pode ser fornecido para um local de tipo `any`. Geralmente, `any` é usado quando um local em um código pode aceitar dados de qualquer tipo, como os parâmetros de `console.log`:

```
let anyValue: any;  
anyValue = "Lucille Ball"; // Ok  
anyValue = 123; // Ok  
  
console.log(anyValue); // Ok
```

O problema de `any` é que ele solicita explicitamente ao TypeScript para não executar a verificação de tipos na capacidade de atribuição ou nos valores das propriedades. Essa falta de segurança será útil se você quiser ignorar o verificador de tipos do TypeScript, mas a desativação da verificação de tipos reduzirá a utilidade do TypeScript para esse valor.

Por exemplo, a chamada a `name.toUpperCase()` a seguir certamente será malsucedida, mas já que `name` é declarada como `any`, o TypeScript não emitirá um alerta de tipo:

```
function greetComedian(name: any) {  
    // No type error...  
    console.log(`Announcing ${name.toUpperCase()}!`);  
}  
  
greetComedian({ name: "Bea Arthur" });  
// Runtime error: name.toUpperCase is not a function
```

Se você quiser indicar que um valor pode ser qualquer coisa, o tipo `unknown` é muito mais seguro.

## unknown

O tipo `unknown` do TypeScript é seu verdadeiro tipo universal. `unknown` é semelhante a `any` já que todos os objetos podem ser passados para locais de tipo `unknown`. A diferença no caso de `unknown` é que o TypeScript é muito mais restritivo com valores de tipo `unknown`:

- O TypeScript não permite o acesso direto a propriedades com valores de tipo `unknown`.
- `unknown` não é atribuível a tipos que não sejam um tipo universal (`any` ou `unknown`).

Tentar acessar uma propriedade que tenha um valor de tipo `unknown`, como no trecho de código a seguir, fará o TypeScript relatar um type error:

```
function greetComedian(name: unknown) {
```

```

        console.log(`Announcing ${name.toUpperCase()}!`);
        // ~~~~~
        // Error: Object is of type 'unknown'.
    }

```

A única maneira de o TypeScript permitir que um código acesse propriedades com um nome de tipo `unknown` será se o tipo do valor for estreitado, como com o uso de `instanceof` ou `typeof`, ou com uma asserção de tipo.

Este trecho de código usa `typeof` para estreitar `name` de `unknown` para `string`:

```

function greetComedianSafety(name: unknown) {
    if (typeof value === "string") {
        console.log(`Announcing ${name.toUpperCase()}!`); // Ok
    } else {
        console.log("Well, I'm off.");
    }
}

greetComedianSafety("Betty White"); // Logs: 4
greetComedianSafety([]); // Does not log

```

Essas duas restrições tornam `unknown` um tipo mais seguro de usar do que `any`. É recomendável que você use `unknown` em vez de `any` quando possível.

## Predicados de tipo

Mostrei anteriormente como estruturas do JavaScript, como `instanceof` e `typeof`, podem ser usadas para estreitar tipos. Isso funciona muito bem no uso direto desse conjunto limitado de verificações, mas o benefício será perdido se você encapsular a lógica em uma função.

Por exemplo, a função `isNumberOrString` a seguir recebe um valor e retorna um booleano indicando se o valor é de tipo `number` ou `string`. Nós, como humanos, conseguimos inferir que o valor existente na instrução `if` deve, portanto, ser de um desses dois tipos já que `isNumberOrString(value)` retornou `true`, mas o TypeScript não consegue. Tudo o que ele sabe é que `isNumberOrString` retorna um booleano – e não que isso se destina a estreitar o tipo de um argumento:

```
function isNumberOrString(value: unknown) {
```

```

        return ['number', 'string'].includes(typeof value);
    }

function logValueIfExists(value: number | string | null | undefined) {
    if (isNumberOrString(value)) {
        // Type of value: number | string | null | undefined
        value.toString();
        // Error: Object is possibly undefined.
    } else {
        console.log("Value does not exist:", value);
    }
}

```

O TypeScript tem uma sintaxe especial para funções que retornam um booleano que se destina a indicar se um argumento é de um tipo específico. Ela chama-se *predicado de tipo* e às vezes também é chamada de “type guard definido pelo usuário”: você, o desenvolvedor, estará criando seu próprio type guard semelhante a `instanceof` ou `typeof`. Normalmente, os predicados de tipo são usados para indicar se um argumento passado como parâmetro é de um tipo mais específico do que o do parâmetro.

Os tipos de retorno de predicados de tipo podem ser declarados com o nome de um parâmetro, a palavra-chave `is` e algum tipo:

```
function typePredicate(input: WideType): input is NarrowType;
```

Podemos alterar a função auxiliar do exemplo anterior para que tenha um tipo de retorno explícito que declare explicitamente `value is number | string`. O TypeScript poderá então inferir que blocos de código somente alcançáveis se `value is number | string for true` devem ter um valor (`value`) de tipo `number | string`. Além disso, blocos de código só alcançáveis se `value is number | string for false` devem ter um valor de tipo `null | undefined`:

```

function isNumberOrString(value: unknown): value is number | string {
    return ['number', 'string'].includes(typeof value);
}

function logValueIfExists(value: number | string | null | undefined) {
    if (isNumberOrString(value)) {
        // Type of value: number | string
        value.toString(); // Ok
    } else {

```

```

    // Type of value: null | undefined
    console.log("value does not exist:", value);
}
}

```

Podemos considerar um predicado de tipo não só como retornando um booleano, mas também como uma indicação de que o argumento é do tipo mais específico.

Geralmente os predicados de tipo são usados para verificarmos se um objeto que sabidamente é uma instância de uma interface é instância de uma interface mais específica.

Aqui, a interface `StandupComedian` contém informações adicionais além das de `Comedian`. O type guard `isStandupComedian` pode ser usado para verificarmos se um `Comedian` (comediante) é especificamente um `StandupComedian` (comediante de stand up):

```

interface Comedian {
    funny: boolean;
}

interface StandupComedian extends Comedian {
    routine: string;
}

function isStandupComedian(value: Comedian): value is StandupComedian {
    return 'routine' in value;
}

function workWithComedian(value: Comedian) {
    if (isStandupComedian(value)) {
        // Type of value: StandupComedian
        console.log(value.routine); // Ok
    }

    // Type of value: Comedian
    console.log(value.routine);
    // ~~~~~
    // Error: Property 'routine' does not exist on type 'Comedian'.
}

```

Cuidado: já que os predicados de tipo também estreitam tipos em caso de

resultados booleanos falsos, você pode obter resultados inesperados se um predicado de tipo não verificar apenas o tipo de sua entrada.

O predicado de tipo `isLongString` retorna `false` se seu parâmetro `input` for `undefined` ou uma `string` com tamanho menor do que 7. Como resultado, a instrução `else` (seu caso falso) é estreitada para considerar `text` como de tipo `undefined`:

```
function isLongString(input: string | undefined): input is string {
    return !(input && input.length >= 7);
}

function workWithText(text: string | undefined) {
    if (isLongString(text)) {
        // Type of text: string
        console.log("Long text:", text.length);
    } else {
        // Type of text: undefined
        console.log("Short text:", text?.length);
        //
        // ~~~~~
        // Error: Property 'length' does not exist on type 'never'.
    }
}
```

É fácil usar incorretamente predicados de tipo que fazem mais do que apenas verificar o tipo de uma propriedade ou valor. Geralmente recomendo evitá-los quando possível. Predicados de tipo mais simples são suficientes na maioria dos casos.

## Operadores de tipos

Nem todos os tipos podem ser representados apenas com o uso de uma palavra-chave ou de um nome de um tipo existente. Poderíamos ter de criar um novo tipo que combinasse ambos, executando alguma transformação nas propriedades de um tipo que já existe.

### `keyof`

As propriedades dos objetos JavaScript podem ser recuperadas com o uso de valores dinâmicos, que normalmente (mas não necessariamente) são de tipo `string`. Representar essas chaves na tipagem pode ser complicado.

O uso de um primitivo geral como `string` permitiria chaves inválidas para o valor do contêiner.

É por isso que com o uso de definições de configuração mais estritas – abordadas no Capítulo 13, “Opções de configuração” – o TypeScript relataria um erro em `ratings[key]`, como veremos no próximo exemplo. O tipo `string` permite valores não permitidos como propriedades na interface `Ratings`, e `Ratings` não declara uma assinatura de índice para permitir qualquer chave `string`:

```
interface Ratings {
    audience: number;
    critics: number;
}

function getRating(ratings: Ratings, key: string): number {
    return ratings[key];
    // ~~~~~
    // Error: Element implicitly has an 'any' type because expression
    // of type 'string' can't be used to index type 'Ratings'.
    // No index signature with a parameter of
    // type 'string' was found on type 'Ratings'.
}

const ratings: Ratings = { audience: 66, critic: 84 };

getRating(ratings, 'audience'); // Ok

getRating(ratings, 'not valid'); // Ok, but shouldn't be

Outra opção seria usar uma união de tipos com literais para as chaves permitidas. Isso seria mais preciso ao fazer a apropriada restrição apenas às chaves existentes no valor do contêiner:

function getCountLiteral(ratings: Ratings, key: 'audience' | 'critic'): number {
    return ratings[key]; // Ok
}

const ratings: Ratings = { audience: 66, critic: 84 };

getCountLiteral(ratings, 'audience'); // Ok

getCountLiteral(ratings, 'not valid');
```

```
// ~~~~~
// Error: Argument of type '"not valid"' is not
// assignable to parameter of type '"audience" | "critic"'.
```

No entanto, e se a interface tiver dezenas de propriedades ou mais? Você terá de digitar as chaves de cada uma dessas propriedades no tipo de união e mantê-las atualizadas. Muito incômodo.

Em vez disso, o TypeScript fornece um operador `keyof` que recebe um tipo existente e retorna uma união de todas as chaves permitidas nesse tipo. Você pode inseri-lo na frente do nome de um tipo onde quer que precise usar um, como em uma anotação de tipo.

No código a seguir, `keyof Ratings` é equivalente a `'audience' | 'critic'`, mas é muito mais fácil de escrever e não precisará ser atualizado manualmente se em algum momento a interface `Ratings` mudar:

```
function getCountKeyof(ratings: Ratings, key: keyof Ratings): number {
    return ratings[key]; // Ok
}

const ratings: Ratings = { audience: 66, critic: 84 };

getCountKeyof(ratings, 'audience'); // Ok

getCountKeyof(ratings, 'not valid');
// ~~~~~
// Error: Argument of type '"not valid"' is not
// assignable to parameter of type 'keyof Ratings'.
```

`keyof` é um ótimo recurso para a criação de tipos de união baseados nas chaves de tipos existentes. Ele também combina bem com outros operadores de tipos do TypeScript, permitindo o uso de alguns padrões muito elegantes que você verá posteriormente neste capítulo e no Capítulo 15, “Operações com tipos”.

## typeof

Outro operador de tipo que o TypeScript oferece é `typeof`. Ele retorna o tipo de um valor fornecido. Isso pode ser útil se o tipo do valor for muito difícil de escrever manualmente.

Aqui, a variável `adaptation` é declarada como tendo o mesmo tipo de

```

original:

const original = {
  medium: "movie",
  title: "Mean Girls",
};

let adaptation: typeof original;

if (Math.random() > 0.5) {
  adaptation = { ...original, medium: "play" }; // Ok
} else {
  adaptation = { ...original, medium: 2 };
  //           ~~~~~
  // Error: Type 'number' is not assignable to type 'string'.
}

```

Embora o operador de *tipo* `typeof` se pareça visualmente com o operador de *runtime* `typeof` usado para retornar uma descrição do tipo de um valor no formato string, os dois são diferentes. Por coincidência eles apenas usam a mesma palavra. Lembre-se: o operador JavaScript é um operador de runtime que retorna o nome de um tipo no formato string. Já que é um operador de tipo, a versão do TypeScript só pode ser usada em tipos e não aparecerá no código compilado.

## keyof typeof

`typeof` recupera o tipo de um valor e `keyof` recupera as chaves permitidas em um tipo. O TypeScript permite que as duas palavras-chave sejam encadeadas para recuperarem de uma só vez as chaves permitidas no tipo de um valor. Quando usados em conjunto, o operador de tipo `typeof` passa a ser muito útil para o trabalho com as operações de tipo de `keyof`.

Neste exemplo, a função `logRating` recebe uma das chaves do valor de `ratings`. Em vez de criar uma interface, o código usa `keyof typeof` para indicar que `key` deve ser uma das chaves do tipo do valor de `ratings`:

```

const ratings = {
  imdb: 8.4,
  metacritic: 82,
};

```

```
function logRating(key: keyof typeof ratings) {
    console.log(ratings[key]);
}

logRating("imdb"); // Ok

logRating("invalid");
// ~~~~~
// Error: Argument of type '"missing"' is not assignable
// to parameter of type '"imdb" | "metacritic"'.
```

Combinando `keyof` e `typeof`, não precisamos escrever – e atualizar – os tipos que representam as chaves permitidas em objetos que não têm um tipo explícito de interface.

## Asserções de tipo

O TypeScript funciona melhor quando o código é “fortemente tipado”: os tipos de todos os valores do código são conhecidos com precisão. Recursos como os tipos universais e os type guards fornecem maneiras de limpar códigos complexos para que eles sejam entendidos pelo verificador de tipos do TypeScript. No entanto, às vezes não conseguimos ser 100% precisos ao informar para a tipagem como o código deve funcionar.

Por exemplo, `JSON.parse` retorna intencionalmente o tipo universal `any`. Não há uma maneira de informar seguramente para a tipagem que um determinado valor de tipo `string` fornecido para `JSON.parse` deve retornar algum tipo de valor específico. (Como veremos no Capítulo 10, “Genéricos”, adicionar a `parse` um tipo genérico que só seja usado uma vez como tipo de retorno violaria uma prática recomendada conhecida como A Regra de Ouro dos Genéricos).

O TypeScript fornece uma sintaxe que sobrescreve o conhecimento da tipagem sobre o tipo de um valor: uma “asserção de tipo”, também conhecida como “conversão de tipo” (type cast). Em um valor que deva ter um tipo diferente, você pode inserir a palavra-chave `as` seguida de um tipo. O TypeScript aceitará sua asserção e tratará o valor como desse tipo.

Neste trecho de código, é possível que o resultado retornado de `JSON.parse` tenha um tipo como `string[]`, `[string, string]` ou `["grace", "frankie"]`. O

código usa asserções de tipo em três das linhas para mudar o tipo de `any` para um desses outros tipos:

```
const rawData = `["grace", "frankie"]`;

// Type: any
JSON.parse(rawData);

// Type: string[]
JSON.parse(rawData) as string[];

// Type: [string, string]
JSON.parse(rawData) as [string, string];

// Type: ["grace", "frankie"]
JSON.parse(rawData) as ["grace", "frankie"];
```

As asserções de tipo só existem na tipagem do TypeScript. Elas são removidas junto com todas as outras sintaxes da tipagem na compilação para JavaScript. O código anterior terá essa aparência ao ser compilado para JavaScript:

```
const rawData = `["grace", "frankie"]`;

// Type: any
JSON.parse(rawData);

// Type: string[]
JSON.parse(rawData);

// Type: [string, string]
JSON.parse(rawData);

// Type: ["grace", "frankie"]
JSON.parse(rawData);
```



Se você estiver trabalhando com bibliotecas ou códigos mais antigos, pode ver uma sintaxe de conversão diferente parecida com `<type>item` em vez de `item as type`. Já que essa sintaxe é incompatível com a sintaxe JSX e, portanto, não funciona em arquivos `.tsx`, seu uso não é recomendado.

Geralmente, a prática recomendada no TypeScript é evitar o uso de

asserções de tipo quando possível. É melhor o código ser totalmente tipado e não precisar interferir no conhecimento que o TypeScript tem de seus tipos com o uso de asserções. No entanto, ocasionalmente existirão casos nos quais as asserções de tipo serão úteis e até mesmo necessárias.

## Asserção dos tipos de erro capturados

A manipulação de erros é outra operação na qual as asserções de tipo podem ser úteis. Geralmente é impossível saber que tipo um erro capturado em um bloco `catch` terá porque o código do bloco `try` pode lançar algum objeto diferente do esperado. Além disso, embora a prática recomendada no JavaScript seja sempre lançar uma instância da classe `Error`, alguns projetos lançam strings literais ou outros valores inesperados.

Se você tiver certeza absoluta de que uma área de código só lançará uma instância da classe `Error`, poderá usar uma asserção de tipo para tratar um erro capturado como `Error`. Este código acessa a propriedade `message` de um erro capturado que ele presume ser uma instância da classe `Error`:

```
try {
    // (code that may throw an error)
} catch (error) {
    console.warn("Oh no!", (error as Error).message);
}
```

Pode ser mais seguro se você usar um estreitamento de tipo, como uma verificação com `instanceof`, para se certificar se o erro lançado tem o tipo esperado. O código a seguir verifica se o erro lançado é uma instância da classe `Error` para saber se deve registrar a mensagem ou o próprio erro:

```
try {
    // (code that may throw an error)
} catch (error) {
    console.warn("Oh no!", error instanceof Error ? error.message : error);
}
```

## Asserções de não nulo

Outro caso de uso comum para as asserções de tipo é na remoção dos tipos `null` e/ou `undefined` de uma variável que só teoricamente, e não na

prática, possa incluí-los. Essa situação é tão comum que o TypeScript tem uma abreviação para ela. Em vez de escrever `as` e o tipo completo de um valor excluindo `null` e `undefined`, você pode usar `!` para indicar a mesma coisa. Em outras palavras, a asserção de não nulo com `!` declara que o tipo não é `null` ou `undefined`.

As duas asserções de tipo a seguir são idênticas já que ambas resultam em `Date` e não em `Date | undefined`:

```
// Inferred type: Date | undefined
let maybeDate = Math.random() > 0.5
    ? undefined
    : new Date();

// Asserted type: Date
maybeDate as Date;

// Asserted type: Date
maybeDate!;
```

As asserções de não nulo são particularmente úteis com APIs como `Map.get` que retornam um valor ou `undefined` se ele não existir.

Aqui, `seasonCounts` é um objeto `Map<string, number>`. Sabemos que ele contém uma chave "I Love Lucy", logo, a variável `knownValue` pode usar um `!` para remover `| undefined` de seu tipo:

```
const seasonCounts = new Map([
    ["I Love Lucy", 6],
    ["The Golden Girls", 7],
]);

// Type: string | undefined
const maybeValue = seasonCounts.get("I Love Lucy");

console.log(maybeValue.toUpperCase());
// ~~~~~
// Error: Object is possibly 'undefined'.

// Type: string
const knownValue = seasonCounts.get("I Love Lucy")!;

console.log(knownValue.toUpperCase()); // Ok
```

## Advertências para as asserções de tipo

As asserções de tipo, como o tipo `any`, são uma saída de emergência necessária para a tipagem do TypeScript. Logo, também como o tipo `any`, elas devem ser evitadas sempre que possível. Pode ser melhor termos tipos mais precisos representando o código do que fazer uma asserção sobre o tipo de um valor. As asserções costumam estar erradas no momento em que são escritas ou passam a estar erradas posteriormente quando o codebase muda.

Por exemplo, suponhamos que o exemplo de `seasonCounts` mudasse com o tempo para ter diferentes valores no mapa. Sua asserção de não nulo poderia continuar fazendo o código passar na verificação de tipos do TypeScript, mas talvez ocorra um erro no runtime:

```
const seasonCounts = new Map([
  ["Broad City", 5],
  ["Community", 6],
]);

// Type: string
const knownValue = seasonCounts.get("I Love Lucy")!;
console.log(knownValue.toUpperCase()); // No type error, but...
// Runtime TypeError: Cannot read property 'toUpperCase' of undefined.
```

As asserções de tipo devem ser usadas moderadamente e só quando você tiver certeza de que é seguro usá-las.

## Asserções versus declarações

Há uma diferença entre usar uma anotação de tipo para declarar o tipo de uma variável e usar uma asserção de tipo para alterar o tipo de uma variável com um valor inicial. O verificador de tipos do TypeScript executa a verificação de capacidade de atribuição no valor inicial de uma variável em relação à anotação de tipo da variável quando os dois existem. Uma anotação de tipo, entretanto, solicita explicitamente ao TypeScript que ignore parte de sua verificação de tipo.

O código a seguir cria dois objetos de tipo `Entertainer` com a mesma falha: uma propriedade `acts` ausente. O TypeScript pode capturar o erro na variável `declared` por causa de sua anotação de tipo : `Entertainer`. No

entanto, não conseguirá capturar o erro na variável `asserted` devido à asserção de tipo:

```
interface Entertainer {
    acts: string[];
    name: string;
}

const declared: Entertainer = {
    name: "Moms Mabley",
};
// Error: Property 'acts' is missing in type
// '{ one: number; }' but required in type 'Entertainer'.

const asserted = {
    name: "Moms Mabley",
} as Entertainer; // Ok, but...

// Both of these statements would fail at runtime with:
// Runtime TypeError: Cannot read properties of undefined (reading
// 'toPrecision')
console.log(declared.acts.join(", "));
console.log(asserted.acts.join(", "));
```

Logo, é preferível usar uma anotação de tipo ou permitir que o TypeScript infira o tipo de uma variável a partir de seu valor inicial.

## Capacidade de atribuição nas asserções

As asserções de tipo devem ser apenas um último recurso, para situações em que o tipo de algum valor não esteja muito correto. O TypeScript só permitirá asserções entre dois tipos se um deles for atribuível ao outro. Se a asserção for entre dois tipos sem qualquer relação, o TypeScript detectará e relatará um type error.

Por exemplo, mudar de um primitivo para outro não é permitido, já que os primitivos não têm nenhuma relação uns com os outros:

```
let myValue = "Stella!" as number;
//           ~~~~~
// Error: Conversion of type 'string' to type 'number'
// may be a mistake because neither type sufficiently
// overlaps with the other. If this was intentional,
// convert the expression to 'unknown' first.
```

Se você precisar realmente mudar o tipo de um valor para outro tipo totalmente não relacionado, poderá usar uma asserção de tipo dupla. Primeiro converta o valor para um tipo universal – `any` ou `unknown` – e depois converta o resultado para o tipo não relacionado:

```
let myValueDouble = "1337" as unknown as number; // Ok, but... eww.
```

As asserções de tipo duplas `as unknown as...` são perigosas e quase sempre sinalizam que há algo incorreto nos tipos do código. Se você usá-las como um último recurso da tipagem, esta pode não conseguir ajudá-lo quando alterações causarem um problema em um código que anteriormente funcionava. Ensino a usar asserções de tipo duplas apenas como um relato preventivo que ajuda a explicar a tipagem, e não para encorajar seu uso.

## Asserções `const`

No Capítulo 4, “Objetos”, introduzi a sintaxe `as const` para a alteração de um tipo de array mutável para um tipo de tupla somente leitura e prometi voltar a usá-la posteriormente no livro. Chegou a hora!

Geralmente, as asserções `const` podem ser usadas para indicar que qualquer valor – seja um array, um primitivo etc. – deve ser tratado como a versão constante e imutável de si próprio. Especificamente, `as const` aplica as três regras a seguir a qualquer que seja o tipo que receber:

- Os arrays são tratados como tuplas `readonly`, e não como arrays mutáveis.
- Os literais são tratados como literais, e não como seus equivalentes primitivos gerais.
- As propriedades dos objetos são consideradas `readonly`.

Você já viu arrays tornarem-se tuplas, como ocorre com este array que está sendo declarado como uma tupla:

```
// Type: (number | string)[]  
[0, ''];
```

```
// Type: readonly [0, '']  
[0, '') as const;
```

Examinaremos as duas outras alterações que `as const` produz.

## Literais para primitivos

Pode ser útil a tipagem considerar um valor literal como tendo seu tipo literal específico, em vez de ampliá-lo para seu primitivo geral.

Por exemplo, assim como as funções que retornam tuplas, pode ser útil uma função produzir um literal específico em vez de um primitivo geral. Essas funções também retornam valores que podem ser mais específicos – aqui, o tipo de retorno de `getNameConst` é o tipo mais específico "Maria Bamford" em vez do tipo geral `string`:

```
// Type: () => string
const getName = () => "Maria Bamford";

// Type: () => "Maria Bamford"
const getNameConst = () => "Maria Bamford" as const;
```

Também pode ser útil que determinados campos de um valor sejam literais mais específicos. Muitas bibliotecas populares demandam que o campo discriminante de um valor seja um literal específico para que os tipos do código possam gerar inferências mais objetivas sobre o valor. A variável `narrowJoke` a seguir tem uma propriedade `style` de tipo "`one-liner`" em vez de `string`, logo, ela pode ser fornecida em um local que precise do tipo `Joke`:

```
interface Joke {
    quote: string;
    style: "story" | "one-liner";
}

function tellJoke(joke: Joke) {
    if (joke.style === "one-liner") {
        console.log(joke.quote);
    } else {
        console.log(joke.quote.split("\n"));
    }
}

// Type: { quote: string; style: "one-liner" }
const narrowJoke = {
```

```

        quote: "If you stay alive for no other reason do it for spite.",
        style: "one-liner" as const,
    };

tellJoke(narrowJoke); // Ok

// Type: { quote: string; style: string }
const wideObject = {
    quote: "Time flies when you are anxious!",
    style: "one-liner",
};

tellJoke(wideObject);
// Error: Argument of type '{ quote: string; style: string; }'
// is not assignable to parameter of type 'LogAction'.
//   Types of property 'style' are incompatible.
//     Type 'string' is not assignable to type '"story" | "one-liner"'.

```

## Objetos somente leitura

Objetos literais como os usados como valor inicial de uma variável geralmente ampliam os tipos das propriedades da mesma forma como os valores iniciais de variáveis `let` os ampliam. Valores string como `'apple'` tornam-se primitivos como `string`, arrays são tipados como arrays em vez de tuplas, e assim por diante. Isso pode ser inconveniente se alguns desses valores ou todos eles precisarem ser usados posteriormente em um local que demande seu tipo literal específico.

Declarar um valor literal com `as const`, entretanto, muda o tipo inferido para que seja o mais específico possível. Todas as propriedades passam a ser `readonly`, os literais são considerados como tendo seu tipo literal específico em vez de seu tipo primitivo geral, os arrays tornam-se tuplas somente leitura, e assim por diante. Em outras palavras, aplicar uma asserção `const` a um valor literal torna esse valor um literal imutável e direciona recursivamente a mesma lógica a todas as suas propriedades.

Como exemplo, o valor de `preferencesMutable` no código a seguir é declarado sem `as const`, logo, os nomes têm o tipo primitivo `string` e podem ser modificados. No entanto, `favoritesConst` é declarada com `as const`, de modo que os valores de suas propriedades são literais e não podem ser

modificados:

```
function describePreference(preference: "maybe" | "no" | "yes") {
  switch (preference) {
    case "maybe":
      return "I suppose...";
    case "no":
      return "No thanks.";
    case "yes":
      return "Yes please!";
  }
}

// Type: { movie: string, standup: string }
const preferencesMutable = {
  movie: "maybe"
  standup: "yes",
};

describePreference(preferencesMutable.movie);
//           ~~~~~
// Error: Argument of type 'string' is not assignable
// to parameter of type '"maybe" | "no" | "yes"'.

preferencesMutable.movie = "no"; // Ok

// Type: readonly { readonly movie: "maybe", readonly standup: "yes" }
const preferencesReadOnly = {
  movie: "maybe"
  standup: "yes",
} as const;

describePreference(preferencesReadOnly.movie); // Ok

preferencesReadOnly.movie = "no";
//           ~~~~
// Error: Cannot assign to 'movie' because it is a read-only property.
```

## Resumo

Neste capítulo, você usou modificadores de tipo para pegar objetos e/ou tipos existentes e lhes dar novos tipos:

- Tipos universais: o altamente permissivo `any` e o altamente restritivo `unknown`.
- Operadores de tipos: uso de `keyof` para capturar as chaves de um tipo e/ou `typeof` para capturar o tipo de um valor.
- Quando usar ou não asserções de tipo para alterar o tipo de um valor.
- Estreitamento de tipos com o uso de asserções `as const`.



Agora que você terminou de ler este capítulo, pratique o que aprendeu em <https://learningtypescript.com/type-modifiers>.

*Por que o tipo literal estava sendo teimoso?  
Ele tinha uma mente estreita.<sup>3</sup>*

---

<sup>1</sup> N.T.: Diz-se na comunidade científica que certa vez um renomado cientista estava dando uma palestra sobre astronomia. Ele falava como a Terra orbita o Sol e como o Sol, por sua vez, orbita o centro de uma vasta quantidade de estrelas a que damos o nome de galáxia. Quando a palestra terminou, uma senhora se levantou e disse que aquilo que o homem havia acabado de falar não tem sentido, pois “o mundo é um prato achatado apoiado no dorso de uma tartaruga gigante”. O cientista questionou onde a tartaruga gigante estaria apoiada e a senhora respondeu que estaria apoiada em outra tartaruga. “Uma tartaruga abaixo da outra. Há tartarugas até lá embaixo”.

<sup>2</sup> N.T.: Original: *Types of types from types.* / “*It’s turtles all the way down,*” / *Anders likes to say.*

<sup>3</sup> N.T.: Original: *Why was the literal type being stubborn? It had a narrow mind.*

## CAPÍTULO 10

# Genéricos

*Variáveis*

*declaradas na tipagem?*

*Um mundo totalmente novo (e tipado)!<sup>1</sup>*

Todas as sintaxes de tipos que você aprendeu até agora devem ser usadas com tipos que sejam totalmente conhecidos quando estiverem sendo escritos. No entanto, um trecho do código pode ter de operar com tipos diferentes dependendo de como for chamado.

Considere esta função `identity` em JavaScript destinada a receber uma entrada de qualquer tipo possível e retornar essa mesma entrada como saída. Como você descreveria o tipo de seu parâmetro e seu tipo de retorno?

```
function identity(input) {  
    return input;  
}  
  
identity("abc");  
identity(123);  
identity({ quote: "I think your self emerges more clearly over time." });
```

Poderíamos declarar `input` como `any`, mas então o tipo de retorno da função também seria `any`:

```
function identity(input: any) {  
    return input;  
}  
  
let value = identity(42); // Type of value: any
```

Já que `input` pode ser qualquer entrada, precisamos de uma maneira de informar que há um relacionamento entre o tipo de `input` e o tipo que a

função retorna. O TypeScript captura relacionamentos entre tipos usando *genéricos (generics)*.

Em TypeScript, estruturas como as funções podem declarar qualquer número de *parâmetros de tipo* genéricos: tipos que são determinados para cada uso da estrutura genérica. Esses parâmetros são usados como tipos na estrutura para representar algum tipo que pode ser diferente em cada instância da estrutura. Os parâmetros de tipo podem ser fornecidos com tipos diferentes, chamados de *argumentos de tipo*, para cada instância da estrutura, mas permanecerão consistentes dentro dessa instância.

Normalmente, os parâmetros de tipo têm nomes com uma única letra como `T` e `U` ou nomes em PascalCase, como `key` e `value`. Em todas as estruturas abordadas neste capítulo, os genéricos serão declarados com o uso dos parênteses angulares `< e >`, como em `someFunction<T>` ou `SomeInterface<T>`.

## Funções genéricas

Uma função pode ser tornada genérica com a inserção de um alias de parâmetro de tipo, incluído em parênteses angulares, imediatamente antes dos parênteses dos parâmetros. Esse parâmetro de tipo ficará então disponível para uso em anotações de tipos de parâmetros, em anotações de tipos de retorno e em anotações de tipo dentro do corpo da função.

A versão de `identity` a seguir declara um parâmetro de tipo `T` para seu parâmetro `input`, o que permite que o TypeScript infira que o tipo de retorno da função é `T`. O TypeScript poderá então inferir um tipo diferente para `T` sempre que `identity` for chamada:

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

const numeric = identity("me"); // Type: "me"
const stringy = identity(123); // Type: 123
```

As arrow functions também podem ser genéricas. Suas declarações genéricas também são inseridas imediatamente antes do parêntese de

abertura que inicia sua lista de parâmetros.

A arrow function a seguir tem declaração funcionalmente igual à declaração anterior:

```
const identity = <T>(input: T) => input;  
  
identity(123); // Type: 123
```

 A sintaxe das arrow functions genéricas tem algumas restrições em arquivos `.tsx` porque entra em conflito com a sintaxe JSX. Consulte o Capítulo 13, “Opções de configuração”, para ver soluções e como configurar o suporte ao JSX e ao React.

O acréscimo de parâmetros de tipo às funções desta forma permite que eles sejam reutilizados em diferentes entradas mantendo ao mesmo tempo a segurança de tipo (type safety) e evitando tipos `any`.

## Chamadas com tipos genéricos explícitos

Quase sempre, quando chamamos funções genéricas, o TypeScript consegue inferir os argumentos de tipo de acordo com como a função está sendo chamada. Por exemplo, nas funções `identity` dos exemplos anteriores, o verificador de tipos do TypeScript usou um argumento fornecido para `identity` para inferir o argumento de tipo do parâmetro correspondente.

Infelizmente, assim como ocorre com as propriedades das classes e os tipos das variáveis, às vezes não existem informações suficientes na chamada de uma função para indicar para o TypeScript para que tipo seu argumento de tipo deve ser convertido. Normalmente isso ocorre quando uma estrutura genérica recebe outra estrutura genérica cujos argumentos de tipo não são conhecidos.

Por padrão, o TypeScript assume o tipo `unknown` para qualquer argumento de tipo que ele não consiga inferir.

Por exemplo, a função `logWrapper` a seguir recebe um callback com um tipo de parâmetro configurado com o parâmetro de tipo `Input`. O TypeScript poderá inferir o argumento de tipo se `logWrapper` for chamada com um callback que declare explicitamente o tipo de seu parâmetro. No entanto,

se o tipo do parâmetro for implícito, o TypeScript não terá como saber qual é o tipo de `Input`:

```
function logWrapper<Input>(callback: (input: Input) => void) {
    return (input: Input) => {
        console.log("Input:", input);
        callback(input);
    };
}

// Type: (input: string) => void
logWrapper((input: string) => {
    console.log(input.length);
});

// Type: (input: unknown) => void
logWrapper((input) => {
    console.log(input.length);
    //
    // ~~~~~
    // Error: Property 'length' does not exist on type 'unknown'.
});
```

Para evitarmos o uso do padrão `unknown`, as funções podem ser chamadas com um argumento de tipo genérico explícito que informe explicitamente ao TypeScript qual deve ser o argumento de tipo. O TypeScript executará a verificação de tipos na chamada genérica para se certificar se o parâmetro que está sendo solicitado é compatível com o tipo que foi fornecido como argumento de tipo.

Aqui, a função `logWrapper` vista anteriormente recebeu uma `string` explícita para seu genérico `Input`. O TypeScript poderá então inferir que o parâmetro `input` de tipo genérico `Input` do callback é de tipo `string`:

```
// Type: (input: string) => void
logWrapper<string>((input) => {
    console.log(input.length);
});

logWrapper<string>((input: boolean) => {
    //
    // ~~~~~
    // Argument of type '(input: boolean) => void' is not
    // assignable to parameter of type '(input: string) => void'.
    // Types of parameters 'input' and 'input' are incompatible.
```

```
//      Type 'string' is not assignable to type 'boolean'.
});
```

Assim como ocorre com as anotações de tipo das variáveis, os argumentos de tipo explícitos podem ser especificados em uma função genérica, mas com frequência não são necessários. Muitos desenvolvedores TypeScript só os especificam quando apropriado.

O uso de `logWrapper` mostrado a seguir especifica `string` tanto como argumento de tipo quanto como tipo de parâmetro da função. Um dos dois pode ser removido:

```
// Type: (input: string) => void
logWrapper<string>((input: string) => { /* ... */ });
```

A sintaxe `Name<Type>` para a especificação de um argumento de tipo será igual para todas as outras estruturas genéricas deste capítulo.

## Múltiplos parâmetros de tipo para funções

As funções podem definir qualquer número de parâmetros de tipo, separados por vírgulas. Cada chamada da função genérica pode definir seu próprio conjunto de valores para cada um dos parâmetros de tipo.

Neste exemplo, `makeTuple` declara dois parâmetros de tipo e retorna um valor tipado como uma tupla somente leitura com um dos parâmetros e depois com o outro:

```
function makeTuple<First, Second>(first: First, second: Second) {
    return [first, second] as const;
}

let tuple = makeTuple(true, "abc"); // Type of value: readonly [boolean,
                                  string]
```

É bom ressaltar que se uma função declarar múltiplos parâmetros de tipo, as chamadas a ela devem declarar explicitamente todos os tipos genéricos ou não declarar nenhum deles. O TypeScript ainda não suporta a inferência de apenas alguns dos tipos de uma chamada genérica.

Aqui, `makePair` também recebe dois parâmetros de tipo, logo, nenhum deles ou os dois devem ser especificados explicitamente:

```
function makePair<Key, Value>(key: Key, value: Value) {
    return { key, value };
```

```

}

// Ok: neither type argument provided
makePair("abc", 123); // Type: { key: string; value: number }

// Ok: both type arguments provided
makePair<string, number>("abc", 123); // Type: { key: string; value: number }
makePair<"abc", 123>("abc", 123); // Type: { key: "abc"; value: 123 }

makePair<string>("abc", 123);
//      ~~~~~
// Error: Expected 2 type arguments, but got 1.

```



Tente não usar mais de um ou dois parâmetros de tipo em uma estrutura genérica. Assim como ocorre com os parâmetros de funções no runtime, quanto maior for a quantidade que você usar, mais difícil será ler e entender o código.

## Interfaces genéricas

As interfaces também podem ser declaradas como genéricas. Elas seguem regras para os genéricos semelhantes às das funções: podem ter qualquer número de parâmetros de tipo declarados entre < e > após seu nome. Esse tipo genérico poderá ser usado posteriormente em qualquer local da declaração, como nos tipos das propriedades.

A declaração de `Box` a seguir tem um parâmetro de tipo `T` para uma propriedade. Criar um objeto declarado como `Box` com um argumento de tipo demandará que a propriedade `inside: T` tenha esse argumento de tipo:

```

interface Box<T> {
  inside: T;
}

let stringyBox: Box<string> = {
  inside: "abc",
};

let numberBox: Box<number> = {
  inside: 123,
}

```

```

let incorrectBox: Box<number> = {
  inside: false,
  // Error: Type 'boolean' is not assignable to type 'number'.
}

```

Fato interessante: os métodos internos de `Array` são definidos no TypeScript como uma interface genérica! `Array` usa um parâmetro de tipo `T` para representar o tipo de dado armazenado dentro de um array. Seus métodos `pop` e `push` se parecem com:

```

interface Array<T> {
  // ...

  /**
   * Removes the last element from an array and returns it.
   * If the array is empty, undefined is returned and the array is not
   * modified.
   */
  pop(): T | undefined;

  /**
   * Appends new elements to the end of an array,
   * and returns the new length of the array.
   * @param items new elements to add to the array.
   */
  push(...items: T[]): number;

  // ...
}

```

## Tipos inferidos de interfaces genéricas

Assim como nas funções genéricas, os argumentos de tipo das interfaces genéricas podem ser inferidos de acordo com o uso. O TypeScript fará o que puder para inferir os argumentos de tipo a partir dos tipos de valores fornecidos para um local declarado como recebendo um tipo genérico.

A função `getLast` a seguir declara um parâmetro de tipo `value` que é então usado para seu parâmetro `node`. Dessa forma, o TypeScript pode inferir `value` de acordo com o tipo de qualquer que seja o valor passado como argumento. Ele pode até mesmo relatar um type error quando um argumento de tipo inferido não tiver o tipo de um valor. Fornecer para

`getLast` um objeto que não inclua `next`, ou cujo argumento de tipo `Value` inferido tenha o mesmo tipo, é permitido. No entanto, uma incompatibilidade entre os argumentos `value` e `next.value` fornecidos para o objeto é considerada um type error:

```
interface LinkedNode<Value> {
    next?: LinkedNode<Value>;
    value: Value;
}

function getLast<Value>(node: LinkedNode<Value>): Value {
    return node.next ? getLast(node.next) : node.value;
}

// Inferred Value type argument: Date
let lastDate = getLast({
    value: new Date("09-13-1993"),
});

// Inferred Value type argument: string
let lastFruit = getLast({
    next: {
        value: "banana",
    },
    value: "apple",
});

// Inferred Value type argument: number
let lastMismatch = getLast({
    next: {
        value: 123
    },
    value: false,
// ~~~~~
// Error: type 'boolean' is not assignable to type 'number'.
});
}
```

É importante mencionar que se uma interface declarar parâmetros de tipo, qualquer anotação de tipo que a referenciar deve fornecer argumentos de tipo correspondentes. No código a seguir, o uso de `crateLike` está incorreto por não incluir um argumento de tipo:

```
interface CrateLike<T> {
    contents: T;
```

```

}

let missingGeneric: CrateLike = {
    // ~~~~~
    // Error: Generic type 'Crate<T>' requires 1 type argument(s).
    inside: "??"
};

}

```

Posteriormente neste capítulo, mostrarei como fornecer valores padrão para os parâmetros de tipo para atender a esse requisito.

## Classes genéricas

As classes, como as interfaces, também podem declarar qualquer número de parâmetros de tipo para serem usados posteriormente nas propriedades. Cada instância da classe pode ter um conjunto de argumentos de tipo diferente para seus parâmetros de tipo.

Esta classe `Secret` declara os parâmetros de tipo `Key` e `Value`, e depois os utiliza para propriedades, tipos de parâmetro do construtor, tipos de parâmetro dos métodos e do retorno:

```

class Secret<Key, Value> {
    key: Key;
    value: Value;

    constructor(key: Key, value: Value) {
        this.key = key;
        this.value = value;
    }

    getValue(key: Key): Value | undefined {
        return this.key === key
            ? this.value
            : undefined;
    }
}

const storage = new Secret(12345, "luggage"); // Type: Secret<number, string>

storage.getValue(1987); // Type: string | undefined

```

Assim como ocorre com as interfaces genéricas, anotações de tipo que

usarem uma classe devem indicar para o TypeScript quais são os tipos genéricos dessa classe. Posteriormente neste capítulo, mostrarei como fornecer valores padrão para os parâmetros de tipo para que esse requisito também seja atendido para as classes.

## Tipos explícitos de classes genéricas

A instanciação de classes genéricas segue as mesmas regras de inferência de argumentos de tipo das chamadas de funções genéricas. Se o argumento de tipo puder ser inferido a partir do tipo de um parâmetro passado para o construtor da classe, como anteriormente em `new Secret(12345, "luggage")`, o TypeScript usará o tipo inferido. Caso contrário, se o argumento de tipo de uma classe não puder ser inferido a partir dos argumentos passados para seu construtor, por padrão ele será `unknown`.

A classe `CurriedCallback` a seguir declara um construtor que recebe uma função genérica. Se a função genérica tiver um tipo conhecido – como da anotação de tipo de um argumento de tipo explícito – o argumento de tipo `Input` da instância da classe poderá ser inferido a partir dele. Caso contrário, por padrão o argumento de tipo `Input` será `unknown`:

```
class CurriedCallback<Input> {
    #callback: (input: Input) => void;

    constructor(callback: (input: Input) => void) {
        this.#callback = (input: Input) => {
            console.log("Input:", input);
            callback(input);
        };
    }

    call(input: Input) {
        this.#callback(input);
    }
}

// Type: CurriedCallback<string>
new CurriedCallback((input: string) => {
    console.log(input.length);
});
```

```
// Type: CurriedCallback<unknown>
new CurriedCallback((input) => {
    console.log(input.length);
    //
    // Error: Property 'length' does not exist on type 'unknown'.
});
});
```

As instâncias das classes também podem evitar serem por padrão `unknown` fornecendo argumentos de tipo explícitos da mesma forma que outras chamadas de funções genéricas o fazem.

Aqui, agora a classe `CurriedCallback` anterior está recebendo uma `string` explícita para seu argumento de tipo `Input`, logo, o TypeScript poderá inferir que o parâmetro de tipo `Input` do callback é uma `string`:

```
// Type: CurriedCallback<string>
new CurriedCallback<string>((input) => {
    console.log(input.length);
});

new CurriedCallback<string>((input: boolean) => {
    //
    // Argument of type '(input: boolean) => void' is not
    // assignable to parameter of type '(input: string) => void'.
    // Types of parameters 'input' and 'input' are incompatible.
    // Type 'string' is not assignable to type 'boolean'.
});
});
```

## Extensão de classes genéricas

As classes genéricas podem ser usadas como classe base quando vêm depois de uma palavra-chave `extends`. O TypeScript não tentará inferir argumentos de tipo para a classe base a partir do uso. Qualquer argumento de tipo sem padrões terá de ser especificado com o uso de uma anotação de tipo explícita.

A classe `SpokenQuote` a seguir fornece `string` como o argumento de tipo `T` de sua classe base `Quote<T>`:

```
class Quote<T> {
    lines: T;

    constructor(lines: T) {
        this.lines = lines;
```

```

        }
    }

class SpokenQuote extends Quote<string[]> {
    speak() {
        console.log(this.lines.join("\n"));
    }
}

new Quote("The only real failure is the failure to try.").lines; // Type:
string
new Quote([4, 8, 15, 16, 23, 42]).lines; // Type: number[]

new SpokenQuote([
    "Greed is so destructive.",
    "It destroys everything",
]).lines; // Type: string[]

new SpokenQuote([4, 8, 15, 16, 23, 42]);
// ~~~~~
// Error: Argument of type 'number' is not
// assignable to parameter of type 'string'.

```

Alternativamente, as classes genéricas derivadas podem passar seu próprio argumento de tipo para sua classe base. Os nomes dos tipos não precisam coincidir; como demonstração, esta classe `AttributedQuote` passa um argumento de tipo `Value` nomeado diferentemente para a classe base `Quote<T>`:

```

class AttributedQuote<Value> extends Quote<Value> {
    speaker: string

    constructor(value: Value, speaker: string) {
        super(value);
        this.speaker = speaker;
    }
}

// Type: AttributedQuote<string>
// (extending Quote<string>)
new AttributedQuote(
    "The road to success is always under construction.",
    "Lily Tomlin",
)

```

);

## Implementação de interfaces genéricas

As classes genéricas também podem implementar interfaces genéricas fornecendo para elas qualquer parâmetro de tipo que for necessário. Isso funciona de maneira semelhante à extensão de uma classe base genérica: qualquer parâmetro de tipo da interface base deve ser declarado pela classe.

A classe `MoviePart` a seguir especifica o argumento de tipo `Role` da interface `ActingCredit` como `string`. A classe `IncorrectExtension` causa um alerta de tipo porque seu argumento `role` é de tipo `boolean` apesar de ela fornecer `string[]` como argumento de tipo para `ActingCredit`:

```
interface ActingCredit<Role> {
    role: Role;
}

class MoviePart implements ActingCredit<string> {
    role: string;
    speaking: boolean;

    constructor(role: string, speaking: boolean) {
        this.role = role;
        this.speaking = speaking;
    }
}

const part = new MoviePart("Miranda Priestly", true);

part.role; // Type: string

class IncorrectExtension implements ActingCredit<string> {
    role: boolean;
    // ~~~~~
    // Error: Property 'role' in type 'IncorrectExtension' is not
    // assignable to the same property in base type 'ActingCredit<string>'.
    // Type 'boolean' is not assignable to type 'string'.
}
```

## Métodos genéricos

Os métodos das classes podem declarar seus próprios tipos genéricos separadamente da instância da sua classe. Cada chamada a um método de classe genérico pode ter um argumento de tipo diferente para cada um de seus parâmetros de tipo.

Esta classe genérica `CreatePairFactory` declara um tipo `Key` e inclui um método `createPair` que também declara um tipo genérico `Value` separado. O tipo de retorno de `createPair` é então inferido como sendo `{ key: Key, value: Value }`:

```
class CreatePairFactory<Key> {
    key: Key;

    constructor(key: Key) {
        this.key = key;
    }

    createPair<Value>(value: Value) {
        return { key: this.key, value };
    }
}

// Type: CreatePairFactory<string>
const factory = new CreatePairFactory("role");

// Type: { key: string, value: number }
const numberPair = factory.createPair(10);
// Type: { key: string, value: string }
const stringPair = factory.createPair("Sophie");
```

## Estáticos genéricos da classe

Os membros estáticos de uma classe ficam separados dos membros da instância e não são associados a nenhuma instância específica da classe. Eles não têm acesso a nenhuma instância da classe ou a informações de tipo específicas de alguma instância. Como resultado, embora os métodos de classe estáticos possam declarar seus próprios parâmetros de tipo, eles não podem acessar nenhum parâmetro de tipo declarado em uma classe.

Aqui, a classe `BothLogger` declara um parâmetro de tipo `onInstance` para seu método `instanceLog` e um parâmetro de tipo `onStatic` separado para seu

método estático `staticLog`. O método estático não pode acessar `OnInstance` porque esse parâmetro foi declarado para instâncias da classe:

```
class BothLogger<OnInstance> {
    instanceLog(value: OnInstance) {
        console.log(value);
        return value;
    }

    static staticLog<OnStatic>(value: OnStatic) {
        let fromInstance: OnInstance;
        // ~~~~~
        // Error: Static members cannot reference class type arguments.

        console.log(value);
        return value;
    }
}

const logger = new BothLogger<number[]>;
logger.instanceLog([1, 2, 3]); // Type: number[]

// Inferred OnStatic type argument: boolean[]
BothLogger.staticLog([false, true]);

// Explicit OnStatic type argument: string
BothLogger.staticLog<string>("You can't change the music of your soul.");
```

## Aliases de tipo genéricos

Uma última estrutura do TypeScript que pode ser tornada genérica com argumentos de tipo são os aliases de tipo. Cada alias de tipo pode receber qualquer número de parâmetros de tipo, como este tipo `Nullish` que está recebendo um `T`:

```
type Nullish<T> = T | null | undefined;
```

Normalmente, os aliases de tipo genéricos são usados com funções para descrever o tipo de uma função genérica:

```
type CreatesValue<Input, Output> = (input: Input) => Output;
```

```
// Type: (input: string) => number
```

```

let creator: CreatesValue<string, number>;
creator = text => text.length; // Ok

creator = text => text.toUpperCase();
// ~~~~~
// Error: Type 'string' is not assignable to type 'number'.

```

## Uniões discriminadas genéricas

Mencionei no Capítulo 4, “Objetos”, que as uniões discriminadas são meu recurso favorito em todo o TypeScript porque combinam um padrão comum e elegante do JavaScript com o estreitamento de tipos do TypeScript. Meu uso predileto para as uniões discriminadas seria para a inclusão de um argumento de tipo que criasse um tipo de “resultado” genérico para representar um resultado bem-sucedido com dados ou uma falha com um erro.

O tipo genérico `Result` a seguir contém um discriminante `succeeded` que deve ser usado para restringir um resultado para saber se é um sucesso ou uma falha. Isso significa que qualquer operação que retorne um `Result` poderá indicar um erro ou um resultado com dados, e pode ter certeza de que os consumidores precisarão verificar se o resultado foi bem-sucedido:

```

type Result<Data> = FailureResult | SuccessfulResult<Data>;

interface FailureResult {
    error: Error;
    succeeded: false;
}

interface SuccessfulResult<Data> {
    data: Data;
    succeeded: true;
}

function handleResult(result: Result<string>) {
    if (result.succeeded) {
        // Type of result: SuccessfulResult<string>
        console.log(`We did it! ${result.data}`);
    } else {
        // Type of result: FailureResult
    }
}

```

```

        console.error(`Awww... ${result.error}`);
    }

    result.data;
    //     ~~~~
    // Error: Property 'data' does not exist on type 'Result<string>'.
    //   Property 'data' does not exist on type 'FailureResult'.
}

```

Quando usados em conjunto, os tipos genéricos e os tipos discriminados fornecem uma excelente maneira de modelar tipos reutilizáveis como `Result`.

## Modificadores de genéricos

O TypeScript inclui uma sintaxe que permite modificar o comportamento de parâmetros de tipo genéricos.

### Padrões genéricos

Mencionei até agora que se um tipo genérico for usado em uma anotação de tipo ou como base da declaração `extends` ou `implements` de uma classe, ele deve fornecer um argumento de tipo para cada parâmetro de tipo. Você pode resolver o problema de fornecer argumentos de tipo explicitamente inserindo um sinal `=` seguido de um tipo padrão após a declaração do parâmetro de tipo. O padrão será usado em qualquer tipo subsequente para o qual o argumento de tipo não for declarado explicitamente e que não puder ser inferido.

A seguir, a interface `Quote` recebe um parâmetro de tipo `T` que por padrão será uma `string`, caso não seja fornecido. A variável `explicit` configura explicitamente `T` com `number` enquanto tanto `implicit` quanto `mismatch` recebem uma `string`:

```

interface Quote<T = string> {
    value: T;
}

let explicit: Quote<number> = { value: 123 };

let implicit: Quote = { value: "Be yourself. The world worships the original." }

```

```

};

let mismatch: Quote = { value: 123 };
//           ~~~
// Error: Type 'number' is not assignable to type 'string'.

```

Os parâmetros de tipo também podem ser por padrão parâmetros de tipo anteriores da mesma declaração. Já que cada parâmetro de tipo introduz um novo tipo para a declaração, eles estarão disponíveis como padrões para parâmetros de tipo posteriores dessa declaração.

O tipo `KeyValuePair` a seguir pode ter diferentes tipos para seus genéricos `Key` e `Value`, mas por padrão os mantêm iguais – no entanto, já que `Key` não tem um padrão, ele precisa ser inferível ou fornecido:

```

interface KeyValuePair<Key, Value = Key> {
    key: Key;
    value: Value;
}

// Type: KeyValuePair<string, string>
let allExplicit: KeyValuePair<string, number> = {
    key: "rating",
    value: 10,
};

// Type: KeyValuePair<string>
let oneDefaulting: KeyValuePair<string> = {
    key: "rating",
    value: "ten",
};

let firstMissing: KeyValuePair = {
    //           ~~~~~
    // Error: Generic type 'KeyValuePair<Key, Value>'
    // requires between 1 and 2 type arguments.
    key: "rating",
    value: 10,
};

```

Lembre-se de que todos os parâmetros de tipo padrão devem vir por último na lista de sua declaração, de maneira semelhante ao que ocorre com os parâmetros de função padrão. Tipos genéricos sem um padrão não podem vir depois de tipos genéricos que tenham um padrão.

A seguir, `inTheEnd` é permitido porque todos os tipos genéricos sem padrões vêm antes dos tipos genéricos com padrões. `inTheMiddle` é um problema porque um tipo genérico sem um padrão vem depois de tipos com padrões:

```
function inTheEnd<First, Second, Third = number, Fourth = string>() {} // Ok

function inTheMiddle<First, Second = boolean, Third = number, Fourth>() {}
//                                         // ~~~~~
// Error: Required type parameters may not follow optional type parameters.
```

## Tipos genéricos restritos

Por padrão, os tipos genéricos podem receber qualquer tipo existente: classes, interfaces, primitivos, uniões etc. No entanto, algumas funções só podem operar com um conjunto de tipos limitado.

O TypeScript permite que um parâmetro de tipo se declare precisando *estender* um tipo: o que significa que só serão permitidos alias de tipos que sejam atribuíveis a esse tipo. A sintaxe que restringe um parâmetro de tipo envolve inserir a palavra-chave `extends` após o nome do parâmetro, seguida do tipo ao qual ele ficará restrito.

Por exemplo, criando uma interface `WithLength` para descrever qualquer coisa que tenha um `length: number` (tamanho numérico) podemos permitir que nossa função genérica receba para seu genérico `T` qualquer tipo que tenha `length`. Strings, arrays e agora até mesmo objetos que tenham um `length: number` serão permitidos, enquanto formas de tipos como `Date` que não têm um `length` numérico resultarão em um type error:

```
interface WithLength {
    length: number;
}

function logWithLength<T extends WithLength>(input: T) {
    console.log(`Length: ${input.length}`);
    return input;
}

logWithLength("No one can figure out your worth but you."); // Type: string
logWithLength([false, true]); // Type: boolean[]
```

```

logWithLength({ length: 123 }); // Type: { length: number }

logWithLength(new Date());
//           ~~~~~
// Error: Argument of type 'Date' is not
// assignable to parameter of type 'WithLength'.
// Property 'length' is missing in type
// 'Date' but required in type 'WithLength'.

```

Abordarei mais operações de tipos que podem ser executadas com genéricos no Capítulo 15, “Operações com tipos”.

## keyof e os parâmetros de tipo restritos

O operador `keyof` introduzido no Capítulo 9, “Modificadores de tipo”, também funciona bem com parâmetros de tipo restritos. O uso conjunto de `extends` e `keyof` permite que um parâmetro de tipo fique restrito às chaves de um parâmetro de tipo anterior. Também é a única maneira de especificar a chave de um tipo genérico.

Veja a versão simplificada do método `get` a seguir da popular biblioteca Lodash. Ela recebe um valor de um contêiner, tipado como `T`, e um nome `key` de uma das chaves de `T` para recuperar em `container`. Já que o parâmetro de tipo `Key` está restrito a ser um `keyof T`, o TypeScript sabe que essa função pode retornar `T[Key]`:

```

function get<T, Key extends keyof T>(container: T, key: Key) {
    return container[key];
}

const roles = {
    favorite: "Fargo",
    others: ["Almost Famous", "Burn After Reading", "Nomadland"],
};

const favorite = get(roles, "favorite"); // Type: string
const others = get(roles, "others"); // Type: string[]

const missing = get(roles, "extras");
//           ~~~~~
// Error: Argument of type '"extras"' is not assignable
// to parameter of type '"favorite" / "others"'.

```

Sem `keyof`, não haveria como tipar corretamente o parâmetro genérico `key`.

Observe a importância do parâmetro de tipo `key` no exemplo anterior. Se só `T` for fornecido como parâmetro de tipo, e o parâmetro `key` puder ser qualquer `keyof T`, o tipo de retorno será o tipo de união de todos os valores das propriedades de `container`. A declaração de função menos específica a seguir indica para o TypeScript que cada chamada pode ter um `key` próprio por meio de um argumento de tipo:

```
function get<T>(container: T, key: keyof T) {
    return container[key];
}

const roles = {
    favorite: "Fargo",
    others: ["Almost Famous", "Burn After Reading", "Nomadland"],
};

const found = get(roles, "favorite"); // Type: string | string[]
```

Ao escrever funções genéricas, certifique-se de saber quando o tipo de um parâmetro dependerá de um tipo de parâmetro anterior. Com frequência você terá de usar parâmetros de tipo restritos para corrigir os tipos dos parâmetros nesses casos.

## Promises

Agora que você viu como os genéricos funcionam, finalmente chegou a hora de falarmos sobre um recurso básico do JavaScript moderno que depende de seus conceitos: as promises (promessas)! Recapitulando, uma promise em JavaScript representa algo que ainda pode estar pendente, como uma solicitação de rede. Cada promise fornece métodos para o registro de callbacks para o caso de a ação pendente ser “resolvida” (ser concluída com sucesso) ou ser “rejeitada” (lançar um erro).

A capacidade de uma promise representar ações semelhantes em qualquer tipo de valor arbitrário é um ajuste natural para os genéricos do TypeScript. As promises são representadas na tipagem do TypeScript como uma classe `Promise` com um único parâmetro de tipo representando o eventual valor resolvido.

## Criação de promises

O construtor de `Promise` é tipado no TypeScript como recebendo um único parâmetro. O tipo desse parâmetro depende de um parâmetro de tipo declarado na classe genérica `Promise`. Uma forma reduzida teria essa aparência:

```
class PromiseLike<Value> {
    constructor(
        executor: (
            resolve: (value: Value) => void,
            reject: (reason: unknown) => void,
        ) => void,
    ) { /* ... */ }
}
```

A criação de uma promise destinada a ser resolvida com um valor geralmente demanda a declaração explícita do argumento de tipo da promise. Por padrão, o TypeScript presumirá que o tipo do parâmetro é `unknown` se não houver esse argumento de tipo genérico explícito. O fornecimento explícito de um argumento de tipo para o construtor de `Promise` permitiria que o TypeScript conhecesse o tipo resultante resolvido da instância da promise:

```
// Type: Promise<unknown>
const resolvesUnknown = new Promise((resolve) => {
    setTimeout(() => resolve("Done!"), 1000);
});

// Type: Promise<string>
const resolvesString = new Promise<string>((resolve) => {
    setTimeout(() => resolve("Done!"), 1000);
});
```

O método `.then` genérico de uma promise introduz um novo parâmetro de tipo que representa o valor resolvido da promise que ele retorna.

Por exemplo, o código a seguir cria uma promise `textEventually` que é resolvida com um valor `string` após um segundo, assim como uma promise `lengthEventually` que aguarda um segundo adicional para ser resolvida com `number`:

```
// Type: Promise<string>
```

```

const textEventually = new Promise<string>((resolve) => {
    setTimeout(() => resolve("Done!"), 1000);
});

// Type: Promise<number>
const lengthEventually = textEventually.then((text) => text.length)

```

## Funções async

Qualquer função declarada em JavaScript com a palavra-chave `async` retorna uma `Promise`. Se um valor retornado por uma função `async` em JavaScript não for um Thenable (um objeto com um método `.then()`; na prática quase sempre uma promise), ele será encapsulado em uma `Promise` como se `Promise.resolve` tivesse sido chamada. O TypeScript reconhece isso e inferirá o tipo de retorno de uma função `async` como sempre sendo uma `Promise` para qualquer que seja o valor retornado.

A seguir, `lengthAfterSecond` retorna uma `Promise<number>` diretamente, enquanto `lengthImmediately` é inferida como retornando uma `Promise<number>` porque é assíncrona e retorna diretamente `number`:

```

// Type: (text: string) => Promise<number>
async function lengthAfterSecond(text: string) {
    await new Promise((resolve) => setTimeout(resolve, 1000))
    return text.length;
}

// Type: (text: string) => Promise<number>
async function lengthImmediately(text: string) {
    return text.length;
}

```

Portanto, qualquer tipo de retorno declarado manualmente em uma função `async` deve sempre ser de tipo `Promise`, mesmo se a função não mencionar promises explicitamente em sua implementação:

```

// Ok
async function givesPromiseForString(): Promise<string> {
    return "Done!";
}

async function givesString(): string {
    // ~~~~~

```

```
// Error: The return type of an async function
// or method must be the global Promise<T> type.
return "Done!";
}
```

## Uso correto dos genéricos

Como nas implementações de `Promise<Value>` anteriores deste capítulo, embora os genéricos possam nos dar muita flexibilidade na descrição de tipos no código, eles podem tornar-se complexos com muita rapidez. Programadores iniciantes em TypeScript costumam passar por uma fase de uso excessivo de genéricos ao ponto de criar código confuso de ler e difícil de manipular. Geralmente, a prática recomendada no TypeScript é usar genéricos somente quando necessário e deixar claro para o que eles estão sendo usados se o forem.



A maioria dos códigos que você escrever em TypeScript não deve fazer uso pesado dos genéricos a ponto de gerar confusão. No entanto, tipos de bibliotecas utilitárias, particularmente módulos de uso geral, podem às vezes usá-los intensivamente. Conhecer os genéricos é útil principalmente para podermos trabalhar de maneira eficiente com esses tipos utilitários.

## Regra de ouro dos genéricos

Um teste rápido que pode ajudar a mostrar se um parâmetro de tipo é necessário para uma função é se ele precisa ser usado pelo menos duas vezes. Os genéricos descrevem relacionamentos entre tipos, logo, se um parâmetro de tipo genérico só aparecer em um local, ele não deve estar definindo um relacionamento entre vários tipos.

Cada parâmetro de tipo de uma função deve ser usado para um parâmetro e depois também para pelo menos outro parâmetro e/ou para o tipo de retorno da função.

Por exemplo, esta função `logInput` usa seu parâmetro de tipo `Input` exatamente uma vez, para declarar seu parâmetro `input`:

```
function logInput<Input extends string>(input: Input) {
```

```
    console.log("Hi!", input);
}
```

Ao contrário das funções `identify` anteriores do capítulo, `logInput` não faz nada com seu parâmetro de tipo, como retornar ou declarar mais parâmetros. Portanto, não há muita utilidade na declaração do parâmetro de tipo `Input`. Podemos reescrever `logInput` sem ele:

```
function logInput(input: string) {
    console.log("Hi!", input);
}
```

*Effective TypeScript* de Dan Vanderkam (O'Reilly, 2019) contém muitas dicas ótimas de como trabalhar com genéricos, incluindo uma seção chamada “The Golden Rule of Generics”. É altamente recomendável que você leia *Effective TypeScript* e principalmente essa seção se estiver passando muito tempo brigando com os genéricos em seu código.

## Convenções de nomeação dos genéricos

A convenção de nomeação padrão para parâmetros de tipo em muitas linguagens, incluindo o TypeScript, é chamar o primeiro argumento de tipo de “T” (que vem de “tipo” ou “template”) e se existirem parâmetros de tipo subsequentes, chamá-los de “U,” “V”, e assim por diante.

Quando alguma informação contextual é conhecida sobre como o argumento de tipo deve ser usado, às vezes a convenção é estendida para a utilização da primeira letra do termo desse uso: por exemplo, bibliotecas de gerenciamento de estados podem se referir a um estado genérico usando “S”. “K” e “V” com frequência se referem a chaves (keys) e valores nas estruturas de dados.

Infelizmente, nomear um argumento de tipo apenas com uma letra pode ser tão confuso quanto nomear uma função ou variável com um único caractere:

```
// What on earth are L and V?!
function labelBox<L, V>(l: L, v: V) { /* ... */ }
```

Se a finalidade de um genérico não ficar clara a partir da letra individual `T`, pode ser melhor usar nomes de tipo genérico descritivos que indiquem para que o tipo é usado:

```
// Much more clear.  
function labelBox<Label, Value>(label: Label, value: Value) { /* ... */ }
```

Sempre que uma estrutura tiver vários parâmetros de tipo, ou se a finalidade de um único argumento de tipo não ficar imediatamente clara, considere usar nomes por extenso para melhorar a legibilidade em vez de abreviações de letra única.

## Resumo

Neste capítulo você tornou “genéricos” classes, funções, interfaces e aliases de tipo, permitindo que eles operem com parâmetros de tipo:

- Uso de parâmetros de tipo para representar tipos diferentes entre os usos de uma estrutura.
- Fornecimento de argumentos de tipo explícitos ou implícitos na chamada a funções genéricas.
- Uso de interfaces genéricas para representar tipos de objeto genéricos.
- Inclusão de parâmetros de tipo em classes e como isso afeta seus tipos.
- Inclusão de parâmetros de tipo em aliases de tipo, principalmente com uniões de tipo discriminadas.
- Modificação de parâmetros de tipo genéricos com padrões (=) e restrições (extends).
- Como as promises e as funções `async` usam genéricos para representar um fluxo de dados assíncrono.
- Melhores práticas para os genéricos, incluindo sua Regra de Ouro e as convenções de nomeação.

Isso conclui a seção *Recursos* deste livro. Parabéns: agora você conhece todos os recursos mais importantes de sintaxe e verificação de tipos da tipagem do TypeScript para a maioria dos projetos!

A próxima seção, *Uso*, abordará como você pode configurar o TypeScript para executá-lo em seu projeto, como interagir com dependências externas e como ajustar a verificação de tipos e o JavaScript gerado. Esses são recursos importantes para o uso do TypeScript em seus projetos.

Existem algumas outras operações com tipos disponíveis na sintaxe do

TypeScript. Você não precisa conhecê-las bem para trabalhar na maioria dos projetos TypeScript – mas é interessante e útil conhecê-las. Eu as inseri na Parte IV, “Material complementar” após a Parte III, “Uso”, como um pequeno bônus caso você tenha tempo.



Agora que você terminou de ler este capítulo, pratique o que aprendeu em <https://learningtypescript.com/generics>.

*Por que os genéricos irritam os desenvolvedores?  
Eles estão sempre fazendo tipo com argumentos.<sup>2</sup>*

---

<sup>1</sup> N.T.: Original: *Variables you / declare in the type system? / A whole new (typed) world!*

<sup>2</sup> N.T.: Original: *Why do generics anger developers? They're always typing arguments.*

PARTE III

# **Uso**

## CAPÍTULO 11

# Arquivos de declaração

*Os arquivos de declaração*

*Têm código puramente de tipagem*

*Sem estruturas de runtime<sup>1</sup>*

Mesmo sendo ótimo escrever código em TypeScript e ainda que seja isso que você queira fazer, será preciso trabalhar com arquivos JavaScript brutos em seus projetos TypeScript. Muitos pacotes são escritos diretamente em JavaScript, e não em TypeScript. Até mesmo os pacotes que são escritos em TypeScript são distribuídos como arquivos JavaScript. Além disso, os projetos TypeScript precisam de uma maneira de receber informações sobre as formas dos tipos de recursos específicos do ambiente como variáveis globais e APIs. Um projeto sendo executado no, digamos, Node.js pode ter acesso a módulos internos do Node não disponíveis nos navegadores – e vice-versa.

O TypeScript permite declarar as formas dos tipos separadamente de sua implementação. Normalmente, as declarações de tipos são escritas em arquivos cujos nomes terminam com a extensão `.d.ts`, conhecidos como *arquivos de declaração (declaration files)*. Os arquivos de declaração costumam ser escritos dentro de um projeto, criados e distribuídos com o pacote npm compilado do projeto ou compartilhados como um pacote de “tipagens” autocontido.

## Arquivos de declaração

Geralmente, um arquivo de declaração `.d.ts` funciona de maneira semelhante a um arquivo `.ts`, exceto pela importante restrição de não ser permitido incluir código de runtime. Os arquivos `.d.ts` contêm apenas

descrições de valores, interfaces, módulos e tipos gerais do runtime. Eles não podem conter nenhum código de runtime que possa ser compilado para JavaScript.

Os arquivos de declaração podem ser importados como qualquer outro arquivo-fonte do TypeScript.

Este arquivo `types.d.ts` exporta uma interface `Character` usada por um arquivo `index.ts`:

```
// types.d.ts
export interface Character {
    catchphrase?: string;
    name: string;
}
// index.ts
import { Character } from "./types";

export const character: Character = {
    catchphrase: "Yee-haw!",
    name: "Sandy Cheeks",
};
```

 Os arquivos de declaração criam o que é conhecido como um *contexto de ambiente*, o que significa uma área de código na qual só podemos declarar tipos, e não valores.

Este capítulo é em grande parte dedicado aos arquivos de declaração e às formas mais comuns das declarações de tipo usadas dentro deles.

## Declaração de valores de runtime

Embora os arquivos de definição não possam criar valores de runtime como funções ou variáveis, eles podem declarar que essas estruturas existem com a palavra-chave `declare`. Isso informa à tipagem que alguma influência externa – com uma tag `<script>` de uma página web – criou o valor com um nome e um tipo específico.

A declaração de uma variável com `declare` usa a mesma sintaxe da declaração de uma variável comum, exceto por não ser permitido um valor inicial.

Este trecho de código declara uma variável `declared`, mas exibe uma mensagem de type error por tentar fornecer um valor para uma variável `initializer`:

```
// types.d.ts
declare let declared: string; // Ok

declare let initializer: string = "Wanda";
//           ~~~~~
// Error: Initializers are not allowed in ambient contexts.
```

As funções e classes também são declaradas de maneira semelhante às suas formas comuns, mas sem os corpos de funções ou métodos.

A função e o método `canGrantWish` a seguir são declarados apropriadamente sem um corpo, mas a função e o método `grantWish` são erros de sintaxe por tentar definir inapropriadamente um corpo:

```
// fairies.d.ts
declare function canGrantWish(wish: string): boolean; // Ok

declare function grantWish(wish: string) { return true; }
//           ^
// Error: An implementation cannot be declared in ambient contexts.

class Fairy {
    canGrantWish(wish: string): boolean; // Ok

    grantWish(wish: string) {
        //           ^
        // Error: An implementation cannot be declared in ambient contexts.
        return true;
    }
}
```

 As regras do `any` implícito do TypeScript funcionam para funções e variáveis declaradas em contextos de ambiente da mesma forma que em um código-fonte comum. Já que os contextos de ambiente podem não fornecer corpos de funções ou os valores iniciais das variáveis, geralmente anotações de tipo explícitas – incluindo anotações de tipo de retorno explícitas – são a única maneira de evitar que elas sejam implicitamente de tipo `any`.

Embora declarações de tipo usando a palavra-chave `declare` sejam mais comuns nos arquivos de definição `.d.ts`, essa palavra-chave também pode ser usada fora dos arquivos de declaração. Um arquivo módulo ou script pode usar `declare`. Isso pode ser útil se uma variável globalmente disponível for usada apenas nesse arquivo.

Aqui, uma variável `myGlobalValue` é definida em um arquivo `index.ts`, logo, ela só pode ser usada neste arquivo:

```
// index.ts
declare const myGlobalValue: string;

console.log(myGlobalValue); // Ok
```

É bom ressaltar que, embora formas de tipos como as das interfaces sejam permitidas com ou sem `declare` em arquivos de definição `.d.ts`, estruturas de runtime como as funções ou variáveis acionarão um alerta de tipo se não houver `declare`:

```
// index.d.ts
interface Writer {} // Ok
declare interface Writer {} // Ok

declare const fullName: string; // Ok: type is the primitive string
declare const firstName: "Liz"; // Ok: type is the literal "value"

const lastName = "Lemon";
// Error: Top-level declarations in .d.ts files must
// start with either a 'declare' or 'export' modifier.
```

## Valores globais

Já que os arquivos do TypeScript que não têm instruções `import` ou `export` são tratados como *scripts* em vez de *módulos*, as estruturas – incluindo os tipos – declaradas neles ficam disponíveis globalmente. Arquivos de definição sem nenhuma importação ou exportação podem se beneficiar desse comportamento para declarar tipos globalmente. Os arquivos de definição globais são particularmente úteis para a declaração de tipos ou variáveis globais para todos os arquivos de uma aplicação.

A seguir, um arquivo `globals.d.ts` declara que `const version: string` existe globalmente. Logo, um arquivo `version.ts` pode referenciar uma variável

`version` global apesar de não importá-la de `globals.d.ts`:

```
// globals.d.ts
declare const version: string;
// version.ts
export function logVersion() {
    console.log(`Version: ${version}`); // Ok
}
```

Valores declarados globalmente são usados com mais frequência em aplicações de navegador que empregam variáveis globais. Embora os frameworks web mais modernos geralmente usem técnicas mais novas como os ECMAScript modules, ainda pode ser útil – principalmente em projetos menores – conseguir armazenar as variáveis globalmente.



Se você descobrir que não consegue acessar automaticamente os tipos globais declarados em um arquivo `.d.ts`, confirme se o arquivo não está importando e exportando algo. Até mesmo uma única exportação fará o arquivo inteiro não estar mais disponível globalmente!

## Mesclagem de interface global

As variáveis não são os únicos elementos globais da tipagem de um projeto TypeScript. Muitas declarações de tipo existem globalmente para APIs e valores globais. Já que as interfaces se mesclam com outras interfaces de mesmo nome, declarar uma interface em um contexto de script global – como em um arquivo de declaração `.d.ts` sem nenhuma instrução `import` ou `export` – a aumenta globalmente.

Por exemplo, uma aplicação web que dependesse de uma variável global definida pelo servidor poderia declará-la como existente na interface global `window`. A mesclagem de interfaces permitiria que um arquivo como `types/window.d.ts` declarasse uma variável como existente na variável global `window` de tipo `Window`:

```
<script type="text/javascript">
window.myVersion = "3.1.1";
</script>
// types/window.d.ts
interface Window {
```

```
    myVersion: string;
}
// index.ts
export function logWindowVersion() {
    console.log(`Window version is: ${window.myVersion}`);
    window.alert("Built-in window types still work! Hooray!")
}
```

## Aumentos globais

Nem sempre é possível evitar instruções `import` ou `export` em um arquivo `.d.ts` que também precise aumentar o escopo global como quando as definições globais são simplificadas pela importação de um tipo definido em outro local. Às vezes tipos declarados em um arquivo módulo precisam ser consumidos globalmente.

Para esses casos, o TypeScript permite o uso de uma sintaxe (`declare global`) que declara um bloco de código como global. Isso marcará o conteúdo desse bloco como estando em um contexto global ainda que o mesmo não ocorra com o código ao redor:

```
// types.d.ts
// (module context)

declare global {
    // (global context)
}

// (module context)
```

A seguir, um arquivo `types/data.d.ts` exporta uma interface `Data` que, posteriormente, será importada tanto por `types/globals.d.ts` quanto pelo arquivo de runtime `index.ts`:

```
// types/data.d.ts
export interface Data {
    version: string;
}
```

Além disso, `types/globals.d.ts` declara uma variável de tipo `Data` globalmente dentro de um bloco `declare global` assim como uma variável disponível apenas nesse arquivo:

```
// types/globals.d.ts
```

```
import { Data } from "./data";

declare global {
    const globallyDeclared: Data;
}
```

```
declare const locallyDeclared: Data;
```

*index.ts* tem então acesso à variável `globallyDeclared` sem uma importação e ainda precisa importar `Data`:

```
// index.ts
import { Data } from "./types/data";

function logData(data: Data) { // Ok
    console.log(`Data version is: ${data.version}`);
}

logData(globallyDeclared); // Ok

logData(locallyDeclared);
// ~~~~~
// Error: Cannot find name 'locallyDeclared'.
```

Pode ser complicado fazer declarações globais e de módulo interagirem de maneira apropriada. O uso adequado das palavras-chave `declare` e `global` do TypeScript pode descrever quais definições de tipo devem estar disponíveis globalmente nos projetos.

## Declarações internas

Agora que você viu como as declarações funcionam, é hora de descobrir seu uso oculto no TypeScript: elas estiveram o tempo todo por trás da verificação de tipos! Objetos globais como `Array`, `Function`, `Map` e `Set` são exemplos de estruturas das quais a tipagem precisa ter informações, mas que não são declaradas no código. Elas são fornecidas por qualquer que seja o runtime no qual o código precise ser executado: o Deno, o Node, um navegador web etc.

## Declarações de biblioteca

Objetos globais internos, como `Array` e `Function`, que existem em todos os

runtimes JavaScript são declarados em arquivos com nomes como *lib.[target].d.ts*. *target* é a versão de suporte mínima do JavaScript demandada pelo projeto, como ES5, ES2020 ou ESNext.

Os arquivos de definição de biblioteca (library definition files) internos, ou “arquivos lib”, são grandes porque representam a totalidade das APIs internas do JavaScript. Por exemplo, as propriedades do tipo interno `Array` são representadas por uma interface global `Array` que começa assim:

```
// lib.es5.d.ts

interface Array<T> {
  /**
   * Gets or sets the length of the array.
   * This is a number one higher than the highest index in the array.
   */
  length: number;

  // ...
}
```

Os arquivos lib são distribuídos como parte do pacote npm do TypeScript. Você pode encontrá-los dentro do pacote em caminhos como `node_modules/typescript/lib/lib.es5.d.ts`. Para IDEs, como o VS Code, que usam suas próprias versões empacotadas do TypeScript na verificação de tipos do código, você pode encontrar o arquivo lib que está sendo usado clicando com o botão direito do mouse em um método interno, como o método `forEach` de um array, e selecionando uma opção como Go to Definition (Figura 11.1).

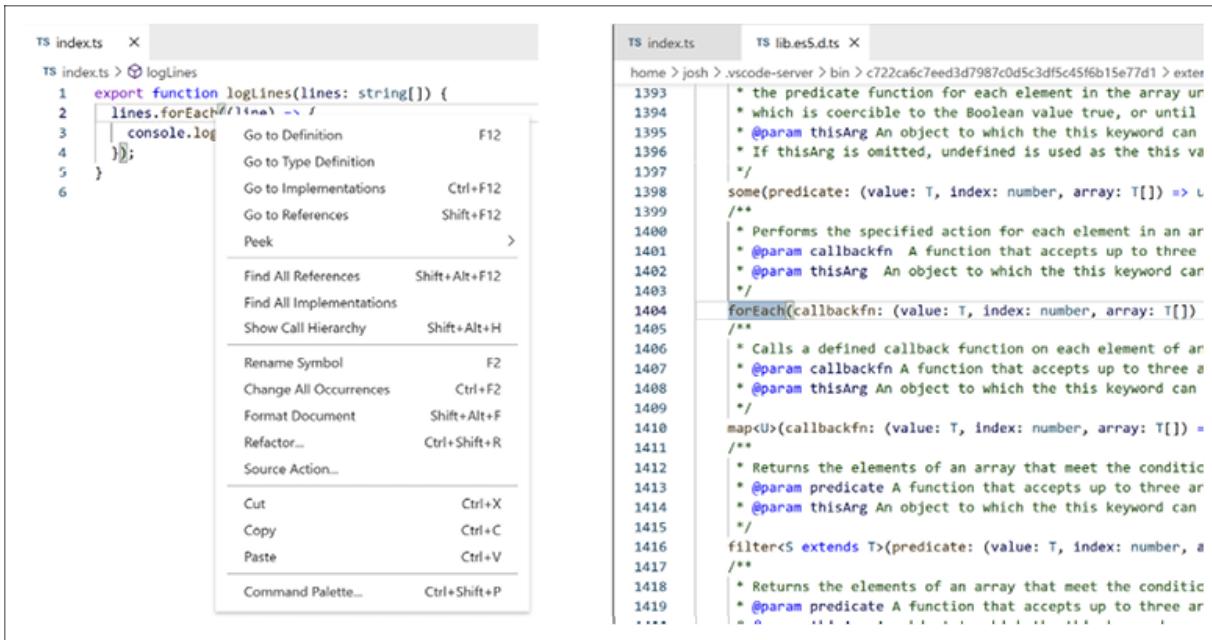


Figura 11.1: Esquerda: indo para a definição em um `forEach`; direita: o arquivo `lib.es5.d.ts` aberto resultante.

## Targets de biblioteca

O TypeScript inclui o arquivo lib apropriado de acordo com a definição do `target` fornecida para a CLI do `tsc` e/ou no arquivo `tsconfig.json` do projeto (por padrão, é usado o "es5"). Arquivos lib sucessivos de versões mais novas do JavaScript demandam que ambos estejam usando a mesclagem de interfaces.

Por exemplo, os membros estáticos `Number`, como `EPSILON` e `isFinite`, adicionados ao ES2015 estão listados em `lib.es2015.d.ts`:

```
// lib.es2015.d.ts

interface NumberConstructor {
  /**
   * The value of Number.EPSILON is the difference between 1 and the
   * smallest value greater than 1 that is representable as a Number
   * value, which is approximately:
   * 2.2204460492503130808472633361816 x 10-16.
  */
  readonly EPSILON: number;

  /**

```

```

    * Returns true if passed value is finite.
    * Unlike the global isFinite, Number.isFinite doesn't forcibly
    * convert the parameter to a number. Only finite values of the
    * type number result in true.
    * @param number A numeric value.
    */
isFinite(number: unknown): boolean;

// ...
}

```

Os projetos TypeScript incluirão os arquivos lib de todos os targets de versões do JavaScript até seu target mínimo. Por exemplo, um projeto com target "es2016" incluiria `lib.es5.d.ts`, `lib.es2015.d.ts` e `lib.es2016.d.ts`.



Recursos da linguagem disponíveis apenas em versões do JavaScript mais novas que a do seu target não estarão disponíveis na tipagem. Por exemplo, se seu target for "es5", recursos da linguagem do ES2015 ou posterior, como `String.prototype.startsWith`, não serão reconhecidos.

Opções de compilador como `target` serão abordadas com mais detalhes no Capítulo 13, “Opções de configuração”.

## Declarações DOM

Fora da linguagem JavaScript propriamente dita, a área de declarações de tipos mais conhecida é para os navegadores web. Os tipos dos navegadores web, geralmente chamados de tipos “DOM”, abordam APIs como `localStorage` e formas de tipos como `HTMLElement` disponíveis principalmente nos navegadores. Os tipos DOM são armazenados em um arquivo `lib.dom.d.ts` junto com os outros arquivos de declaração `lib.*.d.ts`.

Os tipos DOM globais, como muitos globais internos, costumam ser descritos com interfaces globais. Por exemplo, a interface `Storage`, usada para `localStorage` e `sessionStorage`, que começa de forma semelhante à descrita a seguir:

```

// lib.dom.d.ts

interface Storage {
  /**

```

```

    * Returns the number of key/value pairs.
    */
readonly length: number;

/**
 * Removes all key/value pairs, if there are any.
 */
clear(): void;

/**
 * Returns the current value associated with the given key,
 * or null if the given key does not exist.
 */
getItem(key: string): string | null;

// ...
}

```

Por padrão, o TypeScript incluirá tipos DOM em projetos que não sobrescreverem a opção de compilador `lib`. Isso pode ser confuso para desenvolvedores que estejam trabalhando em projetos para serem executados em ambientes que não sejam os de navegador, como o do Node, já que eles podem não conseguir acessar as APIs globais, como `document` e `localStorage`, que a tipagem alertará que existem. Opções de compilador como `lib` serão abordadas com mais detalhes no Capítulo 13, “Opções de configuração”.

## Declarações de módulo

Mais um recurso importante dos arquivos de declaração é sua capacidade de descrever as formas dos módulos. A palavra-chave `declare` pode ser usada antes de um nome no formato string para informar à tipagem o conteúdo desse módulo.

A seguir, o módulo "`my-example-lib`" é declarado como existindo em um arquivo de script de declaração `modules.d.ts` e, em seguida, ele é usado em um arquivo `index.ts`:

```
// modules.d.ts
declare module "my-example-lib" {
  export const value: string;
```

```
}

// index.ts
import { value } from "my-example-lib";

console.log(value); // Ok
```

Você não deve precisar usar as palavras-chave `declare module` com frequência, caso precise usá-las, em seu código. Elas são mais usadas com as declarações de módulo curinga da próxima seção e com os tipos de pacote que serão abordados posteriormente neste capítulo. Além disso, consulte o Capítulo 13, “Opções de configuração” para obter informações sobre `resolveJsonModule`, uma opção de compilador que permite que o TypeScript reconheça nativamente importações provenientes de arquivos `.json`.

## Declarações de módulo curinga

Um uso das declarações de módulo seria para informar às aplicações web que uma extensão de arquivo não JavaScript/TypeScript específica está disponível para ser importada para o código. As declarações de módulo podem conter um único curinga `*` para indicar que todos os módulos correspondentes a esse padrão serão iguais.

Por exemplo, muitos projetos web, como os pré-configurados em starters populares do React, como `create-react-app` e `create-next-app`, suportam módulos CSS para a importação de estilos de arquivos CSS como objetos que possam ser usados no runtime. Eles definiriam módulos com um padrão como `"*.module.css"` que por convenção exportaria um objeto de tipo `{ [i: string]: string }`:

```
// styles.d.ts
declare module "*.module.css" {
    const styles: { [i: string]: string };
    export default styles;
}
// component.ts
import styles from "./styles.module.css";

styles.anyClassName; // Type: string
```



O uso de módulos curingas para representar arquivos locais não é totalmente type safe. O TypeScript não fornece um mecanismo que verifique se o caminho do módulo importado é o de um arquivo local. Alguns projetos usam um sistema de build como o Webpack e/ou geram arquivos *.d.ts* a partir de arquivos locais para assegurar que as importações coincidam.

## Tipos de pacote

Agora que você viu como declarar tipagens dentro de um projeto, é hora de examinarmos o consumo de tipos entre os pacotes. Geralmente, os projetos escritos em TypeScript distribuem pacotes contendo saídas *.js* compiladas. Eles costumam usar arquivos *.d.ts* para declarar as formas de tipagem do TypeScript existentes por trás desses arquivos JavaScript.

### declaration

O TypeScript fornece uma opção `declaration` para a criação de saídas *.d.ts* para arquivos de entrada junto com saídas JavaScript.

Por exemplo, dado o arquivo-fonte *index.ts* a seguir:

```
// index.ts
export const greet = (text: string) => {
    console.log(`Hello, ${text}!`);
};
```

Com o uso de `declaration`, de `module` igual a `"es2015"`, e de `target` igual a `"es2015"`, estas saídas seriam geradas:

```
// index.d.ts
export declare const greet: (text: string) => void;
// index.js
export const greet = (text) => {
    console.log(`Hello, ${text}!`);
};
```

Os arquivos *.d.ts* autogerados são a melhor maneira de um projeto criar definições de tipo para serem usadas pelos consumidores. Geralmente, é recomendável que a maioria dos pacotes escrita em TypeScript que produza saídas em arquivos *.js* também empacotem arquivos *.d.ts* junto

com esses arquivos.

Opções de compilador como `declaration` serão abordadas com mais detalhes no Capítulo 13, “Opções de configuração”.

## Tipos de pacote de dependências

O TypeScript consegue detectar e utilizar arquivos empacotados dentro das dependências `node_modules` de um projeto. Esses arquivos informarão à tipagem sobre as formas de tipos exportadas por esse pacote como se elas tivessem sido escritas dentro do mesmo projeto ou declaradas com um bloco de módulo `declare`.

Um módulo típico do npm que viesse com seus próprios arquivos de declaração `.d.ts` poderia ter uma estrutura de arquivos como esta:

```
lib/
  index.js
  index.d.ts
package.json
```

Como exemplo, o sempre popular executor de testes Jest foi escrito em TypeScript e fornece seus próprios arquivos `.d.ts` no pacote `jest`. Ele depende do pacote `@jest/globals` que fornece funções como `describe` e `it`, que o `jest` torna então disponível globalmente:

```
// package.json
{
  "devDependencies": {
    "jest": "^32.1.0"
  }
}
// using-globals.d.ts
describe("MyAPI", () => {
  it("works", () => { /* ... */ });
});
// using-imported.d.ts
import { describe, it } from "@jest/globals";

describe("MyAPI", () => {
  it("works", () => { /* ... */ });
});
```

Se recriássemos um subconjunto muito limitado dos pacotes de tipagens

do Jest a partir do zero, eles poderiam ter a aparência desses arquivos. O pacote `@jest/globals` exporta as funções `describe` e `it`. Em seguida, o pacote `jest` importa essas funções e aumenta o escopo global com as variáveis `describe` e `it` de tipo correspondente ao de sua função:

```
// node_modules/@jest/globals/index.d.ts
export function describe(name: string, test: () => void): void;
export function it(name: string, test: () => void): void;
// node_modules/jest/index.d.ts
import * as globals from "@jest/globals";

declare global {
    const describe: typeof globals.describe;
    const it: typeof globals.it;
}
```

Essa estrutura permite que projetos que usarem o Jest referenciem versões globais de `describe` e `it`. Alternativamente, os projetos podem optar por importar essas funções do pacote `@jest/globals`.

## Exposição de tipos de pacote

Se o seu projeto for distribuído no npm e fornecer tipos para consumidores, adicione um campo "types" ao arquivo `package.json` do pacote a fim de que ele aponte para o arquivo de declaração raiz. O campo `types` funciona de maneira semelhante ao campo `main` – e com frequência tem a mesma aparência, mas com a extensão `.d.ts` em vez de `.js`.

Por exemplo, neste arquivo de pacote `fictional`, o arquivo principal de runtime `./lib/index.js` aparece junto ao arquivo de tipos `./lib/index.d.ts`:

```
{
  "author": "Pendant Publishing",
  "main": "./lib/index.js",
  "name": "coffeetable",
  "types": "./lib/index.d.ts",
  "version": "0.5.22",
}
```

O TypeScript usaria então o conteúdo de `./lib/index.d.ts` como o material que deve ser fornecido para arquivos consumidores que façam importações a partir do pacote `utilitarian`.



Se o campo `types` não existir no arquivo `package.json` de um pacote, o TypeScript assumirá o valor padrão `./index.d.ts`. Isso espelha o comportamento padrão do npm de assumir um arquivo `./index.js` como o ponto de entrada principal para um pacote se ele não for especificado.

A maioria dos pacotes usa a opção de compilador `declaration` do TypeScript para criar arquivos `.d.ts` junto com saídas `.js` a partir de arquivos-fonte. As opções de compilador serão abordadas no Capítulo 13, “Opções de configuração”.

## DefinitelyTyped

Infelizmente, nem todos os projetos são escritos em TypeScript. Alguns desenvolvedores desafortunados ainda estão escrevendo seus projetos no velho JavaScript sem um verificador de tipos para ajudá-los. Muito triste.

Nossos projetos TypeScript ainda precisam ser informados sobre as formas de tipos dos módulos desses pacotes. A equipe e a comunidade do TypeScript criaram um repositório gigante chamado *DefinitelyTyped* para hospedar definições criadas pela comunidade para os pacotes. O DefinitelyTyped, ou DT na abreviação, é um dos repositórios mais ativos no GitHub. Ele contém milhares de pacotes de definições `.d.ts`, junto com a automação da verificação de propostas de alteração e da publicação de atualizações.

Os pacotes do DT são publicados no npm no escopo `@types` com o mesmo nome do pacote para os quais eles fornecem tipos. Por exemplo, a partir de 2022 o `@types/react` fornece definições de tipos para o pacote `react`.



Normalmente, os `@types` são instalados como `dependencies` ou `devDependencies`, embora ultimamente a diferença entre os dois tenha ficado confusa. Em geral, se o projeto for distribuído como um pacote npm, ele deve usar `dependencies` para que os consumidores do pacote tragam as definições de tipo usadas dentro dele. Se o projeto for uma aplicação autônoma, como uma criada e executada em um servidor, ele deve usar `devDependencies` para mostrar que os tipos são

apenas uma ferramenta do tempo de desenvolvimento.

Por exemplo, para um pacote utilitário que dependesse do `lodash` – que a partir de 2022 fornece um pacote `@types/lodash` separado – o arquivo `package.json` conteria linhas como estas:

```
// package.json
{
  "dependencies": {
    "@types/lodash": "^4.14.182",
    "lodash": "^4.17.21",
  }
}
```

O arquivo `package.json` de uma aplicação autônoma baseada no React poderia conter linhas semelhantes às descritas a seguir:

```
// package.json
{
  "dependencies": {
    "react": "^18.1.0"
  },
  "devDependencies": {
    "@types/react": "^18.0.9"
  },
}
```

Observe que os números do versionamento semântico (“semver”) nem sempre coincidem entre os pacotes `@types/` e os pacotes que eles representam. Você pode encontrar algum número que esteja desatualizado devido a uma versão de patch, como no exemplo do React, a uma versão anterior, o que ocorre no exemplo do Lodash, ou até mesmo a versões com alterações mais importantes.



Já que esses arquivos são criados pela comunidade, eles podem não acompanhar o ritmo do projeto pai ou ter pequenas imprecisões. Se o seu projeto for compilado com sucesso e mesmo assim você encontrar erros de runtime ao chamar bibliotecas, verifique se as assinaturas das APIs que está acessando mudaram. Isso é menos comum, embora já se tenha ouvido falar, em projetos maduros com superfícies de API estáveis.

## Disponibilidade dos tipos

Os pacotes JavaScript mais populares vêm com suas próprias tipagens ou disponibilizam tipagens por meio do DefinitelyTyped.

Se você quiser obter tipos de um pacote cujos tipos ainda não estejam disponíveis, as três opções mais comuns são:

- Envie um pull request para o DefinitelyTyped para criar seu pacote `@types/`.
- Use a sintaxe `declare module` introduzida anteriormente para escrever os tipos dentro de seu projeto.
- Desative `noImplicitAny` como abordado – e cujo uso desaconselho – no Capítulo 13, “Opções de configuração”.

Recomendo que você faça contribuições com tipos para o DefinitelyTyped se tiver tempo. Isso ajudará outros desenvolvedores TypeScript que também quiserem usar o pacote em questão.



Consulte [aka.ms/types](https://aka.ms/types) para ver se um pacote tem tipos embutidos ou se eles são disponibilizados por meio de um pacote `@types/` separado.

## Resumo

Neste capítulo, você usou arquivos de declaração e declarações de valores para informar ao TypeScript sobre módulos e valores não declarados em seu código-fonte:

- Criação de arquivos de declaração com a extensão `.d.ts`.
- Declaração de tipos e valores com a palavra-chave `declare`.
- Alteração de tipos globais com o uso de valores globais, mesclagens de interface global e aumentos globais.
- Configuração e uso de declarações internas de target, de biblioteca e do DOM no TypeScript.
- Declaração de tipos de módulos, incluindo módulos curingas.
- Como o TypeScript seleciona tipos em pacotes.
- Uso de DefinitelyTyped para a aquisição de tipos de pacotes que não

incluem seus próprios tipos.

 Agora que você terminou de ler este capítulo, pratique o que aprendeu em <https://learningtypescript.com/declaration-files>.

*O que os tipos do TypeScript dizem na região sul da América do Norte?*  
“Ora, te garanto!”<sup>2</sup>

---

<sup>1</sup> N.T.: Original: *Declaration files / Have purely type system code / No runtime constructs*

<sup>2</sup> N.T.: Original: *What do TypeScript types say in the American South? “Why, I do **declare!**!”*

## CAPÍTULO 12

# Uso de recursos do IDE

*Quando programamos com um  
IDE pela primeira vez  
parece que temos superpoderes.<sup>1</sup>*

Nenhuma linguagem de programação popular estaria completa sem o realce da sintaxe (syntax highlighting) e outros recursos do IDE para ajudar no desenvolvimento. Uma das maiores vantagens do TypeScript é que seu serviço de linguagem fornece um conjunto de poderosos auxiliares de desenvolvimento para código JavaScript e TypeScript. Este capítulo abordará alguns dos itens mais úteis.

É altamente recomendável que você teste esses recursos do IDE nos projetos TypeScript que desenvolveu no decorrer deste livro. Embora todos os exemplos e screenshots deste capítulo sejam do VS Code, meu editor favorito, qualquer IDE com suporte ao TypeScript aceitará tudo, ou quase tudo, que é abordado no capítulo. A partir de 2022 isso inclui o suporte nativo ou plugins do TypeScript pelo menos para o Atom, Emacs, Vim, Visual Studio e WebStorm.



Este capítulo é uma lista não definitiva de alguns dos recursos de IDE mais úteis do TypeScript, junto com qualquer atalho padrão que exista para eles no VS Code. Provavelmente, você encontrará outros conforme for escrevendo mais código TypeScript.

Normalmente, muitos dos recursos do IDE são disponibilizados no menu de contexto gerado por um clique com o botão direito do mouse sobre um nome no código. IDEs como o VS Code também costumam exibir atalhos de teclado no menu de contexto. Conhecer os atalhos de teclado de seu IDE pode ajudá-lo a escrever código e executar refatorações com

muito mais rapidez.

Este screenshot exibe a lista de comandos e seus atalhos no VS Code para uma variável em TypeScript (Figura 12.1).

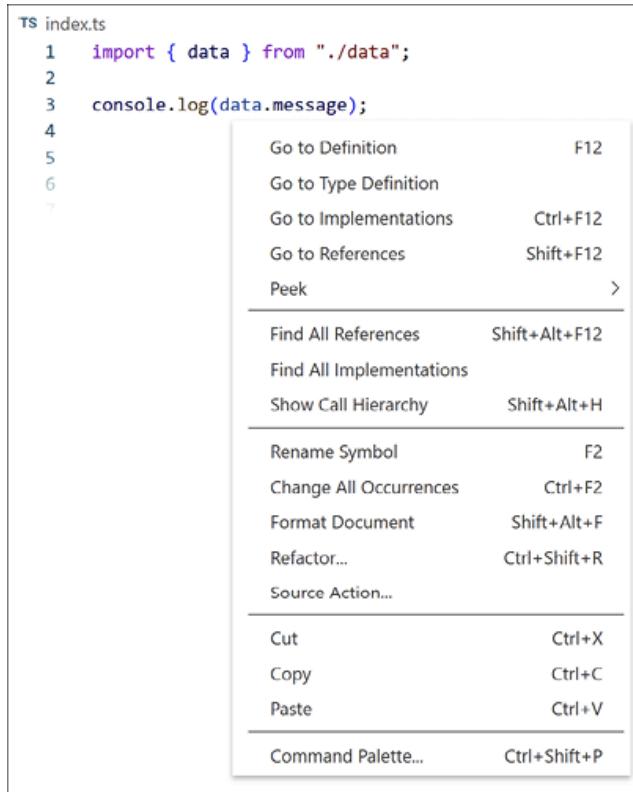


Figura 12.1: O VS Code exibindo uma lista de comandos no menu de contexto de uma variável gerado por um clique com o botão direitito do mouse.

 No VS Code, assim como na maioria das aplicações, setas para cima e para baixo selecionam opções suspensas e a tecla Enter ativa uma.

## Navegação no código

Os desenvolvedores costumam passar muito mais tempo lendo código do que escrevendo. Ferramentas que ajudam na navegação pelo código são muito úteis para diminuir esse tempo. Vários dos recursos fornecidos pelo serviço de linguagem do TypeScript são destinados ao conhecimento do código: especificamente, saltar entre os valores ou as definições de tipo do código e os locais onde eles são usados.

Agora percorrerei as opções de navegação normalmente usadas no menu de contexto junto com seus atalhos no VS Code.

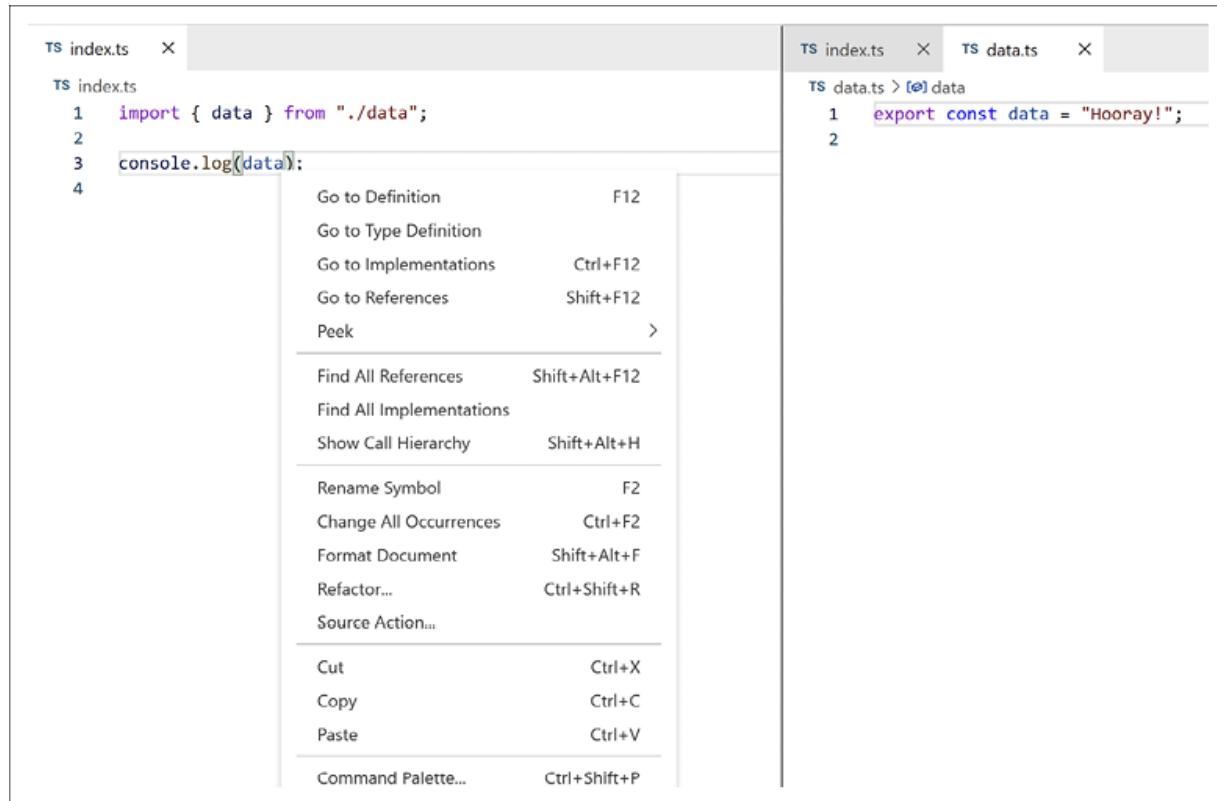
## Busca de definições

O TypeScript pode partir de uma referência a uma definição de tipo e nos levar de volta ao seu local de origem no código. O VS Code também fornece algumas maneiras de fazer um rastreamento regressivo como este:

- Go to Definition (F12) navega diretamente para onde um nome solicitado foi originalmente definido.
- Cmd (Mac) / Ctrl (Windows) + clicar em um nome também aciona a ida à definição.
- Peek > Peek Definition (Option (Mac) / Alt (Windows) + F12) abre uma caixa de inspeção exibindo a definição.

Go to Type Definition é uma versão especializada de Go to Definition que vai até a definição de qualquer que seja o tipo que um valor tiver. Para uma instância de uma classe ou interface, ela revelará a própria classe ou interface em vez de onde a instância foi definida.

Estes screenshots mostram a busca da definição de uma variável `data` importada para um arquivo com Go to Definition (Figura 12.2).



*Figura 12.2: Esquerda: ida para a definição em um nome de variável; direita: o arquivo data.ts aberto resultante.*

Quando a definição for declarada no próprio código, como em um arquivo relativo, o editor o levará a esse arquivo. Normalmente, módulos de fora do código, como os pacotes npm, usam arquivos de declaração *.d.ts*.

## Busca de referências

Dada uma definição de tipo ou valor, o TypeScript pode exibir uma lista com todas as referências feitas a ele ou com os locais em que ele é usado no projeto. O VS Code fornece algumas maneiras de visualizar essa lista.

Go to References (Shift + F12) exibe uma lista de referências a um valor ou definição de tipo – começando por ela própria – em uma caixa de inspeção expansível logo abaixo do nome que recebeu o clique com o botão direito do mouse.

Por exemplo, a seguir temos a busca de referências da declaração de uma variável *data* em um arquivo, *data.ts*, que exibe tanto a declaração quanto

seu uso em outro arquivo, *index.ts* (Figura 12.3).

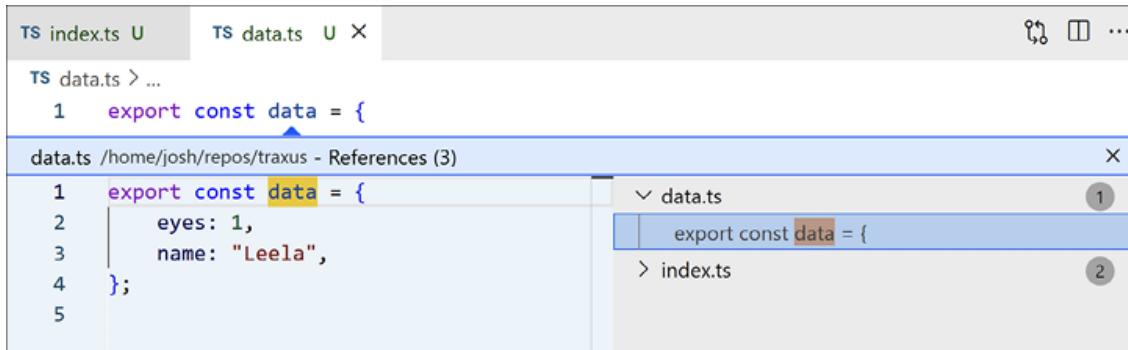


Figura 12.3: Menu de inspeção exibindo referências a uma variável.

Essa caixa de inspeção contém a visualização do arquivo que faz a referência. Você pode usar o arquivo – tipo, comandos de execução do editor, e assim por diante, – como se ele fosse um arquivo aberto regularmente. Também pode clicar duas vezes na visualização do arquivo na caixa de inspeção para abri-lo.

Clicar na lista de nomes de arquivo à direita na caixa de inspeção mudará a visualização para o arquivo clicado. Clicar duas vezes na linha de um arquivo na lista abrirá o arquivo e selecionará a referência correspondente.

Aqui, o VS Code está exibindo a declaração e o uso da mesma variável **data**, mas expandida na visualização da barra lateral à direita (Figura 12.4).

Find All References (Option (Mac) / Alt (Windows) + Shift + F12) também exibe uma lista de referências, mas em uma visualização de barra lateral que permanece visível após a navegação no código. Isso pode ser útil para a abertura ou execução de ações em mais de uma referência de cada vez (Figura 12.5).

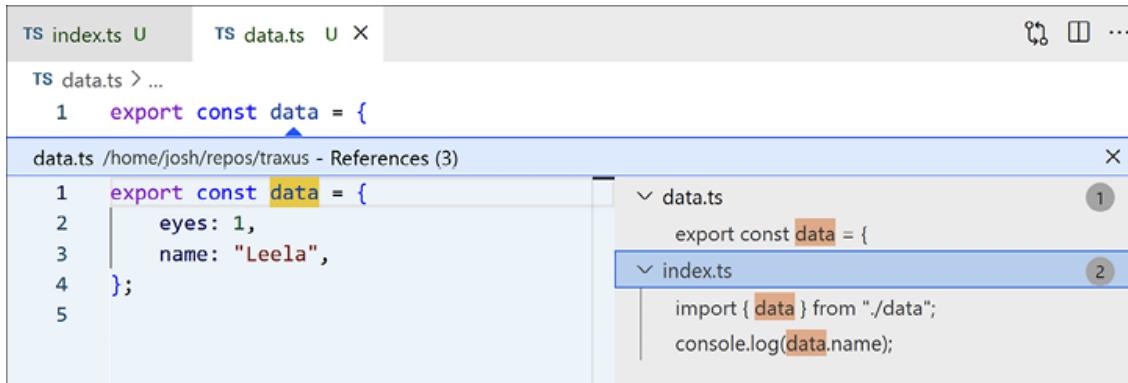


Figura 12.4: Menu de inspeção exibindo uma referência aberta a uma variável.

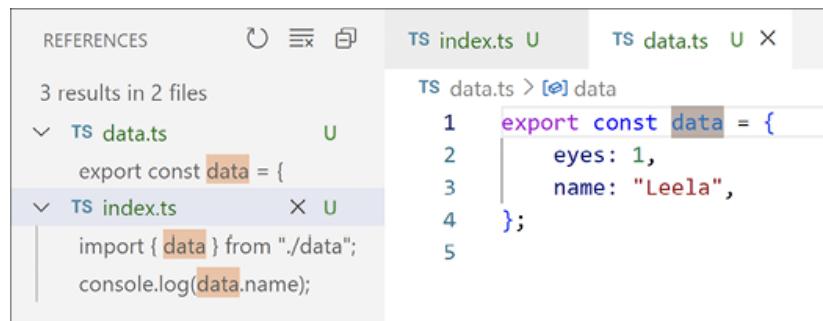


Figura 12.5: Menu Find All References de uma variável.

## Busca de implementações

Go to Implementations (Cmd (Mac) / Ctrl (Windows) + F12) e Find All Implementations são versões especializadas de Go To / Find All References criadas para interfaces e métodos de classe abstratos. Elas encontram todas as implementações de uma interface ou do método abstrato no código (Figura 12.6).



Figura 12.6: Menu Find All Implementations de uma interface AI.

Essas opções serão úteis principalmente se quisermos procurar como valores digitados de um tipo, como o de classe ou interface, estão sendo usados. A opção Find All References pode apresentar excesso de informações, já que também exibirá definições e outras referências de tipo da classe ou interface.

## Criação de código

Os serviços de linguagem dos IDEs, como o serviço TypeScript do VS Code, são executados em segundo plano no editor e reagem às ações executadas nos arquivos. Eles veem as edições feitas nos arquivos quando as digitamos – até mesmo antes das alterações serem salvas. Isso nos permite usar uma série de recursos que ajudam a automatizar tarefas comuns quando escrevemos código TypeScript.

### Preenchimento de nomes

As APIs do TypeScript também podem ser usadas pelos editores para o preenchimento de nomes que existam no mesmo arquivo. Quando começamos a digitar um nome, por exemplo quando fornecemos uma variável declarada anteriormente como argumento de uma função, com frequência os editores que usam TypeScript sugerem autopreenchimentos com uma lista de variáveis com nomes coincidentes. Clicar no nome na lista com o mouse ou pressionar a tecla Enter terminará de preencher o nome (Figura 12.7).

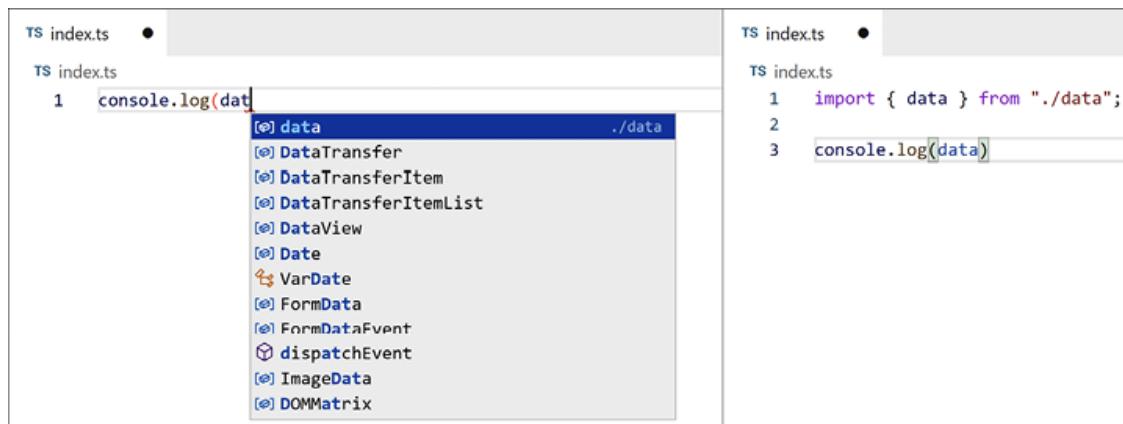
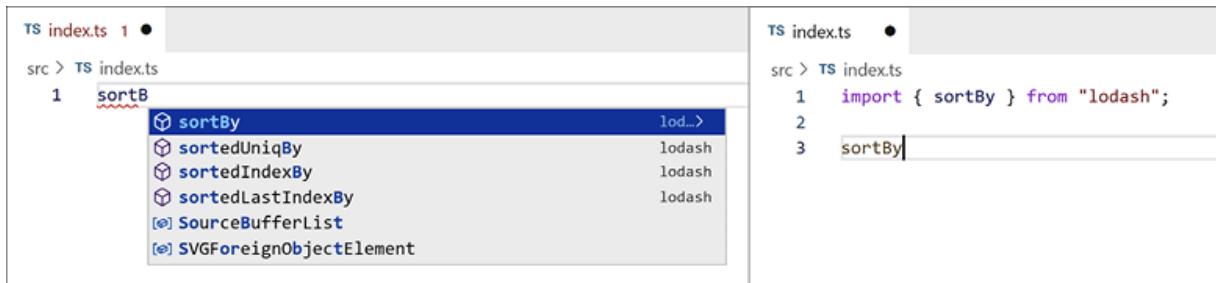


Figura 12.7: Esquerda: autopreenchimentos para uma variável digitada como

`dat; direita: o resultado do autocompletamento para uma variável data importada.`

Acréscimos de importação automática também serão oferecidos para dependências de pacote. Estes screenshots mostram o código de importação e de módulo de um arquivo TypeScript antes e depois de `sortBy` ser importada do pacote "lodash" (Figura 12.8).

As importações automáticas são um dos meus recursos favoritos da experiência do TypeScript. Elas costumam acelerar os geralmente trabalhosos processos de descobrir de onde vêm as importações e de, em seguida, digitá-las explicitamente.



*Figura 12.8: Esquerda: autocompletamentos para uma variável digitada como `sortB`; direita: resultado do autocompletamento para uma variável `sortBy` importada do `lodash`.*

Da mesma forma, se você começar a digitar o nome da propriedade de um valor tipado, editores habilitados com o TypeScript se oferecerão para fazer o autocompletamento com propriedades conhecidas que tenham o tipo do valor (Figura 12.9).



*Figura 12.9: Esquerda: autocompletamento de uma propriedade digitada como `forEach`; direita: resultado do autocompletamento para `.forEach`.*

## Atualizações de importações automáticas

Se você renomear um arquivo ou movê-lo de uma pasta para outra, pode ter de atualizar muitas instruções de importação. Atualizações podem ter de ser feitas tanto nesse arquivo quanto em qualquer outro arquivo que faça importações a partir dele.

Se você arrastar e soltar um arquivo ou renomeá-lo com um caminho de pasta aninhada usando o explorador de arquivos, o VS Code se oferecerá para usar o TypeScript para atualizar os caminhos de arquivo automaticamente.

Estes screenshots mostram um arquivo `src/logging.ts` sendo renomeado para um local `src/shared/logging.ts` e as importações de arquivo sendo atualizadas de maneira correspondente (Figura 12.10).



Edições em vários arquivos podem deixar as alterações feitas sem serem salvas. Lembre-se de salvar qualquer arquivo alterado após executar edições nele.

```
TS index.ts x
src > TS index.ts
1 import { data } from "./data";
2 import { log } from "./logging";
3
4 log(data);
5

TS index.ts x TS logging.ts
src > TS index.ts
1 import { data } from "./data";
2 import { log } from "./shared/logging";
3
4 log(data);
5
```

Figura 12.10: Esquerda: um arquivo `src/index.ts` fazendo importação de `./logging`; meio: renomeação de `src/logging.ts` para `src/shared/logging.ts`; direita: `src/index.ts` com um caminho de importação atualizado.

## Códigos de ação

Muitos dos utilitários de IDE do TypeScript são fornecidos como ações que podemos disparar. Embora algumas só modifiquem o arquivo que está sendo editado, outras podem modificar muitos arquivos ao mesmo tempo. Usar esses códigos de ação (code actions) é uma ótima maneira de fazer o TypeScript executar grande parte das tarefas de escrita manual de

seu código, como calcular caminhos de importação e executar refatorações comuns.

Geralmente, os códigos de ação são representados por algum tipo de ícone nos editores quando disponíveis. O VS Code, por exemplo, exibe uma lâmpada clicável próxima ao cursor de texto quando pelo menos um código de ação está disponível (Figura 12.11).

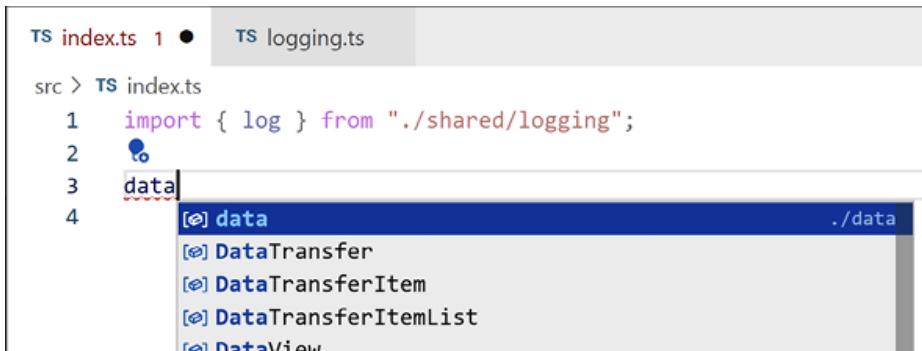


Figura 12.11: Lâmpada de códigos de ação próxima a um nome causando um type error.



Geralmente, os editores expõem atalhos de teclado para a operação de seu menu de códigos de ação ou equivalente, o que permitirá que você dispare qualquer ação deste capítulo sem usar um mouse. O atalho padrão do VS Code para a abertura de um menu de códigos de ação é Cmd + . no Mac e Ctrl + . no Linux/Windows. As setas para cima e para baixo selecionam as opções suspensas e Enter ativa uma delas.

Esses códigos de ação – principalmente renomeações e refatorações – são especialmente poderosos por receberem informações da tipagem do TypeScript. Quando uma ação é aplicada a um tipo, o TypeScript sabe quais valores de todos os arquivos são desse tipo e pode então aplicar qualquer alteração necessária a esses valores.

## Renomeação

Pode ser difícil alterar manualmente um nome que já existe, como o de uma função, interface ou variável. O TypeScript pode executar uma alteração de nome que também atualize todas as referências feitas a ele.

A opção de menu de contexto Rename Symbol (F2) cria uma caixa de texto na qual podemos digitar um novo nome. Disparar uma renomeação para o nome de uma função, por exemplo, forneceria uma caixa de texto para renomearmos essa função e todas as chamadas feitas a ela. Pressione Enter para aplicar o nome (Figura 12.12).

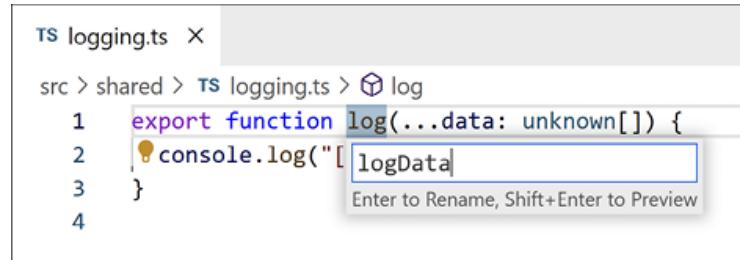


Figura 12.12: Caixa para renomeação de uma função log, com a inserção de logData.

Se quiser ver o que acontecerá antes de aplicar o novo nome, pressione Shift + Enter para abrir um painel Refactor Preview que listará todas as alterações de texto que ocorrerão (Figura 12.13).

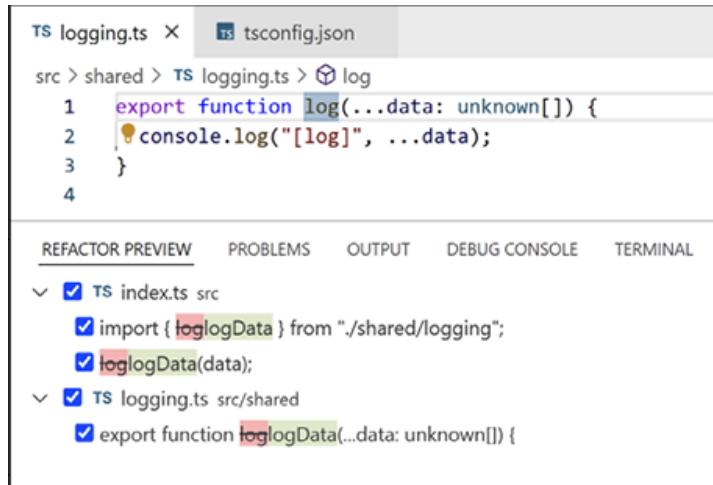


Figura 12.13: Visualização de refatoração para a renomeação de uma função log, com a exibição de logData em dois arquivos.

## Remoção de código não usado

Muitos IDEs alteram sutilmente a aparência de códigos que não são usados, como valores e variáveis importados que nunca são referenciados. O VS Code, por exemplo, reduz aproximadamente um terço de sua

opacidade.

O TypeScript fornece códigos de ação para a exclusão de código não usado. A Figura 12.14 mostra o que aconteceria se pedíssemos ao TypeScript que removesse uma instrução `import` não usada.

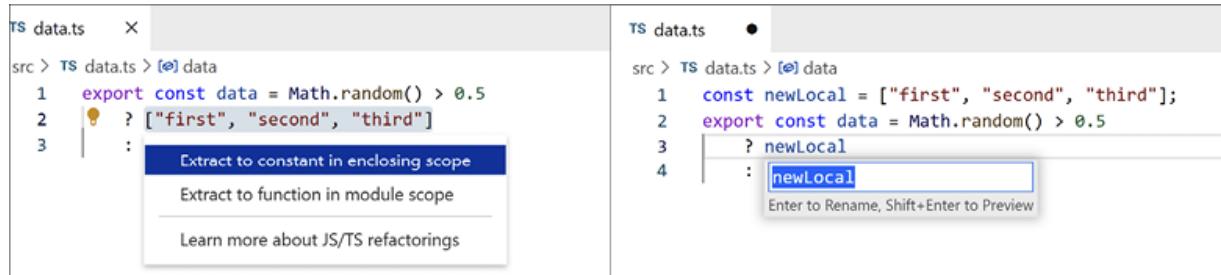


Figura 12.14: Esquerda: selecionando uma importação não usada e abrindo o menu de refatorações; direita: o arquivo após o TypeScript a excluir.

## Outras correções rápidas

Muitas mensagens de erro do TypeScript são para problemas no código que podem ser corrigidos rapidamente, como pequenos erros de digitação em palavras-chave ou nomes de variáveis. Outras correções rápidas normalmente usadas do TypeScript são:

- Declaração de uma propriedade ausente em uma classe ou interface.
- Correção de um nome de campo digitado erroneamente.
- Fornecimento de propriedades ausentes de uma variável declarada como um tipo.

Recomendo verificar a lista de correções rápidas sempre que você detectar uma mensagem de erro que ainda não tinha visto. Nunca se sabe que utilitários convenientes o TypeScript pode estar disponibilizando para resolver isso!

## Refatoração

O serviço de linguagem do TypeScript fornece muitas alterações de código úteis para diferentes estruturas de código. Algumas são tão simples, como mover linhas de código, enquanto outras são tão complexas, como criar novas funções automaticamente.

Quando você selecionar uma área de código, o VS Code exibirá um ícone

de lâmpada próximo à sua seleção. Clique nele para ver a lista de refatorações disponíveis.

Aqui temos um desenvolvedor extraindo um array literal inline para uma variável const (Figura 12.15).

The screenshot shows two side-by-side code editors. The left editor displays the following TypeScript code:

```
TS index.ts U •
TS index.ts
1 import { data } from "./data";
2 Remove import from './data'
3 Convert named imports to namespace import
4 Learn more about JS/TS refactorings
5
6 console.log(`The candles burn out for you;
7 I am free.
8 `);
9
10 );
11
```

A context menu is open over the array literal at line 1, with the option "Remove import from './data'" highlighted in blue. The right editor shows the result of the refactoring:

```
TS index.ts U •
TS index.ts
1
2
3
4
5
6 const data = [
7   "The candles burn out for you",
8   "I am free."
9 ];
10
```

Figura 12.15: Esquerda: selecionando um array literal e abrindo o menu de refatorações; direita: extraindo para uma variável constante.

## Trabalho efetivo com erros

Ler mensagens de erro e tomar medidas relacionadas a elas é um ato corriqueiro no trabalho com qualquer linguagem de programação. Todos os desenvolvedores, independentemente de sua proficiência na linguagem TypeScript, acionam várias mensagens de erro do compilador sempre que escrevem código TypeScript. Usar recursos do IDE para melhorar sua habilidade de trabalhar efetivamente com as mensagens de erro do compilador TypeScript o ajudará a tornar-se mais produtivo com a linguagem.

### Erros do serviço de linguagem

Normalmente, os editores geram qualquer erro relatado pelo serviço de linguagem do TypeScript, como sublinhados em vermelho abaixo do código problemático. A passagem do cursor sobre os caracteres sublinhados exibe uma caixa flutuante próxima a eles com o texto do erro (Figura 12.16).

O VS Code também exibe erros de qualquer arquivo aberto em uma aba Problems de sua seção Panels. O link View Problem no canto inferior esquerdo da caixa de passagem do cursor sobre o erro abre uma exibição inline da mensagem inserida após a linha do problema e antes de qualquer linha subsequente (Figura 12.17).

The screenshot shows the VS Code interface with the 'index.ts' file open. A red underline is under the identifier 'thisVariableDoesNotExist'. A tooltip appears over the error, displaying the message 'Cannot find name 'thisVariableDoesNotExist''. Below the tooltip, there are two buttons: 'View Problem' and 'No quick fixes available'.

Figura 12.16: Informações exibidas com a passagem do cursor sobre uma variável que não existe.

This screenshot is similar to Figura 12.16, but the 'View Problem' link is highlighted with a red arrow. The rest of the UI elements are identical.

Figura 12.17: Exibição inline do link View Problem para uma variável que não existe.

Quando existirem vários problemas no mesmo arquivo-fonte, suas exibições incluirão setas para cima e para baixo para você usar e se alternar entre eles. F8 e Shift + F8 funcionarão como atalhos para você avançar e retroceder, respectivamente, nessa lista de problemas (Figura 12.18).

This screenshot shows the 'index.ts' file with two errors. The first error is 'thisVariableDoesNotExist' at line 1. The second error is 'thisVariableAlsoDoesNotExist' at line 3. A red arrow points to the 'View Problem' link for the second error. At the bottom of the panel, there are navigation arrows for moving between problems, and a status bar indicating '2 of 2 problems'.

*Figura 12.18: Uma das duas exibições inline do link View Problem para variáveis que não existem.*

## Aba Problems

O VS Code tem uma aba Problems em seu painel que, como seu nome sugere, exibe qualquer problema que houver no espaço de trabalho (workspace). Isso inclui erros relatados pelo serviço de linguagem do TypeScript.

Este screenshot mostra uma aba Problems exibindo dois problemas em um arquivo TypeScript (Figura 12.19).

The screenshot shows the VS Code interface with the following details:

- Editor Area:** Shows a file named "index.ts" with two errors highlighted:
  - Line 1: `thisVariableDoesNotExist;`
  - Line 3: `thisVariableAlsoDoesNotExist;`
- Problems Panel:** An open panel titled "PROBLEMS" (with 2 items). It lists the two errors from the editor:
  - Line 1, Column 1: "Cannot find name 'thisVariableDoesNotExist'. ts(2304) [1, 1]"
  - Line 3, Column 1: "Cannot find name 'thisVariableAlsoDoesNotExist'. ts(2304) [3, 1]"
- Bottom Navigation:** Includes tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and a Filter input field.

*Figura 12.19: Aba Problems exibindo dois erros em um arquivo.*

Clicar em um erro dentro da aba Problems levará o cursor de texto para a linha e a coluna incorretas do arquivo.

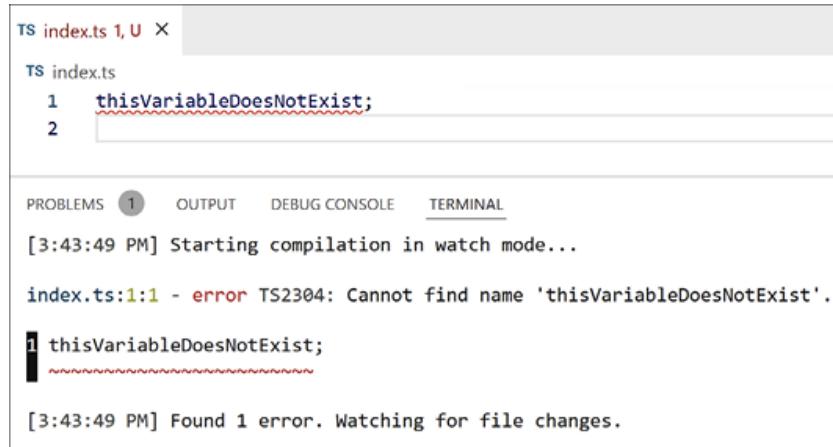
É bom ressaltar que o VS Code só listará problemas de arquivos que estiverem abertos. Se você quiser uma lista atualizada em tempo real com todos os problemas do compilador TypeScript, terá de executá-lo em um terminal.

## Execução de um compilador no terminal

Recomendo executar o compilador TypeScript no modo de observação (abordado no Capítulo 13, “Opções de configuração”) em um terminal ao trabalhar em um projeto TypeScript. Isso fornecerá uma lista atualizada em tempo real com todos os problemas – e não só os existentes em arquivos.

Para fazer isso no VS Code, abra o painel Terminal e execute `tsc -w` (ou `tsc`

`-b -w` se estiver usando referências de projeto, também abordadas no Capítulo 13, “Opções de configuração”). Você deve ver uma tela de terminal exibindo todos os problemas do TypeScript existentes em seu projeto, como neste screenshot (Figura 12.20).



The screenshot shows a terminal window in VS Code with the title bar "TS index.ts 1, U X". The terminal content is as follows:

```
TS index.ts
1 thisVariableDoesNotExist;
2

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL
[3:43:49 PM] Starting compilation in watch mode...
index.ts:1:1 - error TS2304: Cannot find name 'thisVariableDoesNotExist'.
1 thisVariableDoesNotExist;
2

[3:43:49 PM] Found 1 error. Watching for file changes.
```

Figura 12.20: Executando `tsc -w` em um terminal para relatar um problema em um arquivo.

Cmd (Mac) / Ctrl (Windows) + clicar em um nome de arquivo também levará o cursor de texto para a linha e a coluna incorretas do arquivo.



Alguns projetos usam configurações do arquivo `launch.json` do VC Code para executar um terminal com o compilador TypeScript no modo de observação. Consulte [code.visualstudio.com/docs/editor/tasks](https://code.visualstudio.com/docs/editor/tasks) para ver um material de referência completo sobre as tarefas do VS Code.

## Conhecimento dos tipos

Pode ocorrer de você ter de descobrir o tipo de algo que foi definido sem que o tipo ficasse evidente. Seja qual for o valor, você pode passar o cursor sobre seu nome para ver uma caixa flutuante exibindo seu tipo.

Este screenshot mostra a caixa flutuante de uma variável (Figura 12.21).

A screenshot of a code editor showing floating information boxes. The file is 'index.ts'. A box highlights the declaration 'const getData: () => string' at line 3. Another box highlights the import 'import { getData } from "./getData";' at line 1.

```
TS index.ts  x
src > TS index.ts
1 import { getData } from "./getData";
2
3 (alias) const getData: () => string
4 import getData
5 getData;
```

Figura 12.21: Informações flutuantes de uma variável.

Mantenha a tecla Ctrl pressionada enquanto desliza o cursor para também exibir onde o nome foi declarado.

Este screenshot mostra a caixa flutuante com Ctrl pressionada para a mesma variável de antes (Figura 12.22).

A screenshot of a code editor showing expanded floating information boxes. The file is 'index.ts'. A box highlights the declaration 'const getData: () => string' at line 3. Another box highlights the export 'export const getData = () => "Hello, world!";' at line 5. A third box highlights the import 'import { getData } from "./getData";' at line 1.

```
TS index.ts  x
src > TS index.ts
1 import { getData } from "./getData";
2
3 (alias) const getData: () => string
4 import getData
5 export const getData = () => "Hello, world!";
6 getData;
```

Figura 12.22: Informações flutuantes expandidas de uma variável.

As caixas de informações flutuantes também estão disponíveis para tipos, como para aliases de tipo. O screenshot a seguir mostra a passagem do cursor sobre um tipo `keyof typeof` para vermos a união de strings literais equivalente (Figura 12.23).

A screenshot of a code editor showing expanded floating information boxes. The file is 'types.ts'. A box highlights the type alias 'type FruitName = "apple" | "broccoli" | "cherry"' at line 6. Another box highlights the export 'export type FruitName = keyof typeof fruits;' at line 7.

```
TS types.ts  x
src > TS types.ts > [?] FruitName
1 const fruits = {
2   apple: 1,
3   broccoli: 2,
4   cherry: 3,
5 };
6 type FruitName = "apple" | "broccoli" | "cherry"
7 export type FruitName = keyof typeof fruits;
```

Figura 12.23: Informações flutuantes expandidas de um tipo.

Uma estratégia que achei útil para conhecer os componentes de tipos complexos seria criar um alias de tipo que representasse apenas um componente do tipo. Você então poderia passar o cursor sobre esse alias

de tipo para ver em que tipo ele resulta.

Usando o tipo `FruitsType` de antes como exemplo, sua parte `typeof fruits` poderia ser extraída para um tipo intermediário separado com uma refatoração. Assim, você poderia passar o cursor sobre esse tipo intermediário para ver informações de tipo (Figura 12.24).

The figure consists of two side-by-side screenshots of a code editor. The left screenshot shows a file named `TS types.ts` with the following code:src > TS types.ts > [e] FruitName
1 const fruits = {
2 apple: 1,
3 broccoli: 2,
4 cherry: 3,
5 };
6
7 export type FruitName = keyof typeof fruits;
A tooltip window is open over the line `7 export type FruitName = keyof typeof fruits;`, displaying the option "Extract to type alias". The right screenshot shows the same file after the refactoring, with the code modified to use a new type alias:src > TS types.ts > [e] NewType
1 const fruits = {
2 a: apple,
3 b: broccoli,
4 c: cherry,
5 };
6
7 type NewType = typeof fruits;
8
9 export type FruitName = keyof NewType;
The cursor is hovering over the line `7 type NewType = typeof fruits;`.

Figura 12.24: Esquerda: extraindo parte do tipo `FruitsType`; direita: passando o cursor sobre esse tipo extraído.

A estratégia de alias de tipo intermediário é particularmente útil para a depuração das operações de tipo abordadas no Capítulo 15, “Operações com tipos”.

## Resumo

Neste capítulo, você explorou o uso das integrações com o IDE no TypeScript para melhorar sua habilidade de criação de código TypeScript:

- Abertura de menus de contexto em tipos e valores para a listagem dos comandos disponíveis.
- Navegação no código para a busca de definições, referências e implementações.
- Automação da escrita do código com preenchimentos de nomes e importações automáticas.
- Outros códigos de ação incluindo renomeações e refatorações.
- Estratégias para a visualização e o conhecimento de erros do serviço de linguagem.
- Estratégias para o conhecimento de tipos.



Agora que você terminou de ler este capítulo, pratique o que aprendeu em <https://learningtypescript.com/using-ide-features>.

*O que IDEs apaixonados dizem um para o outro?  
“Você me completa!”<sup>2</sup>*

---

<sup>1</sup> N.T.: Original: *Programming with an / IDE the first time feels / like superpowers.*

<sup>2</sup> N.T.: Original: *What do IDEs in love say to each other? “You complete me!”*

## CAPÍTULO 13

# Opções de configuração

*Opções de compilador:  
Tipos, módulos e, que maravilha,  
o tsc da sua maneira.<sup>1</sup>*

O TypeScript é altamente configurável e foi criado para se adaptar a todos os padrões de uso comuns do JavaScript. Ele pode funcionar para projetos que vão de códigos de navegador legados aos ambientes de servidor mais modernos.

Grande parte da configurabilidade do TypeScript vem de sua riqueza de mais de 100 opções de configuração que podem ser fornecidas por meio de:

- Flags de linha de comando (CLI) passadas para o comando `tsc`.
- Arquivos de configuração “TSCConfig” do TypeScript.

Este capítulo não tem como objetivo ser uma referência completa de todas as opções de configuração do TypeScript. Em vez disso, sugiro tratá-lo como um passeio pelas opções mais comuns que você usará. Incluí apenas as que costumam ser mais úteis e são mais empregadas na maioria das configurações de projetos TypeScript. Consulte [aka.ms/tsc](https://aka.ms/tsc) para ver um material de referência completo sobre cada uma dessas opções, entre outras informações.

## Opções do comando `tsc`

No Capítulo 1, “Do JavaScript ao TypeScript”, você usou `tsc index.ts` para compilar um arquivo `index.ts`. O comando `tsc` pode receber a maioria das opções de configuração do TypeScript como flags `--`.

Por exemplo, para executar `tsc` em um arquivo `index.ts` sem gerar o arquivo (apenas executando a verificação de tipos), passe a flag `--noEmit`:

```
tsc index.ts --noEmit
```

Você pode executar `tsc --help` para obter uma lista das flags mais usadas da CLI (command line interface, interface de linha de comando). A lista completa das opções de configuração do `tsc` encontrada em `aka.ms/tsc` pode ser visualizada com `tsc --all`.

## Modo pretty

A CLI do `tsc` pode fazer exibições em um modo “pretty”: estilizadas com cores e espaçamento para torná-las mais fáceis de ler. Por padrão ela usará o modo pretty se detectar que o terminal da saída suporta texto em cores.

Aqui está um exemplo da aparência do comando `tsc` ao exibir dois type errors ocorridos em um arquivo (Figura 13.1).

```
~/learningtypescript$ tsc index.ts
index.ts:1:12 - error TS2322: Type 'string' is not assignable to type 'number'.
1 export let notNumeric: number = "Gotcha!";
~~~~~
index.ts:3:12 - error TS2322: Type 'number' is not assignable to type 'string'.
3 export let notString: string = 1337;
~~~~~
Found 2 errors in the same file, starting at: index.ts:1
```

Figura 13.1: O `tsc` relatando dois erros com os nomes de arquivo em azul, os números de linhas e colunas em amarelo, e com sublinhados em vermelho.

Se você preferir uma saída de CLI mais condensada e/ou sem tantas cores, pode fornecer explicitamente `--pretty false` para solicitar ao TypeScript que use um formato mais conciso e sem cores (Figura 13.2).

```
~/learningtypescript$ tsc index.ts --pretty false
index.ts(1,12): error TS2322: Type 'string' is not assignable to type 'number'.
index.ts(3,12): error TS2322: Type 'number' is not assignable to type 'string'.
```

Figura 13.2: O `tsc` relatando dois erros em texto simples.

## Modo de observação

Minha maneira favorita de usar a CLI do `tsc` é com seu modo `-w/--watch`. Em vez de ser encerrado após a conclusão, o modo de observação (watch mode) manterá o TypeScript sendo executado indefinidamente e atualizará continuamente seu terminal com uma lista em tempo real de todos os erros que ele detectar.

A execução com o modo de observação em um arquivo contendo dois erros é mostrada na Figura 13.3.

```
[8:48:40 AM] Starting compilation in watch mode...

index.ts:1:12 - error TS2322: Type 'string' is not assignable to type 'number'.
1 export let notNumeric: number = "Gotcha!";
~~~~~

index.ts:3:12 - error TS2322: Type 'number' is not assignable to type 'string'.
3 export let notString: string = 1337;
~~~~~

[8:48:41 AM] Found 2 errors. Watching for file changes.
```

Figura 13.3: O `tsc` relatando dois erros no modo de observação.

A Figura 13.4 mostra o `tsc` atualizando a saída no console para indicar que o arquivo foi alterado de maneira que corrige todos os erros.

```
[8:49:18 AM] File change detected. Starting incremental compilation...
[8:49:18 AM] Found 0 errors. Watching for file changes.
```

Figura 13.4: O `tsc` relatando ausência de erros no modo de observação.

O modo de observação é particularmente útil quando trabalhamos em alterações maiores, como no caso de refatorações feitas em muitos arquivos. Você pode usar os type errors do TypeScript como uma lista de verificação para ver o que ainda precisa ser limpo.

## Arquivos TSConfig

Em vez de sempre fornecermos todos os nomes de arquivo e opções de configuração para o `tsc`, a maioria das opções pode ser especificada em

um arquivo `tsconfig.json` (“TSConfig”) em um diretório.

A existência de um arquivo `tsconfig.json` indica que o diretório é a raiz de um projeto TypeScript. A execução do `tsc` em um diretório acionará a leitura de qualquer opção de configuração desse arquivo `tsconfig.json`.

Você também pode passar `-p/---project` para o `tsc` com o caminho de um diretório contendo um arquivo `tsconfig.json` ou qualquer arquivo para fazer o `tsc` usá-lo:

```
tsc -p path/to/tsconfig.json
```

É altamente recomendável que arquivos TSConfig sejam usados sempre que possível para projetos TypeScript. IDEs como o VS Code respeitarão sua configuração ao fornecer recursos do IntelliSense.

Consulte [aka.ms/tsconfig.json](https://aka.ms/tsconfig.json) para ver a lista completa de opções de configuração disponíveis nos arquivos TSConfig.



Se você não definir uma opção em seu arquivo `tsconfig.json`, não pense que sua definição padrão no TypeScript alterará e afetará as configurações de compilação de seu projeto. Isso quase nunca acontece, e mesmo se ocorresse, precisaria haver uma atualização do TypeScript em uma nova versão e o fato ganharia destaque nas notas de lançamento.

## **tsc --init**

A linha de comando do `tsc` inclui um comando `--init` para a criação de um novo arquivo `tsconfig.json`. Esse recém-criado arquivo TSConfig conterá um link para os documentos de configuração assim como a maioria das opções de configuração permitidas do TypeScript com comentários de uma linha descrevendo brevemente seu uso.

A execução deste comando:

```
tsc --init
```

gerará um arquivo `tsconfig.json` totalmente comentado:

```
{
  "compilerOptions": {
    /* Visit https://aka.ms/tsconfig.json to read more about this file */
    // ...
  }
}
```

```
    }  
}
```

Recomendo usar `tsc --init` na criação de seu arquivo de configuração em seus primeiros projetos TypeScript. Seus valores padrão são aplicáveis à maioria dos projetos e os comentários da documentação são úteis para o seu entendimento.

## CLI versus arquivos de configuração

Ao percorrer o arquivo TSConfig criado por `tsc --init`, você pode notar que as opções de configuração desse arquivo estão dentro de um objeto `"compilerOptions"`. A maioria das opções disponíveis tanto na CLI quanto nos arquivos TSConfig se encaixa em uma entre duas categorias:

### *De compilador*

Como cada arquivo incluído será compilado e/ou terá seu tipo verificado pelo TypeScript.

### *De arquivo*

Em que arquivos o TypeScript será ou não será executado.

Outras configurações sobre as quais falaremos após essas duas categorias, como as referências de projeto, geralmente só estão disponíveis em arquivos TSConfig.



Quando uma configuração é fornecida para a CLI do `tsc`, por exemplo, uma alteração pontual para uma build de integração contínua ou de produto, geralmente ela sobrescreve qualquer valor especificado em um arquivo TSConfig. Já que os IDEs costumam fazer leituras no arquivo `tsconfig.json` em um diretório para acessar configurações do TypeScript, é recomendável inserir a maioria das opções de configuração em um arquivo `tsconfig.json`.

## Inclusões de arquivo

Por padrão, o `tsc` será executado em todos os arquivos `.ts` não ocultos (aqueles cujos nomes não comecem com um `.`) do diretório atual e de qualquer diretório filho, ignorando diretórios ocultos e diretórios

chamados *node\_modules*. As configurações do TypeScript podem alterar essa lista de arquivos nos quais ocorrerá a execução.

## include

A maneira mais comum de incluir arquivos é com a propriedade de nível superior "include" em um arquivo *tsconfig.json*. Ela nos permite usar um array de strings para descrever que diretórios e/ou arquivos devem ser incluídos na compilação do TypeScript.

Por exemplo, este arquivo de configuração inclui recursivamente todos os arquivos-fonte TypeScript em um diretório *src/* relativo a *tsconfig.json*:

```
{
  "include": ["src"]
}
```

Curingas globais (glob wildcards) são permitidos em strings *include* para a obtenção de um controle mais granular dos arquivos a serem incluídos:

- \* encontra zero ou mais caracteres (excluindo separadores de diretório);
- ? encontra qualquer caractere individual (excluindo separadores de diretório);
- \*\*/ encontra qualquer diretório aninhado em qualquer nível.

O arquivo de configuração a seguir só permite arquivos *.d.ts* aninhados em um diretório *typings/* e arquivos *src/* com pelo menos dois caracteres em seu nome antes de uma extensão:

```
{
  "include": [
    "typings/**/*.d.ts",
    "src/**/*??.*"
  ]
}
```

Para a maioria dos projetos, uma opção de compilador *include* simples como `["src"]` geralmente é suficiente.

## exclude

A lista de arquivos de *include* para um projeto pode incluir arquivos não

destinados a serem compilados pelo TypeScript. O TypeScript permite que um arquivo `TSConfig` omita caminhos de `include` pela sua especificação em uma propriedade de nível superior `"exclude"`. Como `include`, ela permite usar um array de strings para a descrição de que diretórios e/ou arquivos devem ser excluídos da compilação do TypeScript.

A configuração a seguir inclui todos os arquivos de `src/`, exceto os existentes dentro de qualquer diretório `external/` aninhado e de um diretório `node_modules`:

```
{  
  "exclude": ["**/external", "node_modules"],  
  "include": ["src"]  
}
```

Por padrão, `exclude` contém `["node_modules", "bower_components", "jspm_packages"]` para evitar a execução do compilador TypeScript em arquivos de biblioteca compilados de terceiros.



Se você estiver escrevendo sua própria lista `exclude`, não precisará reinserir `"bower_components"` ou `"jspm_packages"`. A maioria dos projetos JavaScript que instala módulos do node em uma pasta dentro do projeto só faz a instalação em `"node_modules"`.

Lembre-se, `exclude` só atua na remoção de arquivos da lista inicial de `include`. O TypeScript será executado em qualquer arquivo importado por qualquer arquivo incluído, mesmo se o arquivo importado estiver listado explicitamente em `exclude`.

## Extensões alternativas

Por padrão, o TypeScript consegue ler qualquer arquivo cuja extensão seja `.ts`. No entanto, alguns projetos requerem a leitura de arquivos com extensões diferentes, como módulos JSON ou a sintaxe JSX de bibliotecas de UI como a do React.

### Sintaxe JSX

Geralmente, uma sintaxe JSX, como `<Component />`, é usada em bibliotecas

de UI como as do Preact e React. Tecnicamente, a sintaxe JSX não é JavaScript. Como as definições de tipos do TypeScript, ela é uma extensão da sintaxe JavaScript que é compilada para JavaScript comum:

```
const MyComponent = () => {
  // Equivalent to:
  //   return React.createElement("div", null, "Hello, world!");
  return <div>Hello, world!</div>;
};
```

Para usar a sintaxe JSX em um arquivo, você precisa fazer duas coisas:

- Ativar a opção de compilador "jsx" em suas opções de configuração.
- Nomear esse arquivo com uma extensão .tsx.

## jsx

O valor usado para a opção de compilador "jsx" determina como o TypeScript emitirá o código JavaScript para arquivos .tsx. Geralmente, os projetos usam um destes três valores (Tabela 13.1):

*Tabela 13.1: Entradas e saídas da opção de compilador JSX*

Valor	Código de entrada	Código de saída	Extensão do arquivo de saída
"preserve"	<div />	<div />	.jsx
"react"	<div />	React.createElement("div")	.js
"react-native"	<div />	<div />	.js

Os valores de jsx podem ser fornecidos para a CLI do tsc e/ou em um arquivo TSConfig.

```
tsc --jsx preserve
{
  "compilerOptions": {
    "jsx": "preserve"
  }
}
```

Se você não estiver usando diretamente o transpilador interno do TypeScript, que é o que ocorre quando transpilamos o código com uma ferramenta separada, como o Babel, provavelmente poderá utilizar qualquer um dos valores permitidos para "jsx". A maioria das aplicações web baseadas em frameworks modernos, como o Next.js ou o Remix,

manipula a sintaxe de configuração e compilação do React. Se você estiver usando um desses frameworks não deve precisar configurar diretamente o transpilador interno do TypeScript.

## Arrow functions genéricas em arquivos .tsx

No Capítulo 10, “Genéricos”, mencionei que a sintaxe das arrow functions genéricas entram em conflito com a sintaxe JSX. Tentar fornecer um argumento de tipo `<T>` para uma arrow function em um arquivo `.tsx` resultará em um erro de sintaxe por não haver uma tag de fechamento para o elemento `T` de abertura:

```
const identity = <T>(input: T) => input;  
//           ~~~  
// Error: JSX element 'T' has no corresponding closing tag.
```

Para resolver essa ambiguidade de sintaxe, você pode adicionar uma restrição `= unknown` ao argumento de tipo. Por padrão, os argumentos de tipo têm o tipo `unknown`, logo, isso não alterará o comportamento do código. Apenas indicará para o TypeScript que ele está lendo um argumento de tipo, e não um elemento JSX:

```
const identity = <T = unknown>(input: T) => input; // Ok
```

## resolveJsonModule

O TypeScript permitirá a leitura de arquivos `.json` se a opção de compilador `resolveJsonModule` for configurada com `true`. Quando isso ocorrer, arquivos `.json` poderão ser importados como se fossem arquivos `.ts` exportando um objeto. O TypeScript inferirá o tipo desse objeto como se ele fosse uma variável `const`.

Para arquivos JSON contendo um objeto, importações de desestruturação podem ser usadas. Este par de arquivos define uma string `"activist"` em um arquivo `activist.json` e a importa para um arquivo `usesActivist.ts`:

```
// activist.json  
{  
  "activist": "Mary Astell"  
}  
// usesActivist.ts  
import { activist } from "./activist.json";
```

```
// Logs: "Mary Astell"
console.log(activist);
```

Importações padrão também podem ser usadas se a opção de compilador `esModuleInterop` – abordada posteriormente neste capítulo – estiver ativada:

```
// useActivist.ts
import data from "./activist.json";
```

Para arquivos JSON contendo outros tipos literais, como arrays ou números, você terá de usar a sintaxe de importação `* as`. O par de arquivos a seguir define um array de strings em um arquivo `activists.json` e depois o importa para um arquivo `useActivists.ts`:

```
// activists.json
[
  "Ida B. Wells",
  "Sojourner Truth",
  "Tawakkul Karmān"
]
// useActivists.ts
import * as activists from "./activists.json";

// Logs: "3 activists"
console.log(` ${activists.length} activists`);
```

## Emissão

Embora o surgimento de ferramentas de compilador dedicadas como o Babel tenha reduzido a atuação do TypeScript em alguns projetos apenas à verificação de tipos, vários projetos ainda dependem do TypeScript para a compilação da sintaxe TypeScript para JavaScript. É muito útil os projetos poderem receber `typescript` como única dependência e usar seu comando `tsc` para emitir o JavaScript equivalente.

### `outDir`

Por padrão, o TypeScript gera arquivos de saída junto aos arquivos-fonte correspondentes. Por exemplo, a execução do `tsc` em um diretório contendo `fruits/apple.ts` e `vegetables/zucchini.ts` resultaria nos arquivos de saída `fruits/apple.js` e `vegetables/zucchini.js`:

```
fruits/
  apple.js
  apple.ts
vegetables/
  zucchini.js
  zucchini.ts
```

Ocasionalmente pode ser preferível inserir os arquivos de saída em uma pasta diferente. Muitos projetos Node, por exemplo, inserem saídas transformadas em um diretório *dist* ou *lib*.

A opção de compilador `--outDir` do TypeScript permite especificar um diretório raiz diferente para as saídas. Os arquivos de saída são mantidos na mesma estrutura de diretório relativa dos arquivos de entrada.

Por exemplo, a execução de `tsc --outDir dist` no diretório anterior inseriria as saídas dentro de uma pasta *dist*:

```
dist/
  fruits/
    apple.js
  vegetables/
    zucchini.js
fruits/
  apple.ts
vegetables/
  zucchini.ts
```

O TypeScript calcula o diretório raiz para inserção dos arquivos de saída encontrando o subcaminho comum mais longo de todos os arquivos de entrada (excluindo os arquivos de declaração *.d.ts*). Isso significa que em projetos que inserirem todos os arquivos-fonte de entrada em um único diretório, esse diretório será tratado como raiz.

Por exemplo, se o caso acima inserisse todas as entradas em um diretório *src*/ e fosse compilado com `--outDir lib`, *lib/fruits/apple.js* seria criado, e não *lib/src/fruits/apple.js*:

```
lib/
  fruits/
    apple.js
  vegetables/
    zucchini.js
src/
```

```
fruits/  
  apple.ts  
vegetables/  
  zucchini.ts
```

Existe uma opção de compilador `rootDir` para a especificação explícita desse diretório raiz, mas raramente ela é necessária ou usada com valores diferentes de `.` ou `src`.

## target

O TypeScript pode produzir uma saída JavaScript para ser executada em ambientes tão antigos quanto o ES3 (aproximadamente de 1999!). A maioria dos ambientes pode suportar recursos de sintaxe de versões muito mais novas do JavaScript.

O TypeScript inclui uma opção de compilador `target` para a especificação de o quanto será necessário retroceder no suporte à sintaxe para o código JavaScript ser transpilado. Embora por padrão `target` seja igual a `"es3"` por razões de compatibilidade regressiva quando não especificado e `tsc --init` seja igual a `"es2016"`, é aconselhável que você use a sintaxe JavaScript mais recente possível existente para sua plataforma de destino. O suporte a recursos mais novos do JavaScript em ambientes mais antigos demanda a criação de mais código JavaScript, o que gera tamanhos de arquivo um pouco maiores e um desempenho de runtime um pouco pior.

 A partir de 2022, todas as versões lançadas no ano anterior de navegadores que atendem a mais de 0,1% dos usuários no mundo inteiro suportam pelo menos todo o ECMAScript 2019 e quase todo o ECMAScript 2020-2021, enquanto as versões LTS (long-term support, de suporte de longo prazo) do Node.js suportam todo o ECMAScript 2021. Não há por que não usar um `target` tão alto como `"es2019"`.

Por exemplo, veja este arquivo-fonte contendo `consts` do ES2015 e o operador de coalescência nula (nullish coalescing) `??` do ES2020:

```
function defaultNameAndLog(nameMaybe: string | undefined) {  
  const name = nameMaybe ?? "anonymous";  
  console.log("From", nameMaybe, "to", name);
```

```
    return name;
}
```

Com `tsc --target es2020`, ou com um target mais recente, tanto `const` quanto `??` são recursos de sintaxe suportados, logo, o TypeScript só precisaria remover `: string | undefined` desse trecho de código:

```
function defaultNameAndLog(nameMaybe) {
  const name = nameMaybe ?? "anonymous";
  console.log("From", nameMaybe, "to", name);
  return name;
}
```

Com `tsc --target es2015` a `es2019`, o açúcar sintático<sup>2</sup> `??` seria compilado para seu equivalente de versões mais antigas do JavaScript:

```
function defaultNameAndLog(nameMaybe) {
  const name = nameMaybe !== null && nameMaybe !== void 0
    ? nameMaybe
    : "anonymous";
  console.log("From", nameMaybe, "to", name);
  return name;
}
```

Com `tsc --target es3` ou `es5`, `const` também precisaria ser convertida para seu equivalente `var`:

```
function defaultNameAndLog(nameMaybe) {
  var name = nameMaybe !== null && nameMaybe !== void 0
    ? nameMaybe
    : "anonymous";
  console.log("From", nameMaybe, "to", name);
  return name;
}
```

Especificar a opção de compilador `target` com um valor que coincida com o ambiente mais antigo no qual seu código é executado assegurará que o código seja emitido com uma sintaxe moderna e concisa que possa ser executada sem erros de sintaxe.

## Emissão de declarações

O Capítulo 11, “Arquivos de declaração”, abordou como arquivos de declaração `.d.ts` podem ser distribuídos em um pacote para indicar os tipos do código para os consumidores. A maioria dos pacotes usa a opção

de compilador `declaration` do TypeScript para emitir arquivos de saída `.d.ts` a partir de arquivos-fonte:

```
tsc --declaration
{
  "compilerOptions": {
    "declaration": true
  }
}
```

Os arquivos de saída `.d.ts` são emitidos seguindo as mesmas regras de saída dos arquivos `.js`, inclusive respeitando `outDir`.

Por exemplo, executar `tsc --declaration` em um diretório contendo `fruits/apple.ts` e `vegetables/zucchini.ts` resultaria nos arquivos de declaração de saída `fruits/apple.d.ts` e `vegetables/zucchini.d.ts` junto com arquivos de saída `.js`:

```
fruits/
  apple.d.ts
  apple.js
  apple.ts
vegetables/
  zucchini.d.ts
  zucchini.js
  zucchini.ts
```

## **emitDeclarationOnly**

Existe uma opção de compilador `emitDeclarationOnly`, como um acréscimo especializado à opção de compilador `declaration`, que direciona o TypeScript a só emitir arquivos de declaração: nenhum arquivo `.js/.jsx` é emitido. Isso é útil para projetos que usem uma ferramenta externa para gerar saída JavaScript, mas também querem utilizar o TypeScript para gerar arquivos de definição de saída:

```
tsc --emitDeclarationOnly
{
  "compilerOptions": {
    "emitDeclarationOnly": true
  }
}
```

Se `emitDeclarationOnly` for ativada, `declaration` ou a opção de compilador

`composite` abordada posteriormente neste capítulo também deve ser ativada.

Por exemplo, executar `tsc --declaration --emitDeclarationOnly` em um diretório contendo `fruits/apple.ts` e `vegetables/zucchini.ts` resultaria nos arquivos de declaração de saída `fruits/apple.d.ts` e `vegetables/zucchini.d.ts` sem arquivos de saída `.js`:

```
fruits/
  apple.d.ts
  apple.ts
vegetables/
  zucchini.d.ts
  zucchini.ts
```

## Mapas de fonte

Os mapas de fonte (source maps) são descrições de quanto o conteúdo dos arquivos de saída coincide com os arquivos-fonte originais. Eles permitem que ferramentas de desenvolvedor, como os depuradores, exibam o código-fonte original quando navegamos pelo arquivo de saída. São particularmente úteis para depuradores visuais, como os usados em IDEs e nas ferramentas de desenvolvedor dos navegadores para permitir a visualização do conteúdo do arquivo-fonte original no momento da depuração. O TypeScript inclui a capacidade de exibição de mapas de fonte junto com os arquivos de saída.

### `sourceMap`

A opção de compilador `sourceMap` do TypeScript permite exibir mapas de fonte `.js.map` ou `.jsx.map` junto com arquivos de saída `.js` ou `.jsx`. Caso contrário, os arquivos de mapa de fonte recebem o mesmo nome do arquivo de saída JavaScript correspondente e são inseridos no mesmo diretório.

Por exemplo, executar `tsc --sourceMap` em um diretório contendo `fruits/apple.ts` e `vegetables/zucchini.ts` resultaria nos arquivos de mapa de fonte de saída `fruits/apple.js.map` e `vegetables/zucchini.js.map` junto com arquivos `.js`:

```
fruits/
```

```
apple.js
apple.js.map
apple.ts
vegetables/
  zucchini.js
  zucchini.js.map
  zucchini.ts
```

## declarationMap

O TypeScript também pode gerar mapas de fonte para arquivos de declaração `.d.ts`. Sua opção de compilador `declarationMap` o direciona para gerar um mapa de fonte `.d.ts.map` para cada arquivo `.d.ts` mapeado para o arquivo-fonte original. Os mapas de declaração permitem que IDEs, como o VS Code, acessem o arquivo-fonte original quando do uso de recursos do editor como Go to Definition.



`declarationMap` é particularmente útil no trabalho com referências de projeto, que serão abordadas perto do fim deste capítulo.

Por exemplo, executar `tsc --declaration --declarationMap` em um diretório contendo `fruits/apple.ts` e `vegetables/zucchini.ts` resultaria nos arquivos de mapa de fonte de declaração de saída `fruits/apple.d.ts.map` e `vegetables/zucchini.d.ts.map` junto com arquivos de saída `.d.ts` e `.js`:

```
fruits/
  apple.d.ts
  apple.d.ts.map
  apple.js
  apple.ts
vegetables/
  zucchini.d.ts
  zucchini.d.ts.map
  zucchini.js
  zucchini.ts
```

## noEmit

Para projetos que dependam de outras ferramentas para compilar arquivos-fonte para saída JavaScript, o TypeScript pode ser solicitado a não emitir arquivos. A ativação da opção de compilador `noEmit` instrui o

TypeScript a agir apenas como um verificador de tipos.

Executar `tsc --noEmit` em qualquer um dos exemplos anteriores resultaria na criação de nenhum arquivo novo. O TypeScript só relataria qualquer erro de sintaxe ou type error que o encontrasse.

## Verificação de tipos

A maioria das opções de configuração do TypeScript controla o verificador de tipos. Você pode configurá-lo para ser gentil e compreensivo, para só emitir alertas de verificação de tipos quando souber com certeza da existência de um erro, ou para ser severo e rigoroso, demandando que quase todo o código seja bem tipado.

### lib

Para começar, podemos configurar quais APIs globais o TypeScript assumirá como presentes no ambiente de runtime com a opção de compilador `lib`. Ela recebe um array de strings que, por padrão, usará sua opção de compilador `target`, assim como `dom` para indicar a inclusão de tipos de navegador.

Quase sempre, a única razão para personalizarmos `lib` seria para remover a inclusão de `dom` para um projeto que não seja executado no navegador:

```
tsc --lib es2020
{
  "compilerOptions": {
    "lib": ["es2020"]
  }
}
```

Alternativamente, para um projeto que use polyfills<sup>3</sup> para suportar APIs mais novas do JavaScript, `lib` pode incluir `dom` e qualquer versão do ECMAScript:

```
tsc --lib dom,es2021
{
  "compilerOptions": {
    "lib": ["dom", "es2021"]
  }
}
```

Tome cuidado para não modificar `lib` sem fornecer todos os polyfills de runtime corretos. Um projeto com `lib` configurada com "es2021" sendo executado em uma plataforma que só suporte até o ES2020 pode não ter erros de verificação de tipos, mas apresentar erros de runtime ao tentar usar APIs definidas no ES2021 ou posterior, como `String.replaceAll`:

```
const value = "a b c";  
  
value.replaceAll(" ", " ", " ");  
// Uncaught TypeError: value.replaceAll is not a function
```



Considere a opção de compilador `lib` como indicando que APIs internas da linguagem estão disponíveis, enquanto a opção de compilador `target` indica que recursos de sintaxe existem..

## skipLibCheck

O TypeScript fornece uma opção de compilador `skipLibCheck` que solicita que não seja executada a verificação de tipos em arquivos de declaração não incluídos explicitamente no código-fonte. Isso pode ser útil para aplicações que usem muitas dependências que possam precisar de definições de bibliotecas compartilhadas diferentes e conflitantes:

```
tsc --skipLibCheck  
{  
  "compilerOptions": {  
    "skipLibCheck": true  
  }  
}
```

`skipLibCheck` melhora o desempenho do TypeScript permitindo que ele não execute parte da verificação de tipos. Logo, geralmente é uma boa ideia ativá-la na maioria dos projetos.

## Modo estrito

A maioria das opções de compilador relacionadas à verificação de tipos do TypeScript estão agrupadas no que a linguagem chama de *modo estrito*. Por padrão, todas as opções de compilador referentes à rigidez são iguais à `false` e, quando ativadas, levam o verificador de tipos a usar algumas verificações adicionais.

Abordarei as opções mais usadas do modo estrito em ordem alfabética posteriormente neste capítulo. Entre elas, `noImplicitAny` e `strictNullChecks` são particularmente úteis e impactantes para impor que o código seja type-safe.

Você pode ativar todas as verificações do modo estrito habilitando a opção de compilador `strict`:

```
tsc --strict
{
  "compilerOptions": {
    "strict": true
  }
}
```

Se você quiser ativar todas as verificações do modo estrito, exceto algumas, pode ativar `strict` e desativar explicitamente certas verificações. Por exemplo, esta configuração ativa todas as verificações do modo estrito, **exceto** `noImplicitAny`:

```
tsc --strict --noImplicitAny false
{
  "compilerOptions": {
    "noImplicitAny": false,
    "strict": true
  }
}
```



Versões futuras do TypeScript podem introduzir novas opções de compilador para a verificação de tipos de `strict`. Logo, usar `strict` pode gerar novos alertas de verificação de tipos quando você atualizar as versões do TypeScript. No entanto, sempre será possível optar por não usar certas configurações em seu arquivo `TSConfig`.

## `noImplicitAny`

Se o TypeScript não conseguir inferir o tipo de um parâmetro ou propriedade, ele assumirá que se trata do tipo `any`. Geralmente é prática recomendada não permitir tipos `any` implícitos no código já que o tipo `any` não passa por grande parte da verificação de tipos do TypeScript.

A opção de compilador `noImplicitAny` instrui o TypeScript a emitir um

alerta de verificação de tipos se ele precisar recorrer a um `any` implícito. Por exemplo, escrever o parâmetro de função a seguir sem uma declaração de tipo causaria um type error de `noImplicitAny`:

```
const logMessage = (message) => {
  // ~~~~~
  // Error: Parameter 'message' implicitly has an 'any' type.
  console.log(`Message: ${message}`);
};
```

Quase sempre um alerta da opção `noImplicitAny` pode ser resolvido com a inserção de uma anotação de tipo no local que o gerou:

```
const logMessage = (message: string) => { // Ok
  console.log(`Message: ${message}`);
}
```

Ou, no caso de parâmetros de função, com a inclusão da função pai em um local que indique o tipo da função:

```
type LogsMessage = (message: string) => void;

const logMessage: LogsMessage = (message) => { // Ok
  console.log(`Message: ${message}`);
}
```

 `noImplicitAny` é uma flag excelente para garantir a segurança de tipos (type safety) em um projeto. É altamente recomendável tentar ativá-la em projetos totalmente escritos em TypeScript. No entanto, se um projeto ainda estiver fazendo a transição de JavaScript para TypeScript, pode ser mais fácil terminar antes a conversão de todos os arquivos para TypeScript.

## **strictBindCallApply**

Quando o TypeScript foi lançado, ele não tinha recursos de tipagem suficientemente sofisticados para representar os utilitários de função internos `Function.apply`, `Function.bind` ou `Function.call`. Por padrão, essas funções tinham de receber `any` para sua lista de argumentos. Isso não é muito type safe!

Como exemplo, sem `strictBindCallApply` todas as variações de `getLength` a seguir incluem `any` em seus tipos:

```

function getLength(text: string, trim?: boolean) {
    return trim ? text.trim().length : text.length;
}

// Function type: (thisArg: Function, argArray?: any) => any
getLength.apply;

// Returned type: any
getLength.bind(undefined, "abc123");

// Returned type: any
getLength.call(undefined, "abc123", true);

```

Agora que os recursos de tipagem do TypeScript são suficientemente poderosos para representar os argumentos rest genéricos dessas funções, o TypeScript nos permite optar por usar tipos mais restritivos para elas.

A ativação de `strictBindCallApply` permite usar tipos muito mais precisos para as variações de `getLength`:

```

function getLength(text: string, trim?: boolean) {
    return trim ? text.trim().length : text;
}

// Function type:
// (thisArg: typeof getLength, args: [text: string, trim?: boolean]) => number;
getLength.apply;

// Returned type: (trim?: boolean) => number
getLength.bind(undefined, "abc123");

// Returned type: number
getLength.call(undefined, "abc123", true);

```

A prática recomendada no TypeScript é a ativação de `strictBindCallApply`. Sua verificação de tipos melhorada para utilitários de função internos ajuda a aperfeiçoar a segurança de tipos para os projetos que os utilizarem.

## **strictFunctionTypes**

A opção de compilador `strictFunctionTypes` faz os tipos dos parâmetros de função serem verificados com mais rigidez. Um tipo de função não será

mais considerado atribuível a outro tipo de função se os seus parâmetros forem subtipos dos parâmetros desse outro tipo.

Como um exemplo concreto, a função `checkOnNumber` a seguir aceita uma função que deveria ser de tipo `number | string`, mas recebe a função `stringContainsA` que espera somente o tipo `string`. A verificação de tipos padrão do TypeScript permitiria isso – e o programa quebraria por tentar chamar `.match()` em um `number`:

```
function checkOnNumber(containsA: (input: number | string) => boolean) {
    return containsA(1337);
}

function stringContainsA(input: string) {
    return !!input.match(/a/i);
}

checkOnNumber(stringContainsA);
```

Com `strictFunctionTypes`, `checkOnNumber(stringContainsA)` causaria um erro de verificação de tipo:

```
// Argument of type '(input: string) => boolean' is not assignable
// to parameter of type '(input: string | number) => boolean'.
//   Types of parameters 'input' and 'input' are incompatible.
//     Type 'string | number' is not assignable to type 'string'.
//       Type 'number' is not assignable to type 'string'.
checkOnNumber(stringContainsA);
```



Em termos técnicos, os parâmetros de função passam de *bivariante* para *contravariante*. Você pode ler mais sobre a diferença nas *notas de lançamento do TypeScript 2.6*.

## strictNullChecks

No Capítulo 3, “Unões e literais”, discuti o erro de um bilhão de dólares das linguagens: permitir que tipos vazios como `null` e `undefined` sejam atribuíveis a tipos não vazios. A desativação da flag `strictNullChecks` do TypeScript adiciona `null | undefined` a praticamente todos os tipos do código, permitindo, assim, que qualquer variável receba `null` ou `undefined`.

Este trecho de código causaria um type error por atribuir `null` a um valor

que só é tipado como `string` quando `strictNullChecks` é ativada:

```
let value: string;

value = "abc123"; // Always ok

value = null;
// With strictNullChecks enabled:
// Error: Type 'null' is not assignable to type 'string'.
```

A prática recomendada no TypeScript é a ativação de `strictNullChecks`. Isso ajuda a evitar quebras no código e elimina o erro de um bilhão de dólares.

Consulte o Capítulo 3, “Unões e literais”, para ver mais detalhes.

## **strictPropertyInitialization**

No Capítulo 8, “Classes”, abordei a verificação de inicialização estrita nas classes: certificar-se de que cada propriedade seja atribuída definitivamente no construtor da classe. A flag `strictPropertyInitialization` do TypeScript faz uma mensagem de type error ser emitida em propriedades de classe que não tenham um inicializador e não sejam atribuídas definitivamente no construtor.

Geralmente, a prática recomendada no TypeScript é a ativação de `strictPropertyInitialization`. Isso ajuda a evitar quebras no código por erros na lógica de inicialização de classes.

Consulte o Capítulo 8, “Classes”, para ver mais detalhes.

## **useUnknownInCatchVariables**

A manipulação de erros é um conceito inherentemente inseguro em qualquer linguagem. Teoricamente, qualquer função pode lançar qualquer número de erros causados por casos extremos como ler propriedades em instruções `throw undefined` ou escritas pelo usuário. Na verdade, não há garantias nem mesmo de que um erro lançado será uma instância da classe `Error`: o código pode sempre lançar “algo-diferente”.

Como resultado, o comportamento padrão do TypeScript para erros é dar-lhes o tipo `any`, já que eles podem ser qualquer coisa. Isso proporciona

flexibilidade na manipulação de erros ao custo de por padrão dependermos do tipo não muito type safe `any`.

No trecho de código a seguir, `error` é tipado como `any` porque não há como o TypeScript saber que erros podem ser lançados por `someExternalFunction()`:

```
try {
    someExternalFunction();
} catch (error) {
    error; // Default type: any
}
```

Como na maioria dos usos de `any`, seria tecnicamente mais seguro – ao custo de precisarmos frequentemente de asserções ou estreitamentos de tipos explícitos – tratar os erros como `unknown`. Os erros de cláusulas `catch` podem ser anotados como tendo os tipos `any` ou `unknown`.

A correção feita neste trecho de código adiciona um `: unknown` explícito a `error` para alterá-lo para o tipo `unknown`:

```
try {
    someExternalFunction();
} catch (error: unknown) {
    error; // Type: unknown
}
```

A flag de área estrita `useUnknownInCatchVariables` altera o tipo de erro padrão da cláusula `catch` do TypeScript para `unknown`. Com `useUnknownInCatchVariables` ativada, os dois trechos de código teriam o tipo de `error` configurado com `unknown`.

A prática recomendada no TypeScript costuma ser a ativação de `useUnknownInCatchVariables`, já que nem sempre é seguro assumir que os erros serão de um tipo específico.

## Módulos

Os diversos sistemas do JavaScript para a exportação e a importação do conteúdo de módulos – AMD, CommonJS, ECMAScript e assim por diante – representam um dos sistemas de módulos mais complicados de qualquer linguagem de programação moderna. O JavaScript é

relativamente incomum porque a maneira como os arquivos importam o conteúdo uns dos outros com frequência é controlada por frameworks escritos pelo usuário como o Webpack. O TypeScript faz o melhor que pode para fornecer opções que representem configurações de módulo mais sensatas para o ambiente do usuário.

A maioria dos novos projetos TypeScript é escrita com a sintaxe de módulos padronizada do ECMAScript. Como uma recapitulação, veja como os módulos ECMAScript importam um valor (`value`) de outro módulo ("`my-example-lib`") e exportam seu próprio valor (`logValue`):

```
import { value } from "my-example-lib";  
  
export const logValue = () => console.log(value);
```

## module

O TypeScript fornece uma opção de compilador `module` que define qual sistema de módulo o código transpilado usará. Na criação de código-fonte com módulos ECMAScript, o TypeScript pode transpilar as instruções `export` e `import` para um sistema de módulo diferente de acordo com o valor de `module`.

Por exemplo, você pode definir que um projeto escrito em ECMAScript seja exibido como módulos CommonJS na linha de comando:

```
tsc --module commonjs
```

ou em um arquivo TSConfig:

```
{  
  "compilerOptions": {  
    "module": "commonjs"  
  }  
}
```

O trecho de código anterior seria exibido com uma aparência como esta:

```
const my_example_lib = require("my-example-lib");  
exports.logValue = () => console.log(my_example_lib.value);
```

Se sua opção de compilador `target` for "`es3`" ou "`es5`", o valor padrão de `module` será "`commonjs`". Caso contrário, por padrão `module` será "`es2015`" para especificar a exibição de módulos ECMAScript.

## moduleResolution

*Resolução de módulos* é o processo pelo qual o caminho importado em uma importação é mapeado para um módulo. O TypeScript fornece uma opção `moduleResolution` que você pode usar para especificar a lógica desse processo. Você pode fornecer para ela uma entre duas estratégias lógicas:

- `node`: comportamento usado por resolvedores de CommonJS como o tradicional Node.js;
- `nodenext`: alinhamento com o comportamento especificado para módulos ECMAScript.

As duas estratégias são semelhantes. A maioria dos projetos poderia usar qualquer uma das duas sem que fosse percebida alguma diferença. Você pode ler mais sobre as complexidades existentes por trás da resolução de módulos em <https://www.typescriptlang.org/docs/handbook/module-resolution.html>.



`moduleResolution` não altera como o TypeScript emite o código. Ela só é usada para descrever o ambiente de runtime no qual o código deve ser executado.

Tanto o trecho de CLI quanto o de arquivo JSON a seguir funcionariam para especificar a opção de compilador `moduleResolution`:

```
tsc --moduleResolution nodenext
{
  "compilerOptions": {
    "moduleResolution": "nodenext"
  }
}
```



Por razões de compatibilidade regressiva, o TypeScript mantém o valor padrão de `moduleResolution` como o valor `classic` que foi usado para projetos anos atrás. Certamente você não vai querer a estratégia `classic` em nenhum projeto moderno.

## Interoperabilidade com o CommonJS

No trabalho com módulos JavaScript, existe uma diferença entre a

exportação “padrão” de um módulo e sua saída em “namespace”. A exportação *padrão* de um módulo é a propriedade `.default` do objeto exportado. A exportação do *namespace* de um módulo é o próprio objeto exportado.

A Tabela 13.2 recapitula a diferença entre as exportações e importações padrão e de namespace.

*Tabela 13.2: Formas de exportação e importação de módulos CommonJS e ECMAScript*

Área de sintaxe	CommonJS	Módulos ECMAScript
Exportação padrão	<code>module.exports.default = value;</code>	<code>export default value;</code>
Importação padrão	<code>const { default: value } = require(...);</code>	<code>import value from "...";</code>
Exportação de namespace	<code>module.exports = value;</code>	Não suportada
Importação de namespace	<code>const value = require(...);</code>	<code>import * as value from "...";</code>

A tipagem do TypeScript baseia seu tratamento das importações e exportações de arquivos nos módulos ECMAScript. No entanto, se o seu projeto depender de pacotes npm, como ocorre com a maioria dos projetos, provavelmente algumas dessas dependências ainda serão publicadas como módulos CommonJS. Além disso, embora alguns pacotes seguidores das regras dos módulos ECMAScript evitem incluir uma exportação padrão, muitos desenvolvedores preferem as importações mais sucintas do estilo padrão em vez das importações de estilo namespace. O TypeScript inclui algumas opções de compilador que melhoram a interoperabilidade entre os formatos dos módulos.

## **esModuleInterop**

A opção de configuração `esModuleInterop` adiciona um pouco de lógica ao código JavaScript emitido pelo TypeScript quando `module` não é um formato de módulo ECMAScript como `"es2015"` ou `"esnext"`. Essa lógica permite que os módulos ECMAScript façam importações de módulos mesmo se eles não seguirem as regras dos módulos ECMAScript relativas às importações padrão ou de namespace.

Uma razão comum para a ativação de `esModuleInterop` seria para pacotes como "`react`" que não fornecem uma exportação padrão. Se um módulo tentar usar uma importação de estilo padrão do pacote "`react`", o TypeScript relatará um type error se `esModuleInterop` não estiver ativada:

```
import React from "react";
// ~~~~~
// Module '"file:///node_modules/@types/react/index"' can
// only be default-imported using the 'esModuleInterop' flag.
```

É bom ressaltar que `esModuleInterop` só altera diretamente como o código JavaScript emitido funcionará com importações. A opção de configuração `allowSyntheticDefaultImports` a seguir é que informa à tipagem sobre a interoperabilidade das importações.

## **allowSyntheticDefaultImports**

A opção de compilador `allowSyntheticDefaultImports` informa à tipagem que os módulos ECMAScript podem fazer importações padrão de arquivos que de outra forma seriam exportações de namespace incompatíveis do CommonJS.

Por padrão ela só será `true` se uma das afirmações a seguir for verdadeira:

- `module` for igual a "`system`" (um formato de módulo mais antigo, raramente usado e não abordado neste livro);
- `esModuleInterop` for `true` e `module` não for um formato de módulos ECMAScript como "`es2015`" ou "`esnext`".

Em outras palavras, se `esModuleInterop` for `true`, mas `module` for "`esnext`", o TypeScript assumirá que o código de saída JavaScript compilado não está usando auxiliares de interoperabilidade. Ele relataria um type error para uma importação padrão de pacotes como "`react`":

```
import React from "react";
// Module '"file:///node_modules/@types/react/index"' can only be
// default-imported using the 'allowSyntheticDefaultImports' flag`.
```

## **isolatedModules**

Transpiladores externos, como o Babel, que só operam em um arquivo de cada vez não podem usar informações de tipagem para emitir JavaScript.

Como resultado, os recursos de sintaxe do TypeScript que dependem de informações de tipo para emitir JavaScript geralmente não são suportados nesses transpiladores. A ativação da opção de compilador `isolatedModules` solicita ao TypeScript que relate um erro em qualquer instância que possa causar problemas para eles:

- Enums const, abordados no Capítulo 14, “Extensões de sintaxe”.
- Arquivos script (e não de módulo).
- Exportações de tipo autônomas, abordadas no Capítulo 14, “Extensões de sintaxe”.

Geralmente, recomendo ativar `isolatedModules` se o seu projeto usar uma ferramenta diferente do TypeScript para fazer a transpilação para JavaScript.

## JavaScript

Embora o TypeScript seja amigável e eu espere que você sempre queira escrever código nele, não é preciso escrever todos os seus arquivos-fonte em TypeScript. Mesmo que por padrão o TypeScript ignore arquivos com extensão `.js` ou `.jsx`, usar suas opções de compilador `allowJs` e/ou `checkJs` permitirá que ele leia, compile e até mesmo – com capacidade limitada – verifique tipos de arquivos JavaScript.



Uma estratégia comum para a conversão de projetos JavaScript existentes para TypeScript seria começar com apenas alguns arquivos sendo convertidos para TypeScript. E mais arquivos poderiam ser adicionados com o tempo até não haver mais nenhum arquivo JavaScript. Você não precisa apostar tudo no TypeScript antes de estar pronto!

### `allowJs`

A opção de compilador `allowJs` permite que as estruturas declaradas em arquivos JavaScript sejam consideradas na verificação de tipos de arquivos TypeScript. Quando combinada com a opção de compilador `jsx`, arquivos `.jsx` também são permitidos.

Por exemplo, veja este arquivo `index.ts` importando um valor (`value`) declarado em um arquivo `values.js`:

```
// index.ts
import { value } from "./values";

console.log(`Quote: '${value.toUpperCase()}'`);
// values.js
export const value = "We cannot succeed when half of us are held back.";
```

Sem `allowJs` ativada, a instrução `import` não teria um tipo conhecido. Por padrão ele seria implicitamente `any` ou uma mensagem de type error como “Could not find a declaration file for module `./values`” (Não é possível encontrar um arquivo de declaração para o módulo `./values`) seria exibida.

`allowJs` também adiciona arquivos JavaScript à lista de arquivos compilados para o target ECMAScript e emitidos como JavaScript. Mapas de fonte e arquivos de declaração também serão produzidos se as opções próprias para isso estiverem ativadas:

```
tsc --allowJs
{
  "compilerOptions": {
    "allowJs": true
  }
}
```

Com `allowJs` ativada, o valor importado seria de tipo `string`. Nenhum type error seria relatado.

## checkJs

O TypeScript pode ir além de apenas incluir arquivos JavaScript na verificação de tipos de arquivos TypeScript: ele também pode verificar o tipo de arquivos JavaScript. A opção de compilador `checkJs` atende a duas finalidades:

- Por padrão `allowJs` passa a ser igual a `true` se ainda não o for.
- Ativação do verificador de tipos em arquivos `.js` e `.jsx`.

Ativar `checkJs` fará o TypeScript tratar os arquivos JavaScript como se eles fossem arquivos TypeScript sem nenhuma sintaxe específica do

TypeScript. Incompatibilidades de tipos, nomes de variáveis escritos incorretamente etc., causarão type errors, como ocorreria normalmente em um arquivo TypeScript:

```
tsc --checkJs
{
  "compilerOptions": {
    "checkJs": true
  }
}
```

Com `checkJs` ativada, este arquivo JavaScript geraria um alerta de verificação de tipos por um nome de variável incorreto:

```
// index.js
let myQuote = "Each person must live their life as a model for others.";

console.log(quote);
//           ~~~~~
// Error: Cannot find name 'quote'. Did you mean 'myQuote'?
```

Sem `checkJs` ativada, o TypeScript não teria relatado um type error para esse possível bug.

## @ts-check

Alternativamente, `checkJs` pode ser ativada por arquivo com a inclusão de um comentário `// @ts-check` no início do arquivo. Isso ativará a opção `checkJs` apenas para esse arquivo JavaScript:

```
// index.js
// @ts-check
let myQuote = "Each person must live their life as a model for others.";

console.log(quote);
//           ~~~~~
// Error: Cannot find name 'quote'. Did you mean 'myQuote'?
```

## Supporte ao JSDoc

Já que o JavaScript não tem a rica sintaxe de tipos do TypeScript, os tipos de valores declarados em arquivos JavaScript com frequência não são tão precisos como os declarados em arquivos TypeScript. Por exemplo, enquanto o TypeScript consegue inferir o valor de um objeto declarado

como uma variável em um arquivo JavaScript, não há uma maneira nativa no JavaScript de declarar nesse arquivo que o valor adere a alguma interface específica.

Mencionei no Capítulo 1, “Do JavaScript ao TypeScript”, que o padrão JSDoc da comunidade fornece algumas maneiras de descrever tipos com o uso de comentários. Quando `allowJs` e/ou `checkJs` estiverem ativadas, o TypeScript reconhecerá qualquer definição JSDoc existente no código.

Por exemplo, este trecho de código declara em JSDoc que a função `sentenceCase` recebe uma `string`. O TypeScript poderá então inferir que ela retorna uma `string`. Com `checkJs` ativada, o TypeScript saberia que posteriormente seria preciso relatar um type error por ela ter recebido uma `string[]`:

```
// index.js

/**
 * @param {string} text
 */
function sentenceCase(text) {
    return `${text[0].toUpperCase()} ${text.slice(1)}.`;
}
sentenceCase("hello world");// Ok

sentenceCase(["hello", "world"]);
//           ~~~~~
// Error: Argument of type 'string[]' is not
// assignable to parameter of type 'string'.
```

O suporte ao JSDoc no TypeScript é útil por adicionar incrementalmente a verificação de tipos a projetos que não tenham tempo ou familiaridade do desenvolvedor para a conversão para TypeScript.



A lista completa da sintaxe JSDoc suportada está disponível em <https://www.typescriptlang.org/docs/handbook/jsdoc-supported-types.html>.

## Extensões de configuração

À medida que você for escrevendo cada vez mais projetos TypeScript, notará que está escrevendo as mesmas configurações de projeto

repetidamente. Embora o TypeScript não permita que arquivos de configuração sejam escritos em JavaScript e usem `import` ou `require`, ele oferece um mecanismo para um arquivo TSConfig aceitar “estender”, ou copiar, valores de configuração de outro arquivo de configuração.

## **extends**

Um arquivo TSConfig pode estender outro arquivo TSConfig com a opção de configuração `extends`. `extends` recebe um caminho de outro arquivo TSConfig e indica que todas as configurações deste arquivo devem ser copiadas. Ela se comporta de maneira semelhante à palavra-chave `extends` nas classes: qualquer opção declarada na configuração derivada, ou filha, sobrescreverá opções de mesmo nome da configuração base, ou pai.

Por exemplo, muitos repositórios que têm vários arquivos TSConfig, como os monorepos que contêm vários diretórios `packages/*`, por convenção criam um arquivo `tsconfig.base.json` para os arquivos `tsconfig.json` o estenderem:

```
// tsconfig.base.json
{
  "compilerOptions": {
    "strict": true
  }
}
// packages/core/tsconfig.json
{
  "extends": "../../tsconfig.base.json",
  "includes": ["src"]
}
```

Observe que as `compilerOptions` são incluídas recursivamente. Cada opção de compilador de um arquivo TSConfig base será copiada para um arquivo TSConfig derivado, a não ser que o arquivo TSConfig derivado sobrescreva essa opção específica.

Se o exemplo anterior adicionasse um arquivo TSConfig que incluísse a opção `allowJs`, esse novo arquivo TSConfig derivado ainda teria `compilerOptions.strict` configurada com `true`:

```
// packages/js/tsconfig.json
{
  "extends": "../../tsconfig.base.json",
  "compilerOptions": {
    "allowJs": true
  },
  "includes": ["src"]
}
```

## Extensão de módulos

A propriedade `extends` pode apontar para um dos dois tipos de importação JavaScript a seguir:

*Absoluta*

Começando com @ ou uma letra.

*Relativa*

Um caminho de arquivo local começando com .

Quando o valor de `extends` for um caminho absoluto, isso indica que o arquivo TSConfig deve ser estendido a partir de um módulo npm. O TypeScript usará o sistema de resolução de módulo comum do Node para encontrar um pacote com o mesmo nome. Se o arquivo `package.json` desse pacote tiver um campo "tsconfig" contendo uma string de caminho relativo, o arquivo TSConfig desse caminho será usado. Caso contrário, o arquivo `tsconfig.json` do pacote será usado.

Muitas organizações usam pacotes npm para padronizar as opções de compilador do TypeScript entre os repositórios e/ou dentro do monorepos. Os arquivos TSConfig a seguir contêm o que você poderia definir para um monorepo de uma organização @my-org. `packages/js` precisa especificar a opção de compilador `allowJs`, enquanto `packages/ts` não altera nenhuma opção de compilador:

```
// packages/tsconfig.json
{
  "compilerOptions": {
    "strict": true
  }
}
// packages/js/tsconfig.json
```

```
{  
  "extends": "@my-org/tsconfig",  
  "compilerOptions": {  
    "allowJs": true  
  },  
  "includes": ["src"]  
}  
// packages/ts/tsconfig.json  
{  
  "extends": "@my-org/tsconfig",  
  "includes": ["src"]  
}
```

## Bases de configuração

Em vez de criar sua própria configuração a partir do zero ou das sugestões de `--init`, você pode começar com um arquivo TSConfig “base” pré-criado e personalizado para um ambiente de runtime específico. Essas bases de configuração pré-criadas estão disponíveis no registro do pacote npm em `@tsconfig/`, como em `@tsconfig/recommended` ou `@tsconfig/node16`.

Por exemplo, para instalar a base TSConfig recomendada para `deno`, precisaríamos do seguinte:

```
npm install --save-dev @tsconfig/deno  
# or  
yarn add --dev @tsconfig/deno
```

Uma vez que um pacote base de configuração for instalado, ele poderá ser referenciado como qualquer outra extensão de configuração de pacote npm:

```
{  
  "extends": "@tsconfig/deno/tsconfig.json"  
}
```

A lista completa de bases TSConfig está documentada em <https://github.com/tsconfig/bases>.



Geralmente, é uma boa ideia saber que opções de configuração do TypeScript seu arquivo está usando, mesmo se você não estiver alterando-as por conta própria.

## Referências de projeto

Para cada um dos arquivos de configuração do TypeScript que mostrei até agora assumimos que eles estavam gerenciando todos os arquivos-fonte de um projeto. Em projetos maiores pode ser útil usar arquivos de configuração diferentes para áreas distintas do projeto. O TypeScript permite definir um sistema de “referências de projeto” no qual vários projetos podem ser criados em conjunto. Definir referências de projeto dá um pouco mais de trabalho do que usar um único arquivo `TSConfig`, mas traz muitos benefícios importantes:

- Você pode especificar opções de compilador diferentes para certas áreas de código.
- O TypeScript poderá armazenar saídas de build em cache para projetos individuais, o que geralmente resulta em um tempo de build significativamente mais rápido para projetos grandes.
- As referências de projeto impõem o uso de uma “árvore de dependências” (permitindo que apenas certos projetos importem arquivos de outros projetos específicos), o que pode ajudar a estruturar áreas discretas de código.



Normalmente, as referências de projeto são usadas em projetos maiores contendo várias áreas de código distintas, como monorepos e sistemas de componentes modulares. Provavelmente você não vai querer usá-las para projetos pequenos que não tenham dezenas ou uma quantidade maior de arquivos.

As três seções a seguir mostram como criar configurações de projeto que ajudem a usar as referências de projeto:

- O modo `composite` de um arquivo `TSConfig` impõe que ele funcione de maneiras adequadas aos modos de build (modos de compilação) de vários `TSConfig`.
- A configuração `references` em um arquivo `TSConfig` indica de quais composições de `TSConfig` ele depende.
- O modo de build usa referências compostas de `TSConfig` para orquestrar a compilação de seus arquivos.

## **composite**

O TypeScript permite que um projeto use a opção de configuração `composite` para indicar que as entradas e saídas de seu sistema de arquivos obedecerão a restrições que tornarão mais fácil para as ferramentas de build determinarem se suas saídas de build estão atualizadas em comparação às suas entradas de build. Quando `composite` for `true`:

- Se ainda não tiver sido definida explicitamente, por padrão a configuração de `rootDir` usará o diretório que contém o arquivo `TSConfig`.
- Todos os arquivos de implementação devem seguir um padrão de inclusão ou estar listados no array `files`.
- `declaration` deve ser ativada.

O trecho de configuração a seguir atende todas as condições para a ativação do modo `composite` em um diretório `core/`:

```
// core/tsconfig.json
{
  "compilerOptions": {
    "declaration": true
  },
  "composite": true
}
```

Essas alterações ajudarão o TypeScript a impor que todos os arquivos de entrada do projeto criem um arquivo `.d.ts` correspondente. Geralmente, `composite` é mais útil quando usada com a opção de configuração `references` descrita a seguir.

## **references**

Um projeto TypeScript pode indicar que depende das saídas geradas por um projeto TypeScript de padrão composite com uma configuração `references` em seu arquivo `TSConfig`. A importação de módulos de um projeto referenciado será vista na tipagem como a importação de seus arquivos de declaração `.d.ts` de saída.

Este trecho de configuração define que um diretório `shell/` referencia um diretório `core/` como sua entrada:

```
// shell/tsconfig.json
{
  "references": [
    { "path": "../core" }
  ]
}
```



A opção de configuração `references` não será copiada dos TSConfigs base para os TSConfigs derivados por meio de `extends`.

Geralmente, `references` é mais útil quando usada com o modo de build descrito a seguir.

## Modo de build

Uma vez que uma área de código tiver sido configurada para usar referências de projeto, será possível utilizar o `tsc` em seu modo alternativo de “build” por meio da flag `-b`/`--b` da CLI. O modo de build transforma o `tsc` em algo como um coordenador de build do projeto. Ele permite que o `tsc` só recompile os projetos que tiverem sido alterados desde o último build, com base na última vez em que seu conteúdo e suas saídas de arquivo foram gerados.

Mais precisamente, o modo de build do TypeScript fará o seguinte ao receber um TSConfig:

1. Encontrará os projetos referenciados por esse TSConfig.
2. Detectará se eles estão atualizados.
3. Compilará os projetos desatualizados na ordem correta.
4. Compilará o TSConfig fornecido se ele ou alguma de suas dependências tiverem mudado.

O fato do modo de build do TypeScript não precisar recompilar projetos atualizados pode melhorar significativamente o desempenho da compilação.

## Configurações de coordenador

Um padrão útil comum para a definição de referências de projeto do TypeScript em um repositório seria a configuração de um arquivo

`tsconfig.json` de nível raiz com um array `files` vazio e referências para todas as referências de projeto do repositório. Esse TSConfig raiz não instruirá o TypeScript a compilar nenhum arquivo. Em vez disso, ele apenas solicitará ao TypeScript que compile os projetos referenciados quando necessário.

Este arquivo `tsconfig.json` solicita a compilação dos projetos `packages/core` e `packages/shell` em um repositório:

```
// tsconfig.json
{
  "files": [],
  "references": [
    { "path": "./packages/core" },
    { "path": "./packages/shell" }
  ]
}
```

Pessoalmente gosto de padronizar a inserção de um script em meu arquivo `package.json` chamado `build` ou `compile` para chamar `tsc -b` como um atalho:

```
// package.json
{
  "scripts": {
    "build": "tsc -b"
  }
}
```

## Opções do modo de build

O modo de build suporta algumas opções da CLI específicas de build:

- `--clean`: exclui as saídas dos projetos especificados (pode ser combinada com `--dry`);
- `--dry`: mostra o que seria feito, mas na verdade não compila nada;
- `--force`: age como se todos os projetos estivessem desatualizados;
- `-w/-watch`: semelhante ao modo de observação típico do TypeScript.

Já que o modo de build suporta o modo de observação, executar um comando como `tsc -b -w` pode ser uma maneira rápida de obter uma lista atualizada de todos os erros de compilador em um projeto grande.

## Resumo

Neste capítulo, você percorreu várias das opções de configuração importantes fornecidas pelo TypeScript:

- Uso do `tsc`, incluindo seus modos `pretty` e de observação.
- Uso de arquivos `TSConfig`, incluindo a criação de um com `tsc --init`.
- Alteração dos arquivos que serão incluídos pelo compilador TypeScript.
- Permitir a sintaxe `JSX` em arquivos `.tsx` e/ou a sintaxe `JSON` em arquivos `.json`.
- Alteração do diretório, do target de versão do `ECMAScript`, do arquivo de declaração e/ou das saídas de mapa de fonte com arquivos.
- Alteração dos tipos de biblioteca internos usados na compilação.
- Modo estrito e flags de rigidez úteis como `noImplicitAny` e `strictNullChecks`.
- Suporte a diferentes sistemas de módulos e alteração da resolução de módulo.
- Permitir a inclusão de arquivos `JavaScript` e optar por usar a verificação de tipos nesses arquivos.
- Uso de `extends` para o compartilhamento de opções de configuração entre arquivos.
- Uso de referências de projeto e do modo de build para a orquestração de build com vários `TSConfig`.



Agora que você terminou de ler este capítulo, pratique o que aprendeu em <https://learningtypescript.com/configuration-options>.

*Qual é a opção de compilador do TypeScript favorita dos disciplinadores?*  
`strict`.<sup>4</sup>

---

<sup>1</sup> N.T.: Original: *Compiler options: / Types and modules and oh my! / tsc your way.*

<sup>2</sup> N.T.: Em ciência da computação, um açúcar sintático é uma sintaxe dentro da linguagem de

programação que tem por finalidade tornar suas estruturas mais fáceis de serem lidas e expressas.

3 N.T.: Os polyfills são usados para conseguirmos empregar recursos mais modernos do JavaScript em navegadores mais antigos.

4 N.T.: Original: *What is a disciplinarian's favorite TypeScript compiler option?* **strict**.

## PARTE IV

# Material complementar

O JavaScript existe há algumas décadas e as pessoas já fizeram muitas coisas estranhas com ele. A sintaxe e a tipagem do TypeScript têm de poder representar todas essas coisas estranhas a fim de permitir que qualquer desenvolvedor JavaScript trabalhe com o TypeScript. Como resultado, existem aspectos da linguagem TypeScript que não são vistos na maioria dos códigos do dia a dia, mas que são relevantes, e até mesmo necessários, para o trabalho com alguns tipos de projetos.

Considero essas partes da linguagem como “material complementar” porque você pode deixá-las totalmente de lado e mesmo assim ser um desenvolvedor TypeScript produtivo. Na verdade, no que diz respeito aos tipos lógicos introduzidos perto do fim da seção, espero que você não precise usá-los com muita frequência – caso precise usá-los.

# CAPÍTULO 14

## Extensões de sintaxe

*“O TypeScript não adiciona nada ao runtime JavaScript”.  
... era tudo uma mentira?!<sup>1</sup>*

Quando o TypeScript foi lançado em 2012, a complexidade das aplicações web estava crescendo com maior rapidez do que o JavaScript conseguia adicionar recursos que a suportassem. A linguagem mais popular que compilava JavaScript na época, o CoffeeScript, se destacou ao divergir da linguagem com a introdução de novas e empolgantes estruturas sintáticas. Atualmente, estender a sintaxe JavaScript com novos recursos de runtime específicos de uma linguagem de superconjunto como o TypeScript é considerado prática não recomendada por várias razões:

- A mais importante: as extensões de sintaxe de runtime podem entrar em conflito com a nova sintaxe de versões mais recentes do JavaScript.
- Elas tornam mais difícil para programadores iniciantes na linguagem saberem onde o JavaScript termina e a outra linguagem começa.
- Elas aumentam a complexidade dos transpiladores que recebem código da linguagem de superconjunto e emitem JavaScript.

Logo, é com pesar que informo que os primeiros projetistas introduziram três extensões de sintaxe ao JavaScript na linguagem TypeScript:

- Classes, que estavam alinhadas com as classes JavaScript quando a especificação foi ratificada.
- Enums, um açúcar sintático simples semelhante a um objeto de chaves e valores.
- Namespaces, uma solução que antecede os módulos modernos para a

estruturação e a organização do código.

 Felizmente, o “pecado original” do TypeScript de adicionar extensões de sintaxe de runtime ao JavaScript não é uma decisão de design que os projetistas da linguagem tomaram desde seus primórdios. O TypeScript não adiciona novas estruturas de sintaxe de runtime até que elas tenham feito um progresso significativo no processo de ratificação para serem adicionadas ao próprio JavaScript.

As classes do TypeScript acabam se parecendo e se comportando de forma quase idêntica às classes do JavaScript (ufa!) com exceção do comportamento de `useDefineForClassFields` (uma opção de configuração não abordada neste livro) e das propriedades de parâmetros (abordadas aqui). Os enums ainda são usados em alguns projetos porque ocasionalmente eles são úteis. Praticamente nenhum projeto novo usa namespaces.

O TypeScript adotou uma proposta experimental para os “decorators” do JavaScript que também abordarei.

## Propriedades de parâmetros de classe

 Recomendo evitar a utilização de propriedades de parâmetros de classe a não ser que você esteja trabalhando em um projeto que faça uso intensivo de classes ou de um framework que se beneficiaria delas.

É comum nas classes JavaScript que elas recebam um parâmetro em um construtor que seja atribuído imediatamente a uma propriedade da classe.

Esta classe `Engineer` recebe um único parâmetro `area` de tipo `string` e o atribui a uma propriedade `area` de tipo `string`:

```
class Engineer {  
    readonly area: string;  
  
    constructor(area: string) {  
        this.area = area;  
        console.log(`I work in the ${area} area.`);  
    }  
}
```

```
}

// Type: string
new Engineer("mechanical").area;
```

O TypeScript inclui uma sintaxe de abreviação para a declaração desses tipos de “propriedades de parâmetros”: propriedades que são atribuídas a uma propriedade membro do mesmo tipo no começo do construtor de uma classe. Inserir `readonly` e/ou um dos modificadores de privacidade – `public`, `protected` ou `private` – na frente do parâmetro de um construtor indica para o TypeScript que ele também deve declarar uma propriedade com esse mesmo nome e tipo.

O exemplo de `Engineer` anterior poderia ser reescrito em TypeScript usando uma propriedade de parâmetro para `area`:

```
class Engineer {
    constructor(readonly area: string) {
        console.log(`I work in the ${area} area.`);
    }
}

// Type: string
new Engineer("mechanical").area;
```

As propriedades de parâmetros são atribuídas no começo do construtor da classe (ou depois da chamada a `super()` se a classe for derivada de uma classe base). Elas podem ser combinadas com outros parâmetros e/ou propriedades em uma classe.

A classe `NamedEngineer` a seguir declara uma propriedade `fullName` comum, um parâmetro `name` comum e uma propriedade de parâmetro `area`:

```
class NamedEngineer {
    fullName: string;

    constructor(
        name: string,
        public area: string,
    ) {
        this.fullName = `${name}, ${area} engineer`;
    }
}
```

O código TypeScript equivalente sem propriedades de parâmetro tem aparência semelhante, mas com algumas linhas de código a mais para atribuir `area` explicitamente:

```
class NamedEngineer {  
    fullName: string;  
    area: string;  
  
    constructor(  
        name: string,  
        area: string,  
    ) {  
        this.area = area;  
        this.fullName = `${name}, ${area} engineer`;  
    }  
}
```

As propriedades de parâmetros são uma questão que às vezes é debatida na comunidade TypeScript. A maioria dos projetos prefere evitá-las categoricamente, já que elas são uma extensão de sintaxe de runtime e, portanto, apresentam as mesmas desvantagens que mencionei anteriormente. Elas também não podem ser usadas com a sintaxe mais recente # de campos privados de classes.

Por outro lado, são adequadas quando usadas em projetos muito inclinados à criação de classes. As propriedades de parâmetros resolvem um problema de conveniência de ser preciso declarar o nome e o tipo da propriedade de parâmetro duas vezes, que é próprio do TypeScript e não do JavaScript.

## Decorators experimentais

 Recomendo evitar os decorators se possível até que uma versão do ECMAScript seja ratificada com sua sintaxe. Se você estiver trabalhando em uma versão de um framework, como o Angular ou o NestJS, que recomende o uso de decorators TypeScript, a documentação do framework mostrará como usá-los.

Muitas linguagens que contêm classes permitem anotar, ou decorar, essas classes e/ou suas propriedades com algum tipo de lógica de runtime para

modificá-las. As funções decorator são uma proposta para o JavaScript para permitir a anotação de classes e propriedades com a inserção de um símbolo @ e do nome de uma função antes.

Por exemplo, o trecho de código a seguir mostra a sintaxe do uso de um decorator em uma classe MyClass:

```
@myDecorator  
class MyClass { /* ... */ }
```

Os decorators ainda não foram ratificados em ECMAScript, logo, então o TypeScript não dá suporte a eles por padrão a partir da versão 4.7.2. No entanto, o TypeScript inclui uma opção de compilador `experimentalDecorators` que permite que uma versão antiga e experimental deles seja usada no código. Ela pode ser ativada como outras opções de compilador por meio da CLI do `tsc` ou em um arquivo `TSConfig`, mostrado aqui:

```
{  
  "compilerOptions": {  
    "experimentalDecorators": true  
  }  
}
```

Cada uso de um decorator será executado uma vez, assim que a entidade que ele estiver decorando for criada. Cada tipo de decorator – de acessador, classe, método, parâmetro e propriedade – recebe um conjunto de argumentos diferente descrevendo a entidade que ele decora.

Por exemplo, o decorator `logOnCall` a seguir usado em um método da classe `Greeter` recebe a própria classe `Greeter`, a chave da propriedade ("log") e um objeto `descriptor` descrevendo a propriedade. A modificação de `descriptor.value` para logar antes da chamada ao método `greet` original na classe `Greeter` “decora” o método `greet`:

```
function logOnCall(target: any, key: string, descriptor: PropertyDescriptor) {  
  const original = descriptor.value;  
  console.log("[logOnCall] I am decorating", target.constructor.name);  
  
  descriptor.value = function (...args: unknown[]) {  
    console.log(`[descriptor.value] Calling '${key}' with:`, ...args);  
    return original.call(this, ...args);  
  }  
}
```

```

}

class Greeter {
  @logOnCall
  greet(message: string) {
    console.log(`[greet] Hello, ${message}!`);
  }
}

new Greeter().greet("you");
// Output log:
// "[logOnCall] I am decorating", "Greeter"
// "[descriptor.value] Calling 'greet' with:", "you"
// "[greet] Hello, you!"

```

Não detalharei as nuances e especificidades de como a antiga opção `experimentalDecorators` funciona para cada um dos tipos de decorators possíveis. O suporte aos decorators no TypeScript é experimental e não está alinhado com os últimos esboços da proposta do ECMAScript. Escrevermos nossos próprios decorators raramente é justificável em um projeto TypeScript.

## Enums

 Recomendo não usar enums a menos que você tenha um conjunto de literais que sejam repetidos com frequência, possam ser descritos por um nome comum e cujo código seja mais fácil de ler se alterado para um enum.

A maioria das linguagens de programação contém o conceito de “enum,” ou tipo enumerado, para representar um conjunto de valores relacionados. Os enums podem ser considerados um conjunto de valores literais armazenados em um objeto com um nome amigável para cada valor.

O JavaScript não inclui uma sintaxe de enum porque objetos tradicionais podem ser usados em seu lugar. Por exemplo, embora os códigos de status HTTP possam ser armazenados e usados como números, muitos desenvolvedores acham mais legível armazená-los em um objeto que os represente com seu nome amigável:

```
const StatusCodes = {
  InternalServerError: 500,
  NotFound: 404,
  Ok: 200,
  // ...
} as const;

StatusCodes.InternalServerError; // 500
```

O problema dos objetos semelhantes a enums do TypeScript é que não há uma maneira de representar adequadamente na tipagem que um valor deve ser igual a um de seus valores. Um método comum usa os modificadores de tipo `keyof` e `typeof` do Capítulo 9, “Modificadores de tipo”, para chegar a uma solução, mas isso demanda a digitação de muita sintaxe.

O tipo `StatusCodeValue` a seguir usa o valor da variável `StatusCodes` anterior para criar uma união de tipos dos valores de códigos de status numéricos possíveis:

```
// Type: 200 | 404 | 500
type StatusCodeValue = (typeof StatusCodes)[keyof typeof StatusCodes];
let statusCodeValue: StatusCodeValue;

statusCodeValue = 200; // Ok

statusCodeValue = -1;
// Error: Type '-1' is not assignable to type 'StatusCodeValue'.
```

O TypeScript fornece uma sintaxe `enum` para a criação de um objeto com valores literais de tipo `number` ou `string`. Comece com a palavra-chave `enum`, depois insira o nome de um objeto – convencionalmente em PascalCase – e, em seguida, forneça um objeto `{}` contendo chaves separadas por vírgulas no enum. Opcionalmente cada chave pode usar o sinal `=` antes de um valor inicial.

O objeto `StatusCodes` anterior ficaria com esta aparência como o enum `StatusCode`:

```
enum StatusCode {
  InternalServerError = 500,
  NotFound = 404,
  Ok = 200,
```

```
}

StatusCode.InternalServerError; // 500
```

Assim como ocorre com os nomes das classes, o nome de um enum como `statusCode` pode ser usado como o nome do tipo em uma anotação de tipo. Aqui, a variável `statusCode` de tipo `statusCode` pode receber `statusCode.Ok` ou um valor numérico:

```
let statusCode: StatusCode;

statusCode = StatusCode.Ok; // Ok
statusCode = 200; // Ok
```



Por conveniência, o TypeScript permite que qualquer número seja atribuído ao valor numérico de um enum ao custo de uma menor segurança de tipos (type safety). `statusCode = -1` também teria sido permitido no trecho de código anterior.

Os enums são compilados para um objeto equivalente na saída JavaScript compilada. Cada um de seus membros torna-se uma chave do objeto com o valor correspondente e vice-versa.

O `enum` `statusCode` anterior criaria um JavaScript como este:

```
var StatusCode;
(function (StatusCode) {
    StatusCode[StatusCode["InternalServerError"] = 500] =
        "InternalServerError";
    StatusCode[StatusCode["NotFound"] = 404] = "NotFound";
    StatusCode[StatusCode["Ok"] = 200] = "Ok";
})(StatusCode || (StatusCode = {}));
```

Os enums são um tópico um pouco controverso na comunidade TypeScript. Por um lado, eles violam a regra geral do Typescript de nunca adicionar novas estruturas de sintaxe de runtime ao JavaScript. Eles apresentam uma nova sintaxe não JavaScript para os desenvolvedores aprenderem e apresentam algumas peculiaridades para opções como `preserveConstEnums`, que será abordada posteriormente neste capítulo.

Por outro lado, são muito úteis para a declaração explícita de conjuntos de valores conhecidos. Os enums são usados extensivamente nos repositórios de código-fonte tanto do TypeScript quanto do VS Code!

## Valores numéricos automáticos

Os membros do enum não precisam ter um valor inicial explícito. Quando os valores forem omitidos, o TypeScript considerará o primeiro valor como `0` e incrementará cada valor subsequente em `1` unidade. Permitir que o TypeScript selecione os valores para membros do enum é uma boa opção quando o valor não importar, só precisar ser único e estar associado ao nome da chave.

Este enum `VisualTheme` permite que o TypeScript selecione todos os valores, o que resulta em três inteiros:

```
enum VisualTheme {  
    Dark, // 0  
    Light, // 1  
    System, // 2  
}
```

No JavaScript emitido parece que os valores foram definidos explicitamente:

```
var VisualTheme;  
(function (VisualTheme) {  
    VisualTheme[VisualTheme["Dark"] = 0] = "Dark";  
    VisualTheme[VisualTheme["Light"] = 1] = "Light";  
    VisualTheme[VisualTheme["System"] = 2] = "System";  
})(VisualTheme || (VisualTheme = {}));
```

Em enums com valores numéricos, qualquer membro que não tiver um valor explícito será `1` unidade maior do que o valor anterior.

Como exemplo, para um enum `Direction` o único requisito é que seu membro `Top` tenha um valor igual a `1` e os valores restantes também serão inteiros positivos:

```
enum Direction {  
    Top = 1,  
    Right,  
    Bottom,  
    Left,  
}
```

Em sua saída JavaScript também parece que os membros restantes tinham os valores explícitos `2`, `3` e `4`:

```
var Direction;
```

```
(function (Direction) {
  Direction[Direction["Top"] = 1] = "Top";
  Direction[Direction["Right"] = 2] = "Right";
  Direction[Direction["Bottom"] = 3] = "Bottom";
  Direction[Direction["Left"] = 4] = "Left";
})(Direction || (Direction = {}));
```

 Modificar a ordem de um enum fará o número subjacente mudar. Se você mantiver esses valores em outro local, como em um banco de dados, tome cuidado ao alterar a ordem do enum ou ao remover uma entrada. Seus dados podem repentinamente ficar corrompidos porque o número salvo não representará mais o que seu código espera.

## Enums com valores string

Os enums também podem usar strings para seus membros em vez de números.

Este enum `LoadStyle` usa valores string amigáveis para seus membros:

```
enum LoadStyle {
  AsNeeded = "as-needed",
  Eager = "eager",
}
```

A saída JavaScript de enums com valores string para seus membros parece estruturalmente igual a de enums com valores numéricos:

```
var LoadStyle;
(function (LoadStyle) {
  LoadStyle["AsNeeded"] = "as-needed";
  LoadStyle["Eager"] = "eager";
})(LoadStyle || (LoadStyle = {}));
```

Os enums com valores string são úteis como aliases de constantes compartilhadas com nomes legíveis. Em vez de usarmos uma união de tipos de strings literais, os enums com valores string proporcionam autopreenchimentos e renomeações mais poderosos dessas propriedades feitos pelo editor – como abordado no Capítulo 12, “Uso de recursos do IDE”.

Uma desvantagem dos valores string para membros é que eles não podem

ser calculados automaticamente pelo TypeScript. Só membros que vêm após um membro com valor numérico podem ser calculados automaticamente.

O TypeScript poderia fornecer um valor implícito igual a `9001` para o membro `ImplicitNumber` deste enum porque o valor anterior é o número `9000`, mas seu membro `NotAllowed` geraria um erro porque ele vem após um membro de valor string:

```
enum Wat {
    FirstString = "first",
    SomeNumber = 9000,
    ImplicitNumber, // Ok (value 9001)
    AnotherString = "another",

    NotAllowed,
    // Error: Enum member must have initializer.
}
```

 Teoricamente, você poderia criar um enum de membros com valores tanto numéricos quanto de string. Na prática, provavelmente esse enum seria desnecessariamente confuso, logo, talvez seja melhor não fazer isso.

## Enums const

Já que os enums criam um objeto de runtime, usá-los produz mais código do que a estratégia alternativa comum de usar uniões de valores literais. O TypeScript permite declarar enums antecedidos pelo modificador `const` para solicitar que a linguagem omita do código JavaScript compilado sua definição de objetos e pesquisa de propriedades.

Este enum `DisplayHint` é usado como valor de uma variável `displayHint`:

```
const enum DisplayHint {
    Opaque = 0,
    Semitransparent,
    Transparent,
}
let displayHint = DisplayHint.Transparent;
```

O código da saída JavaScript compilada não conterá a declaração do

enum e usará um comentário para seu valor:

```
let displayHint = 2 /* DisplayHint.Transparent */;
```

Para projetos em que for desejável a criação de definições de objetos enum, existe uma opção de compilador `preserveConstEnums` que mantém presente a declaração do enum. Os valores usariam literais diretamente em vez de serem acessados no objeto enum.

O trecho de código anterior também omitiria a pesquisa de propriedades em sua saída JavaScript compilada:

```
var DisplayHint;
(function (DisplayHint) {
    DisplayHint[DisplayHint["Opaque"] = 0] = "Opaque";
    DisplayHint[DisplayHint["Semitransparent"] = 1] = "Semitransparent";
    DisplayHint[DisplayHint["Transparent"] = 2] = "Transparent";
})(DisplayHint || (DisplayHint = {}));

let displayHint = 2 /* Transparent */;
```

A opção `preserveConstEnums` pode ajudar a reduzir o tamanho do código JavaScript emitido, embora nem todas as maneiras de transpilar código TypeScript a suportem. Consulte o Capítulo 13, “Opções de configuração”, para obter mais informações sobre a opção de compilador `isolatedModules` e sobre quando os enums `const` podem não ter suporte.

## Namespaces



A menos que você esteja criando definições de tipo para o repositório DefinitelyTyped, não use namespaces. Os namespaces não seguem a semântica moderna de módulos do JavaScript. Sua atribuição automática de membros pode tornar a leitura do código confusa. Só estou abordando-os porque você pode encontrá-los em arquivos `.d.ts`.

Antes dos módulos ECMAScript serem ratificados, não era raro as aplicações web empacotarem grande parte de seu código de saída em um único arquivo carregado pelo navegador. Esses arquivos individuais gigantes com frequência criavam variáveis globais para conter referências a valores importantes em diferentes áreas do projeto. Era mais fácil as

páginas incluírem esse arquivo único do que usarmos um carregador de módulo antigo como o RequireJS – e geralmente era menos difícil de carregar, já que muitos servidores ainda não suportavam o streaming de download do HTTP/2. Projetos criados para saída em arquivo único precisavam de uma maneira de organizar as seções de código e essas variáveis globais.

A linguagem TypeScript forneceu uma solução com o conceito de “módulos internos”, agora conhecidos como namespaces. Um `namespace` é um objeto globalmente disponível com conteúdos “exportados” acessíveis para serem chamados como membros desse objeto. Os namespaces são definidos com a palavra-chave `namespace` seguida de um bloco de código `{}`. Tudo que estiver no bloco do namespace será avaliado dentro de um closure de função.

Este namespace `Randomized` cria uma variável `value` e a usa internamente:

```
namespace Randomized {  
    const value = Math.random();  
    console.log(`My value is ${value}`);  
}
```

Sua saída JavaScript cria um objeto `Randomized` e avalia o conteúdo do bloco dentro de uma função, logo, a variável `value` não está disponível fora do namespace:

```
var Randomized;  
(function (Randomized) {  
    const value = Math.random();  
    console.log(`My value is ${value}`);  
})(Randomized || (Randomized = {}));
```

 Os namespaces e a palavra-chave `namespace` eram originalmente chamados de “módulos” e “`module`”, respectivamente, em TypeScript. Olhando em retrospecto, essa foi uma escolha infeliz dado o surgimento dos carregadores de módulos modernos e dos módulos ECMAScript. A palavra-chave `module` ainda é ocasionalmente encontrada em muitos projetos antigos, mas pode – e deve – ser seguramente substituída por `namespace`.

## Exportações de namespaces

Um recurso importante dos namespaces que os tornou úteis era o fato de um namespace poder “exportar” conteúdos tornando-os membros do objeto de namespace. Outras áreas de código podiam então referenciar esses membros pelo nome.

Aqui, um namespace `Settings` exporta os valores de `describe`, `name` e `version` usados de maneira interna e externa em relação ao namespace:

```
namespace Settings {  
  export const name = "My Application";  
  export const version = "1.2.3";  
  export function describe() {  
    return `${Settings.name} at version ${Settings.version}`;  
  }  
  
  console.log("Initializing", describe());  
}  
  
console.log("Initialized", Settings.describe());
```

A saída JavaScript mostra que os valores são sempre referenciados como membros de `settings` (por exemplo, `settings.name`) no uso tanto interno quanto externo:

```
var Settings;  
(function (Settings) {  
  Settings.name = "My Application";  
  Settings.version = "1.2.3";  
  function describe() {  
    return `${Settings.name} at version ${Settings.version}`;  
  }  
  Settings.describe = describe;  
  console.log("Initializing", describe());  
})(Settings || (Settings = {}));  
console.log("Initialized", Settings.describe());
```

Com o uso de `var` para o objeto de saída e com os conteúdos exportados sendo referenciados como membros desses objetos, por design os namespaces também funcionam bem quando divididos entre vários arquivos. O namespace `Settings` anterior poderia ser reescrito para vários arquivos:

```

// settings/constants.ts
namespace Settings {
    export const name = "My Application";
    export const version = "1.2.3";
}
// settings/describe.ts
namespace Settings {
    export function describe() {
        return `${Settings.name} at version ${Settings.version}`;
    }

    console.log("Initializing", describe());
}
// index.ts
console.log("Initialized", Settings.describe());

```

A saída JavaScript concatenada teria uma aparência como esta:

```

// settings/constants.ts
var Settings;
(function (Settings) {
    Settings.name = "My Application";
    Settings.version = "1.2.3";
})(Settings || (Settings = {}));
// settings/describe.ts
(function (Settings) {
    function describe() {
        return `${Settings.name} at version ${Settings.version}`;
    }

    Settings.describe = describe;
    console.log("Initialized", describe());
})(Settings || (Settings = {}));
console.log("Initialized", Settings.describe());

```

Nas formas de declaração tanto de arquivo único quanto de vários arquivos, o objeto de saída no runtime teria três chaves. Então, ficaria:

```

const Settings = {
    describe: function describe() {
        return `${Settings.name} at version ${Settings.version}`;
    },
    name: "My Application",
    version: "1.2.3",
};

```

A principal diferença com o uso de um namespace é que ele pode ser

dividido em diferentes arquivos e mesmo assim os membros podem referenciar uns aos outros com o nome do namespace.

## Namespaces aninhados

Os namespaces podem ser “aninhados” em um número ilimitado de níveis pela exportação de um namespace a partir de outro namespace ou com a inserção de um ou mais pontos (.) dentro de um nome.

As duas declarações de namespace a seguir se comportariam de forma idêntica:

```
namespace Root.Nested {  
    export const value1 = true;  
}  
  
namespace Root {  
    export namespace Nested {  
        export const value2 = true;  
    }  
}
```

Ambas são compiladas para código estruturalmente idêntico:

```
(function (Root) {  
    let Nested;  
    (function (Nested) {  
        Nested.value2 = true;  
    })(Nested || (Nested = {}));  
})(Root || (Root = {}));
```

Os namespaces aninhados são uma maneira útil de impor mais delinearção entre seções dentro de projetos maiores organizados com namespaces. Muitos desenvolvedores optavam por usar um namespace raiz com o nome de seu projeto – talvez dentro de um namespace para sua empresa ou organização – e namespaces filhos para cada área principal do projeto.

## Namespaces em definições de tipo

Atualmente, a única qualidade redentora dos namespaces – e a única razão para eu tê-los incluído neste livro – é que eles podem ser úteis para definições de tipo do repositório DefinitelyTyped. Muitas bibliotecas JavaScript – principalmente as que serviram de base de aplicações web

mais antigas como o jQuery – são definidas para serem incluídas em navegadores web com uma tag `<script>` tradicional não relacionada a módulos. Suas tipagens precisam indicar que elas criam uma variável global disponível para todo o código – estrutura capturada perfeitamente pelos namespaces.

Além disso, muitas bibliotecas JavaScript habilitadas para navegador são definidas tanto para ser importadas em sistemas de módulos mais modernos quanto também para criar um namespace global. O TypeScript permite que a definição de tipo de um módulo inclua `export as namespace`, seguido de um nome global, para indicar que o módulo também está disponível globalmente com esse nome.

Por exemplo, este arquivo de declaração de um módulo exporta `value` e está disponível globalmente:

```
// node_modules/@types/my-example-lib/index.d.ts
export const value: number;
export as namespace libExample;
```

A tipagem saberia que tanto `import("my-example-lib")` quanto `window.libExample` retornariam o módulo, com uma propriedade `value` de tipo `number`:

```
// src/index.ts
import * as libExample from "my-example-lib"; // Ok
const value = window.libExample.value; // Ok
```

## Prefira módulos em vez de namespaces

Em vez de usar namespaces, os arquivos `settings/constants.ts` e `settings/describe.ts` dos exemplos anteriores poderiam ser reescritos de acordo com padrões modernos com módulos ECMAScript:

```
// settings/constants.ts
export const name = "My Application";
export const version = "1.2.3";
// settings/describe.ts
import { name, version } from "./constants";

export function describe() {
    return `${Settings.name} at version ${Settings.version}`;
}
```

```
console.log("Initializing", describe());
// index.ts
import { describe } from "./settings/describe";

console.log("Initialized", describe());
```

Um código TypeScript estruturado com namespaces não pode ser limpo (ter os arquivos não usados removidos) facilmente em builders modernos, como o Webpack, porque os namespaces criam vínculos implícitos, em vez de serem declarados explicitamente, entre arquivos da maneira como os módulos ECMAScript o fazem. Geralmente é preferível escrever código de runtime usando módulos ECMAScript e não namespaces do TypeScript.



A partir de 2022, o próprio TypeScript passou a ser escrito em namespaces, mas a equipe está trabalhando na migração para módulos. Talvez, quando você estiver lendo este texto, eles já tenham terminado essa conversão! Cruze os dedos.

## Importações e exportações somente de tipo

Gostaria de terminar este capítulo com boas notícias. Um último conjunto de extensões de sintaxe, as importações e exportações somente de tipo, podem ser muito úteis e não adicionam nenhuma complexidade à saída JavaScript emitida.

O transpilador do TypeScript removerá dos arquivos valores de importações e exportações usados apenas na tipagem porque eles não serão usados no JavaScript de runtime.

Por exemplo, o arquivo *index.ts* a seguir cria uma variável `action` e um tipo `ActivistArea` e posteriormente exporta os dois com uma declaração de exportação autônoma. Quando ele for compilado para *index.js*, o transpilador do TypeScript saberá que deve remover `ActivistArea` dessa declaração de exportação autônoma:

```
// index.ts
const action = { area: "people", name: "Bella Abzug", role: "politician" };
```

```
type ActivistArea = "nature" | "people";

export { action, ActivistArea };
// index.js
const action = { area: "people", name: "Bella Abzug", role: "politician" };

export { action };
```

Saber que é preciso remover tipos reexportados, como `ActivistArea`, requer conhecimento da tipagem do TypeScript. Transpiladores como o Babel que só agem em um único arquivo de cada vez não têm acesso à tipagem do TypeScript para saber se cada nome só é usado na tipagem. A opção de compilador `isolatedModules`, abordada no Capítulo 13, “Opções de configuração”, ajuda a assegurar que o código seja transpilado em ferramentas que não sejam o TypeScript.

O TypeScript permite adicionar o modificador `type` na frente de nomes individuais importados ou do objeto `{...}` inteiro em declarações `export` e `import`. Isso indica que eles só devem ser usados na tipagem. Marcar a importação padrão de um pacote como `type` também é permitido.

No trecho de código a seguir, só a importação e a exportação de `value` são mantidas quando `index.ts` é transpilado para a saída `index.js`:

```
// index.ts
import { type TypeOne, value } from "my-example-types";
import type { TypeTwo } from "my-example-types";
import type DefaultType from "my-example-types";

export { type TypeOne, value };
export type { DefaultType, TypeTwo };
// index.js
import { value } from "my-example-types";

export { value };
```

Alguns desenvolvedores TypeScript preferem usar as importações somente de tipo para deixar mais claro quais importações só serão usadas como tipos. Se uma importação for marcada como somente de tipo, tentar usá-la como um valor de runtime resultará em um erro do TypeScript.

A seguir, `ClassOne` é importada normalmente e pode ser usada no runtime,

mas `ClassTwo` não pode porque é importada como um tipo:

```
import { ClassOne, type ClassTwo } from "my-example-types";

new ClassOne(); // Ok

new ClassTwo();
// ~~~~~
// Error: 'ClassTwo' cannot be used as a value
// because it was imported using 'import type'.
```

Em vez de adicionar complexidade ao JavaScript emitido, as importações e exportações somente de tipo deixam claro para os transpiladores externos ao TypeScript quando é possível remover trechos de código. Logo, a maioria dos desenvolvedores TypeScript não as trata com a desaprovação que direcionam às extensões de sintaxe anteriores abordadas neste capítulo.

## Resumo

Neste capítulo, você trabalhou com algumas das extensões de sintaxe do JavaScript incluídas no TypeScript:

- Declaração de propriedades de parâmetros de classe em construtores da classe.
- Uso de decorators para ampliar as classes e seus campos.
- Representação de grupos de valores com enums.
- Uso de namespaces para a criação de agrupamentos entre arquivos ou em definições de tipo.
- Importações e exportações somente de tipo.



Agora que você terminou de ler este capítulo, pratique o que aprendeu em <https://learningtypescript.com/syntax-extensions>.

*Como é chamado o custo do suporte a extensões JavaScript legadas no TypeScript?  
“Sintaxe”?*<sup>2</sup>

---

1 N.T.: Original: “*TypeScript does not add / to the JavaScript runtime.*” ...was that all a lie?!

2 N.T.: Original: *What do you call the cost of supporting legacy JavaScript extensions in TypeScript?*

“Sin tax.” O termo “sin-tax” não tem equivalente em português, e poderia ser traduzido como “custo do pecado”. O tradutor decidiu então manter “sintaxe”, com esta ressalva.

## CAPÍTULO 15

# Operações com tipos

*Condicionais, mapas  
Grandes poderes sobre os tipos  
trazem grande confusão<sup>1</sup>*

O TypeScript fornece níveis de poder extraordinários para a definição de tipos na tipagem. Até mesmo os modificadores lógicos do Capítulo 10, “Genéricos”, parecem acanhados em comparação com os recursos das operações com tipos deste capítulo. Quando você terminar o capítulo poderá misturar, combinar e modificar tipos de acordo com outros tipos – o que lhe dará maneiras poderosas de representar tipos no sistema de tipos.



Quase todos esses tipos extravagantes são provenientes de técnicas que provavelmente você não vai querer usar com muita frequência. Recomendo conhecê-los para os casos em que forem úteis, mas cuidado: eles podem ser difíceis de ler se usados em excesso. Divirta-se!

## Tipos mapeados

O TypeScript fornece uma sintaxe para a criação de um novo tipo baseado nas propriedades de outro tipo: em outras palavras, no *mapeamento* de um tipo para outro. Um *tipo mapeado* no TypeScript é aquele que recebe outro tipo e executa alguma operação em cada propriedade desse tipo.

Os tipos mapeados criam um novo tipo gerando uma nova propriedade para cada chave de um conjunto de chaves. Eles usam uma sintaxe semelhante às assinaturas de índice, mas em vez de empregar um tipo de

chave estático com :, como em [i: string], usam um tipo calculado a partir de outro tipo com in, como em [K in OriginalType]:

```
type NewType = {
    [K in OriginalType]: NewProperty;
};
```

Um uso comum para os tipos mapeados seria na criação de um objeto cujas chaves fossem cada uma das strings literais de um tipo de união existente. O tipo AnimalCounts a seguir cria um novo tipo de objeto no qual as chaves são cada um dos valores do tipo de união Animals e todos os valores são de tipo number:

```
type Animals = "alligator" | "baboon" | "cat";
```

```
type AnimalCounts = {
    [K in Animals]: number;
};

// Equivalent to:
// {
//     alligator: number;
//     baboon: number;
//     cat: number;
// }
```

Os tipos mapeados baseados em literais de uniões existentes são uma maneira conveniente de economizar espaço na declaração de grandes interfaces. No entanto, eles brilham realmente quando podem atuar em outros tipos e até mesmo adicionar ou remover modificadores de membros.

## Tipos mapeados a partir de outros tipos

Normalmente, os tipos mapeados agem em tipos existentes usando o operador keyof para capturar as chaves desse tipo existente. Instruindo um tipo para mapear as chaves de um tipo existente, podemos *fazer o mapeamento* desse tipo existente para um novo tipo.

Este tipo AnimalCounts acaba sendo igual ao tipo AnimalCounts de antes pelo mapeamento do tipo AnimalVariants para um novo tipo equivalente:

```
interface AnimalVariants {
    alligator: boolean;
```

```

    baboon: number;
    cat: string;
}

type AnimalCounts = {
    [K in keyof AnimalVariants]: number;
};

// Equivalent to:
// {
//     alligator: number;
//     baboon: number;
//     cat: number;
// }

```

As chaves do novo tipo mapeadas com `keyof` – chamado de `K` nos códigos anteriores – são sabidamente chaves do tipo original. Isso significa que o valor de cada membro do tipo mapeado pode referenciar o valor do membro correspondente do tipo original com a mesma chave.

Se o objeto original fosse `SomeName` e o mapeamento fosse `[K in keyof SomeName]`, cada membro do tipo mapeado poderia referenciar o valor do membro `SomeName` equivalente como `SomeName[K]`.

Este tipo `NullableBirdVariants` recebe um tipo original `BirdVariants` e adiciona `| null` a cada membro:

```

interface BirdVariants {
    dove: string;
    eagle: boolean;
}

type NullableBirdVariants = {
    [K in keyof BirdVariants]: BirdVariants[K] | null,
};

// Equivalent to:
// {
//     dove: string | null;
//     eagle: boolean | null;
// }

```

Em vez de copiar tediosamente cada campo de um tipo original para qualquer número de outros tipos, com os tipos mapeados podemos definir um conjunto de membros uma vez e recriar novas versões deles em

massa quantas vezes precisarmos.

## Tipos mapeados e assinaturas

No Capítulo 7, “Interfaces”, mencionei que o TypeScript fornece duas maneiras de declarar membros de interfaces como funções:

- Sintaxe de método, como `member(): void`: declara que um membro da interface é uma função para ser chamada como membro do objeto.
- Sintaxe de propriedade, como `member: () => void`: declara que um membro da interface é igual a uma função autônoma.

Os tipos mapeados não diferenciam as sintaxes de método e propriedade em tipos de objeto. Eles tratam os métodos como propriedades nos tipos originais.

Este tipo `ResearcherProperties` contém os membros `property` e `method` de `Researcher`:

```
interface Researcher {  
    researchMethod(): void;  
    researchProperty: () => string;  
}  
  
type JustProperties<T> = {  
    [K in keyof T]: T[K];  
};  
  
type ResearcherProperties = JustProperties<Researcher>;  
// Equivalent to:  
// {  
//     researchMethod: () => void;  
//     researchProperty: () => string;  
// }
```

Na prática, a diferença entre métodos e propriedades não aparece com muita frequência na maioria dos códigos usados. É raro encontrarmos um uso prático para um tipo mapeado que receba um tipo de classe.

## Alteração de modificadores

Os tipos mapeados também podem alterar os modificadores de controle de acesso – `readonly` e a optionalidade de `?` – dos membros dos tipos

originais. `readonly` ou `?:` pode ser inserido em membros de tipos mapeados com o uso da mesma sintaxe de interfaces típicas.

O tipo  `ReadonlyEnvironmentalist` a seguir cria uma versão da interface `Environmentalist` com todos os membros recebendo `readonly`, enquanto `Optional ReadonlyConservationist` dá um passo além e cria outra versão que adiciona `?:` a todos os membros de  `ReadonlyEnvironmentalist`:

```
interface Environmentalist {
    area: string;
    name: string;
}

type ReadonlyEnvironmentalist = {
    readonly [K in keyof Environmentalist]: Environmentalist[K];
};

// Equivalent to:
// {
//     readonly area: string;
//     readonly name: string;
// }

type OptionalReadonlyEnvironmentalist = {
    [K in keyof ReadonlyEnvironmentalist]?: ReadonlyEnvironmentalist[K];
};

// Equivalent to:
// {
//     readonly area?: string;
//     readonly name?: string;
// }
```



Alternativamente, o tipo `Optional ReadonlyEnvironmentalist` poderia ser escrito com `readonly [K in keyof Environmentalist]?: Environmentalist[K]`.

A remoção de modificadores é executada com o acréscimo de `-` antes do modificador em um novo tipo. Em vez de escrever `readonly` ou `?:`, você pode escrever `-readonly` ou `-?:`, respectivamente.

O tipo  `Conservationist` a seguir contém membros `readonly` e/ou opcionais marcados com `?` que passam a ser graváveis em  `WritableConservationist` e depois também tornam-se obrigatórios em  `RequiredWritableConservationist`:

```
interface Conservationist {
```

```

        name: string;
        catchphrase?: string;
        readonly born: number;
        readonly died?: number;
    }

type WritableConservationist = {
    -readonly [K in keyof Conservationist]: Conservationist[K];
};

// Equivalent to:
// {
//     name: string;
//     catchphrase?: string;
//     born: number;
//     died?: number;
// }

type RequiredWritableConservationist = {
    [K in keyof WritableConservationist]-?: WritableConservationist[K];
};

// Equivalent to:
// {
//     name: string;
//     catchphrase: string;
//     born: number;
//     died: number;
// }

```



Alternativamente, o tipo `RequiredWritableConservationist` poderia ser escrito com `-readonly [K in keyof Conservationist]-?: Conservationist[K]`.

## Tipos mapeados genéricos

O poder máximo dos tipos mapeados vem da sua combinação com os genéricos, permitindo que um único tipo de mapeamento seja reutilizado entre diferentes tipos. Os tipos mapeados podem acessar a chave (`keyof`) do nome de qualquer tipo de seu escopo, incluindo um parâmetro de tipo do próprio tipo mapeado.

Normalmente, os tipos mapeados genéricos são usados para representar a transformação dos dados à medida que eles fluem por uma aplicação. Por

exemplo, poderia ser desejável que uma área da aplicação pudesse receber valores de tipos existentes, mas sem poder modificar os dados.

O tipo genérico `MakeReadonly` a seguir pode receber qualquer tipo e cria uma nova versão com o modificador `readonly` adicionado a todos os seus membros:

```
type MakeReadonly<T> = {
    readonly [K in keyof T]: T[K];
}

interface Species {
    genus: string;
    name: string;
}

type ReadonlySpecies = MakeReadonly<Species>;
// Equivalent to:
// {
//     readonly genus: string;
//     readonly name: string;
// }
```

Outra transformação que normalmente os desenvolvedores precisam representar ocorre quando uma função recebe parte de uma interface e retorna uma instância totalmente preenchida dessa interface.

O tipo `MakeOptional` e a função `createGenusData` a seguir permitem fornecer uma parcela da interface `GenusData` e obter um objeto com os padrões já preenchidos:

```
interface GenusData {
    family: string;
    name: string;
}

type MakeOptional<T> = {
    [K in keyof T]?: T[K];
}
// Equivalent to:
// {
//     family?: string;
//     name?: string;
// }
```

```

/**
 * Spreads any {overrides} on top of default values for GenusData.
 */
function createGenusData(overrides?: MakeOptional<GenusData>): GenusData {
    return {
        family: 'unknown',
        name: 'unknown',
        ...overrides,
    }
}

```

Algumas operações executadas pelos tipos mapeados genéricos são tão úteis que o TypeScript fornece tipos utilitários para elas prontos para serem usados. Tornar opcionais todas as propriedades, por exemplo, é algo que pode ser feito com o uso do tipo interno `Partial<T>`. Você pode encontrar uma lista desses tipos internos em <https://www.typescriptlang.org/docs/handbook/utility-types.html>.

## Tipos condicionais

É elegante mapear tipos existentes para outros tipos, mas ainda não adicionamos condições lógicas à tipagem. Faremos isso agora.

A tipagem do TypeScript é um exemplo de *linguagem de programação lógica*. Ela permite criar novas estruturas (tipos) de acordo com a verificação lógica de tipos anteriores. Isso é feito com o conceito de *tipo condicional*: um tipo que é definido como um entre dois tipos possíveis, tendo como base um tipo existente.

A sintaxe dos tipos condicionais parece com a dos ternários:

```
LeftType extends RightType ? IfTrue : IfFalse
```

A verificação lógica em um tipo condicional resume-se sempre a se o tipo da esquerda *estende* o tipo da direita, ou seja, se é atribuível a ele.

O tipo condicional `CheckStringAgainstNumber` a seguir verifica se `string extends number` – ou, em outras palavras, se o tipo `string` é atribuível ao tipo `number`. Não é, logo, o tipo resultante se enquadra no caso “se falso”: `false`:

```
// Type: false
type CheckStringAgainstNumber = string extends number ? true : false;
```

Grande parte do restante deste capítulo envolverá a combinação de outros recursos da tipagem com os tipos condicionais. À medida que os trechos de código forem se tornando mais complexos, lembre-se: cada tipo condicional é apenas lógica booleana. Eles recebem algum tipo e geram um entre dois resultados possíveis.

## Tipos condicionais genéricos

Os tipos condicionais podem verificar qualquer nome de tipo de seu escopo, incluindo um parâmetro de tipo do próprio tipo condicional. Isso significa que você pode escrever tipos genéricos reutilizáveis para criar novos tipos baseados em qualquer outro tipo.

Transformar o tipo `CheckStringAgainstNumber` anterior em um `CheckAgainstNumber` genérico fornecerá um tipo que será `true` ou `false` dependendo se o tipo anterior for atribuível a `number`. `string` continua não sendo `true`, enquanto tanto `number` quanto `0 | 1` são:

```
type CheckAgainstNumber<T> = T extends number ? true : false;

// Type: false
type CheckString = CheckAgainstNumber<'parakeet'>;

// Type: true
type CheckString = CheckAgainstNumber<1891>;

// Type: true
type CheckString = CheckAgainstNumber<number>;
```

O tipo `CallableSetting` a seguir é um pouco mais útil. Ele recebe um `T` genérico e verifica se `T` é uma função. Se for, o tipo resultante será `T` – como no caso de `GetNumbersSetting` no qual `T` é `() => number[]`. Caso contrário, o tipo resultante será uma função que retorna `T`, como no caso de `StringSetting` em que `T` é `string` e, portanto, o tipo resultante é `() => string`:

```
type CallableSetting<T> =
  T extends () => any
    ? T
    : () => T
```

```

// Type: () => number[]
type GetNumbersSetting = CallableSetting<() => number[]>;

// Type: () => string
type StringSetting = CallableSetting<string>;

```

Os tipos condicionais também podem acessar membros de tipos fornecidos com a sintaxe de pesquisa de membros de objetos. Eles podem usar essas informações em sua cláusula `extends` e/ou nos tipos resultantes.

Um padrão usado pelas bibliotecas JavaScript que se adequa bem aos tipos condicionais genéricos é a alteração do tipo de retorno de uma função de acordo com um objeto de opções fornecido para ela.

Por exemplo, muitas funções de banco de dados ou equivalentes poderiam usar uma propriedade como `throwIfNotFound` para alterar a função a fim de lançar um erro em vez de retornar `undefined` se um valor não for encontrado. O tipo `QueryResult` a seguir modela esse comportamento resultando no tipo mais estrito `string` em vez de em `string | undefined` se a propriedade `throwIfNotFound` das opções não for especificamente `true`:

```

interface QueryOptions {
  throwIfNotFound: boolean;
}

type QueryResult<Options extends QueryOptions> =
  Options["throwIfNotFound"] extends true ? string : string | undefined;

declare function retrieve<Options extends QueryOptions>(
  key: string,
  options?: Options,
): Promise<QueryResult<Options>>;

// Returned type: string | undefined
await retrieve("Biruté Galdikas");

// Returned type: string | undefined
await retrieve("Jane Goodall", { throwIfNotFound: Math.random() > 0.5 });

// Returned type: string
await retrieve("Dian Fossey", { throwIfNotFound: true });

```

Pela combinação de um tipo condicional com um parâmetro de tipo

genérico, a função `retrieve` pode informar com mais precisão à tipagem como ela alterará o fluxo de controle de seu programa.

## Distributividade dos tipos

Os tipos condicionais são *distribuídos* por uniões, o que significa que o tipo resultante será uma união da aplicação desse tipo condicional a cada um dos constituintes (tipos que compõem o tipo de união). Em outras palavras, `ConditionalType<T | U>` é o mesmo que `Conditional<T> | Conditional<U>`.

É difícil explicar a distributividade dos tipos, mas é importante para sabermos como os tipos condicionais se comportam com as uniões.

Considere o tipo `ArrayifyUnlessString` a seguir que converte seu parâmetro de tipo `T` para um array a menos que `T extends string`. `HalfArrayified` seja equivalente a `string | number[]` porque `ArrayifyUnlessString<string | number>` é o mesmo que `ArrayifyUnlessString<string> | ArrayifyUnlessString<number>`:

```
type ArrayifyUnlessString<T> = T extends string ? T[] : T[];  
  
// Type: string | number[]  
type HalfArrayified = ArrayifyUnlessString<string | number>;
```

Se os tipos condicionais do TypeScript não fossem distribuídos em uniões, `HalfArrayified` seria `(string | number)[]` porque `string | number` não é atribuível a `string`. Ou seja, os tipos condicionais aplicam sua lógica a cada constituinte de um tipo de união, e não ao tipo de união inteiro.

## Tipos inferidos

O acesso a membros de tipos fornecidos funciona bem para informações que são armazenadas como um membro de um tipo, mas não consegue capturar outras informações como parâmetros de função ou tipos de retorno. Os tipos condicionais podem acessar partes arbitrárias de sua condição usando uma palavra-chave `infer` dentro de sua cláusula `extends`. Inserir a palavra-chave `infer` e um novo nome para um tipo dentro de uma cláusula `extends` significa que o novo tipo estará disponível dentro do caso `true` do tipo condicional.

Este tipo `ArrayItems` recebe um parâmetro de tipo `T` e verifica se `T` é um

array de algum novo tipo `Item`. Se for, o tipo resultante será `Item`; caso contrário, será `T`:

```
type ArrayItems<T> =
  T extends (infer Item)[]
    ? Item
    : T;

// Type: string
type StringItem = ArrayItems<string>;

// Type: string
type StringArrayItem = ArrayItems<string[]>;

// Type: string[]
type String2DItem = ArrayItems<string[][]>;
```

Os tipos inferidos também podem ser usados na criação de tipos condicionais recursivos. O tipo `ArrayItems` visto anteriormente poderia ser estendido para recuperar o tipo de item de um array de qualquer dimensionalidade recursivamente:

```
type ArrayItemsRecursive<T> =
  T extends (infer Item)[]
    ? ArrayItemsRecursive<Item>
    : T;

// Type: string
type StringItem = ArrayItemsRecursive<string>;

// Type: string
type StringArrayItem = ArrayItemsRecursive<string[]>;

// Type: string
type String2DItem = ArrayItemsRecursive<string[][]>;
```

Observe que embora `ArrayItems<string[][]>` tenha resultado em `string[]`, `ArrayItemsRecursive<string[][]>` resultou em `string`. A possibilidade dos tipos genéricos serem recursivos permite que eles se mantenham aplicando modificações – como aqui, na recuperação do tipo do elemento de um array.

## Tipos condicionais mapeados

Os tipos mapeados aplicam uma alteração a cada membro de um tipo existente. Os tipos condicionais aplicam uma alteração a um único tipo existente. Combinados, eles permitem aplicar uma lógica condicional a cada membro de um tipo de template genérico.

Este tipo `MakeAllMembersFunctions` transforma em função cada membro de um tipo que não for uma função:

```
type MakeAllMembersFunctions<T> = {
  [K in keyof T]: T[K] extends (...args: any[]) => any
    ? T[K]
    : () => T[K]
};

type MemberFunctions = MakeAllMembersFunctions<{
  alreadyFunction: () => string,
  notYetFunction: number,
}>;
// Type:
// {
//   alreadyFunction: () => string,
//   notYetFunction: () => number,
// }
```

Os tipos condicionais mapeados são uma maneira conveniente de modificar todas as propriedades de um tipo existente com o uso de alguma verificação lógica.

## never

No Capítulo 4, “Objetos”, introduzi o tipo `never`, um tipo vazio, o que significa que ele não pode ter valores e não pode ser alcançado. A inclusão de uma anotação de tipo `never` no local correto poderia fazer o TypeScript ser mais agressivo na detecção de caminhos de código que nunca fossem alcançados na tipagem e nos exemplos anteriores de código de runtime.

### never e as interseções e uniões

Outra maneira de descrever o tipo vazio `never` seria como um tipo que não pode existir. Isso faz `never` apresentar alguns comportamentos interessantes com os tipos de interseção `&` e os tipos de união `|`:

- `never` em um tipo de interseção & reduz o tipo de interseção apenas a `never`;
- `never` é ignorado em um tipo de união `|`.

Os tipos `NeverIntersection` e `NeverUnion` a seguir ilustram esses comportamentos:

```
type NeverIntersection = never & string; // Type: never
type NeverUnion = never | string; // Type: string
```

Especificamente, o comportamento de ser ignorado nos tipos de união torna `never` útil para a exclusão de valores de tipos condicionais e mapeados.

## never e os tipos condicionais

Normalmente, os tipos condicionais genéricos usam `never` para excluir tipos das uniões. Já que `never` é ignorado em uniões, o resultado de um condicional genérico em uma união de tipos só incluirá os que não forem `never`.

Este tipo condicional genérico `OnlyStrings` exclui tipos que não sejam strings, logo, o tipo `RedOrBlue` exclui `0` e `null` da união:

```
type OnlyStrings<T> = T extends string ? T : never;

type RedOrBlue = OnlyStrings<"red" | "blue" | 0 | false>;
// Equivalent to: "red" | "blue"
```

`never` também costuma ser combinado com tipos condicionais inferidos para a criação de utilitários para tipos genéricos. Inferências de tipo com `infer` têm de se enquadrar no caso true de um tipo condicional, logo, se o caso false não precisar ser usado, `never` será um tipo adequado para ser inserido aí.

Este tipo `FirstParameter` recebe um tipo de função `T`, verifica se é uma função com `arg: infer Arg` e, em caso afirmativo, retorna `Arg`:

```
type FirstParameter<T extends (...args: any[]) => any> =
  T extends (arg: infer Arg) => any
    ? Arg
    : never;

type GetsString = FirstParameter<
```

```
(arg0: string) => void  
>; // Type: string
```

Usar `never` no caso false do tipo condicional permitiu que `FirstParameter` extraísse o tipo do parâmetro de tipo da função.

## never e os tipos mapeados

O comportamento de `never` em uniões também o torna útil para a exclusão de membros em tipos mapeados. É possível excluir chaves de um objeto com o uso dos três recursos da tipagem descritos a seguir:

- `never` é ignorado em uniões;
- os tipos mapeados podem mapear membros de tipos;
- tipos condicionais podem ser usados para transformar tipos em `never` se uma condição for atendida.

Se combinarmos os três recursos, poderemos criar um tipo mapeado que altere cada membro do tipo original para a chave original ou para `never`. Assim, solicitar os membros desse tipo com `[keyof T]` produzirá uma união dos resultados de todos esses tipos mapeados, excluindo `never`.

O tipo `OnlyStringProperties` a seguir transforma cada membro de `T[K]` na chave `K` se esse membro for uma string ou em `never` se ele não o for:

```
type OnlyStringProperties<T> = {  
  [K in keyof T]: T[K] extends string ? K : never;  
}[keyof T];  
  
interface AllEventData {  
  participants: string[];  
  location: string;  
  name: string;  
  year: number;  
}  
  
type OnlyStringEventData = OnlyStringProperties<AllEventData>;  
// Equivalent to: "location" | "name"
```

Outra maneira de considerar o tipo `OnlyStringProperties<T>` seria pelo fato de ele excluir todas as propriedades que não sejam de tipo `string` (ele as transforma em `never`) e retornar todas as chaves restantes (`[keyof T]`).

## Tipos template literals

Já vimos muita coisa sobre os tipos condicionais e/ou mapeados. Passaremos para tipos que fazem menos uso de lógica e por enquanto nos concentraremos nas strings. Até agora mostrei duas estratégias para a tipagem de valores string:

- o tipo primitivo `string`: para quando o valor puder ser qualquer string existente;
- tipos literais como `""` e `"abc"`: para quando o valor só puder ser desse tipo (ou de uma união deles).

Ocasionalmente, entretanto, você pode querer indicar que uma string atende a algum padrão: parte da string é conhecida, mas parte não. Entram em cena os *tipos template literals*, uma sintaxe do TypeScript para indicar que um tipo string adere a um padrão. Eles parecem template literal strings – daí seu nome – mas com tipos primitivos ou uniões de tipos primitivos interpolados.

O tipo template literal a seguir indica que a string deve começar com `"Hello"`, mas pode terminar com qualquer string (`string`). Termos que começam com `"Hello"` como `"Hello, world!"` atendem, mas `"World! Hello!"` ou `"hi"` não:

```
type Greeting = `Hello${string}`;

let matches: Greeting = "Hello, world!"; // Ok

let outOfOrder: Greeting = "World! Hello!";
// ~~~~~
// Error: Type '"World! Hello!"' is not assignable to type ``Hello ${string}``.

let missingAltogether: Greeting = "hi";
// ~~~~~
// Error: Type '"hi"' is not assignable to type ``Hello ${string}``.
```

Tipos de strings literais (string literals) – e uniões deles – podem ser usados na interpolação de tipos em vez do primitivo geral `string` para restringirmos os tipos template literals a padrões de strings mais rígidos. Os tipos template literals podem ser muito úteis para a descrição de strings que precisem ser iguais às de um conjunto restrito de strings

permitidas.

Aqui, `BrightnessAndColor` só aceita strings que comecem com um tipo de brilho (`Brightness`), terminem com uma cor (`Color`) e tenham um hífen (-) entre os dois:

```
type Brightness = "dark" | "light";
type Color = "blue" | "red";

type BrightnessAndColor = `${Brightness}-${Color}`;
// Equivalent to: "dark-red" | "light-red" | "dark-blue" | "light-blue"

let colorOk: BrightnessAndColor = "dark-blue"; // Ok

let colorWrongStart: BrightnessAndColor = "medium-blue";
// ~~~~~
// Error: Type '"medium-blue"' is not assignable to type
// '"dark-blue" | "dark-red" | "light-blue" | "light-red"'.

let colorWrongEnd: BrightnessAndColor = "light-green";
// ~~~~~
// Error: Type '"light-green"' is not assignable to type
// '"dark-blue" | "dark-red" | "light-blue" | "light-red"'.
```

Sem os tipos template literal teríamos de escrever laboriosamente todas as quatro combinações dos tipos `Brightness` e `Color`. Isso acabaria ficando confuso se adicionássemos mais strings literais a cada um deles!

O TypeScript permite que os tipos template literal contenham qualquer primitivo (exceto `symbol`) ou uma união deles: `string`, `number`, `bignum`, `boolean`, `null` ou `undefined`.

Este tipo `ExtolNumber` permite qualquer string que comece com "much ", inclua uma string que seja um número e termine com "wow":

```
type ExtolNumber = `much ${number} wow`;

function extol(extolee: ExtolNumber) { /* ... */ }

extol('much 0 wow'); // Ok
extol('much -7 wow'); // Ok
extol('much 9.001 wow'); // Ok

extol('much false wow');
// ~~~~~
```

```
// Error: Argument of type '"much false wow"' is not  
// assignable to parameter of type ``much ${number} wow``.
```

## Tipos intrínsecos de manipulação de strings

Para ajudar no trabalho com tipos string, o TypeScript fornece um pequeno conjunto de tipos utilitários genéricos intrínsecos (o que significa que eles são internos do TypeScript) que recebem uma string e aplicam alguma operação nela. A partir do TypeScript 4.7.2, existem quatro:

- `Uppercase`: Converte um tipo de string literal para letras maiúsculas.
- `Lowercase`: Converte um tipo de string literal para letras minúsculas.
- `Capitalize`: Converte o primeiro caractere de um tipo de string literal para letras maiúsculas.
- `Uncapitalize`: Converte o primeiro caractere de um tipo de string literal para letras minúsculas.

Todos eles podem ser usados como um tipo genérico que recebe uma string. Por exemplo, a seguir estamos usando `Capitalize` para capitalizar a primeira letra de uma string:

```
type FormalGreeting = Capitalize<"hello.">; // Type: "Hello."
```

Esses tipos intrínsecos de manipulação de strings podem ser muito úteis para a manipulação de chaves de propriedades em tipos de objeto.

## Chaves de template literals

Os tipos template literal são intermediários entre o primitivo `string` e as strings literais, o que significa que ainda são strings. Eles podem ser usados em qualquer local no qual você poderia usar strings literais.

Por exemplo, você pode usá-los como a assinatura de índice em um tipo mapeado. Este tipo `ExistenceChecks` tem uma chave para cada string de `DataKey`, mapeada com `check${Capitalize<DataKey>}`:

```
type DataKey = "location" | "name" | "year";  
  
type ExistenceChecks = {  
    [K in `check${Capitalize<DataKey>}`]: () => boolean;  
};
```

```

// Equivalent to:
// {
//   checkLocation: () => boolean;
//   checkName: () => boolean;
//   checkYear: () => boolean;
// }

function checkExistence(chcks: ExistenceChecks) {
  chcks.checkLocation(); // Type: boolean
  chcks.checkName(); // Type: boolean

  chcks.checkWrong();
  // ~~~~~
  // Error: Property 'checkWrong' does not exist on type 'ExistenceChecks'.
}

```

## Remapeamento de chaves de tipos mapeados

O TypeScript permite criar novas chaves para membros de tipos mapeados com base nos membros originais usando os tipos template literal. A inserção da palavra-chave `as` seguida de um tipo template literal para a assinatura de índice em um tipo mapeado altera as chaves de tipo resultantes para que fiquem de acordo com o tipo template literal. Isso permite que o tipo mapeado tenha uma chave diferente para cada propriedade mapeada, mantendo, ao mesmo tempo, a referência ao valor original.

Aqui, `DataEntryGetters` é um tipo mapeado cujas chaves são `getLocation`, `getName` e `getYear`. Cada chave é mapeada para uma nova chave com um tipo template literal. Cada valor mapeado é uma função cujo tipo de retorno é um `DataEntry` usando a chave `K` original como argumento de tipo:

```

interface DataEntry<T> {
  key: T;
  value: string;
}

type DataKey = "location" | "name" | "year";

type DataEntryGetters = {
  [K in DataKey as `get${Capitalize<K>}`]: () => DataEntry<K>;
};

```

```
// Equivalent to:
// {
//   getLocation: () => DataEntry<"location">;
//   getName: () => DataEntry<"name">;
//   getYear: () => DataEntry<"year">;
// }
```

Os remapeamentos de chaves podem ser combinados com outras operações com tipos para criar tipos mapeados que sejam baseados em formas de tipos existentes. Uma combinação interessante é usar `keyof` `typeof` em um objeto existente para criar um tipo mapeado a partir do tipo desse objeto.

O tipo `ConfigGetter` a seguir é baseado no tipo `config`, mas cada campo é uma função que retorna o config original e as chaves são modificadas e tornam-se diferentes da chave original:

```
const config = {
  location: "unknown",
  name: "anonymous",
  year: 0,
};

type LazyValues = {
  [K in keyof typeof config as `${K}Lazy`]: () => Promise<typeof config[K]>;
};

// Equivalent to:
// {
//   location: Promise<string>;
//   name: Promise<string>;
//   year: Promise<number>;
// }

async function withLazyValues(configGetter: LazyValues) {
  await configGetter.locationLazy; // Resultant type: string

  await configGetter.missingLazy();
  // ~~~~~
  // Error: Property 'missingLazy' does not exist on type 'LazyValues'.
};
```

É bom ressaltar que, em JavaScript, as chaves dos objetos podem ser de tipo `string` ou `symbol` – e as chaves `symbol` não podem ser usadas como tipos template literal porque não são primitivas. Se você tentar usar a chave

remapeada de um tipo template literal em um tipo genérico, o TypeScript emitirá um alerta informando que `symbol` não pode ser usado em um tipo template literal:

```
type TurnIntoGettersDirect<T> = {
  [K in keyof T as `get${K}`]: () => T[K]
  // ~~~~~
  // Error: Type 'keyof T' is not assignable to type
  // 'string | number | bigint | boolean | null | undefined'.
  // Type 'string | number | symbol' is not assignable to type
  // 'string | number | bigint | boolean | null | undefined'.
  // Type 'symbol' is not assignable to type
  // 'string | number | bigint | boolean | null | undefined'.
};

};
```

Para eliminar essa restrição, você pode usar um tipo de interseção `string &` para impor que somente tipos que possam ser strings sejam usados. Já que `string & symbol` resulta em `never`, a template string inteira será reduzida a `never` e o TypeScript a ignorará:

```
const someSymbol = Symbol("");

interface HasStringAndSymbol {
    StringKey: string;
    [someSymbol]: number;
}

type TurnIntoGetters<T> = {
    [K in keyof T as `get${string & K}`]: () => T[K]
};

type GettersJustString = TurnIntoGetters<HasStringAndSymbol>;
// Equivalent to:
// {
//     getStringKey: () => string;
// }
```

O comportamento do TypeScript de excluir tipos `never` em uniões se mostrou útil novamente!

# Operações com tipos e a complexidade

As operações com tipos descritas neste capítulo estão entre os recursos

mais poderosos e inovadores da tipagem em qualquer linguagem de programação atual. A maioria dos desenvolvedores ainda não está suficientemente familiarizada com eles para poder depurar erros em usos significativamente complexos. Ferramentas de desenvolvimento de padrão industrial, como os recursos de IDE que abordei no Capítulo 12, “Usando recursos do IDE”, geralmente não são criadas para a visualização de operações com tipos de várias camadas usadas umas com as outras.

Se você achar que precisa usar operações com tipos, por favor – para ajudar qualquer desenvolvedor que tenha de ler seu código, incluindo você futuramente – tente mantê-las em um nível mínimo se possível. Use nomes legíveis que ajudem os leitores a entender o código quando o percorrerem. Deixe comentários descritivos para qualquer coisa com as quais você acha que futuros leitores possam ter dificuldades.

*Depurar é duas vezes mais difícil do que escrever o código. Logo, se você escrever o código da maneira mais inteligente possível, por definição não será suficientemente inteligente para depurá-lo.*

BRIAN KERNIGHAN

## Resumo

Neste capítulo, você liberou o verdadeiro poder do TypeScript operando com tipos em sua tipagem:

- Uso de tipos mapeados para transformar tipos existentes em novos tipos.
- Introdução de lógica em operações com tipos condicionais.
- Saber como `never` interage com interseções, uniões, tipos condicionais e tipos mapeados.
- Representação de padrões de tipos string com o uso de tipos template literal.
- Combinanção de tipos template literal e tipos mapeados para modificar as chaves dos tipos.



Agora que você terminou de ler este capítulo, pratique o que

aprendeu em <https://learningtypescript.com/type-operations>.

*Se você se perder na tipagem, o que deve usar?  
Um tipo mapeado!<sup>2</sup>*

---

<sup>1</sup> N.T.: Original: *Conditionals, maps / With great power over types / comes great confusion*

<sup>2</sup> N.T.: Original: *When you're lost in the type system, what do you use? A mapped type!*

# Glossário

## *anotação de tipo*

Uma anotação após um nome usada para indicar seu tipo. É composta de : e do nome de um tipo.

### `any`

Tipo que pode ser usado em qualquer local e pode receber qualquer coisa. `any` pode agir como tipo universal, já que qualquer tipo pode ser fornecido para um local em que o tipo seja `any`. Provavelmente, na maioria dos casos, você vai querer usar `unknown` para ter uma segurança de tipo (type safety) mais precisa.

Consulte também `unknown`, tipo universal.

### `any` *implícito*

Quando o TypeScript não consegue deduzir imediatamente o tipo de uma propriedade de classe, de um parâmetro de função ou de uma variável, ele presume implicitamente que o tipo seja `any`. Os tipos `any` implícitos de propriedades de classes e parâmetros de funções podem ser configurados para ser erros de tipo com o uso da opção de compilador `noImplicitAny`.

### `any` *modificável*

Caso especial de `any` implícito para variáveis que não têm uma anotação de tipo ou um valor inicial. Seu tipo será modificado para aquele que elas geralmente usam.

Consulte também `any` *implícito*.

### *argumento*

Algo que é fornecido como entrada, usado para se referir a um valor que está sendo passado para uma função. Para as funções, um argumento é o valor que está sendo passado para uma chamada, enquanto um parâmetro é o valor existente dentro da função. Consulte também

*parâmetro.*

*argumento de tipo genérico, argumento de tipo*

Tipo fornecido como parâmetro de tipo para uma estrutura genérica.

*arquivo de declaração*

Arquivo com a extensão `.d.ts`. Os arquivos de declaração criam um contexto de ambiente, o que significa que eles só podem declarar tipos e não podem declarar implementações. Consulte também *contexto de ambiente*.

*asserção, assserção de tipo*

Uma indicação para o TypeScript de que um valor é de um tipo diferente do esperado.

*assserção const*

Abreviação da asserção de tipo `as const` que solicita ao TypeScript para usar a forma mais literal e somente leitura possível do tipo de um valor.

*asserção de não nulo*

Abreviatura com `!` que declara que um tipo não é `null` ou `undefined`.

*assinatura de chamada*

Descrição do sistema de tipos de como uma função pode ser chamada. Inclui uma lista de parâmetros e um tipo de retorno.

*assinatura de implementação*

Assinatura final declarada em uma função sobrecarregada, usada para os parâmetros de sua implementação. Consulte também *sobrecarga de funções*.

*assinatura de sobrecarga*

Uma das assinaturas declaradas em uma função sobrecarregada para descrever uma maneira dela ser chamada. Consulte também *sobrecarga de funções*.

*atribuível, capacidade de atribuição*

Se um tipo pode ser usado no lugar de outro.

*camel case*

Convenção de nomeação na qual a primeira letra de cada palavra após a primeira palavra em um nome é capitalizada, como em camelCase. É a convenção usada para nomes de membros em muitas estruturas da tipagem do TypeScript, incluindo membros de classes e interfaces.

### *classe*

Açúcar sintático JavaScript para funções que são atribuídas a um protótipo. O TypeScript permite trabalhar com classes JavaScript.

### *compilar*

Converter o código-fonte para outro formato. O TypeScript inclui um compilador que, além de fazer a verificação de tipos, converte o código-fonte TypeScript em arquivos JavaScript e/ou de declaração. Consulte também *transpilar*.

### *contexto de ambiente*

Área no código em que podemos declarar tipos, mas não implementações. Geralmente é usado em referência a arquivos de declaração *.d.ts*. Consulte também *arquivo de declaração*.

### *constituinte, tipo constituinte*

Um dos tipos de um tipo interseção ou união.

### *decorator*

Proposta experimental do JavaScript que permite anotar uma classe ou o membro de uma classe com uma função marcada por um símbolo `@`. Isso faria a função ser executada nessa classe ou no membro da classe no momento da criação.

### *DefinitelyTyped*

O imenso repositório de definições de tipos criadas pela comunidade para os pacotes (DT, na abreviação). Contém milhares de definições *.d.ts*, além da automação da verificação de propostas de alteração e da publicação de atualizações. Essas definições são publicadas como pacotes no escopo `@types/` do npm, como em `@types/react`.

### *dinamicamente tipado, tipagem dinâmica*

Classificação de linguagem de programação que não inclui nativamente

um verificador de tipos. Exemplos de linguagens dinamicamente tipadas são o JavaScript e o Ruby.

### *discriminante*

Membro de uma união discriminada que tem o mesmo nome, porém é de um tipo diferente em cada constituinte.

### *distributividade*

Propriedade dos tipos condicionais do TypeScript quando recebem tipos de template de união: seu tipo resultante será uma união da aplicação desse tipo condicional a cada um dos constituintes (tipos existentes no tipo de união). `ConditionalType<T | U>` é igual a `Conditional<T> | Conditional<U>`.

### *duck typed*

Termo comum para como o sistema de tipos do JavaScript se comporta. É originário da frase “se anda como um pato e faz quack como um pato, provavelmente é um pato”. Significa que o JavaScript permite que qualquer valor seja passado para qualquer local; se for solicitado a um objeto um membro que não existe, o resultado será `undefined`. Consulte também *estruturalmente tipado*.

### *enum*

Conjunto de valores literais armazenados em um objeto com um nome amigável para cada valor. Os enums são um raro exemplo de extensão de sintaxe específica do TypeScript para JavaScript vanilla.

### *erro de um bilhão de dólares*

Termo fácil de lembrar usado no setor empresarial para muitos sistemas de tipos que permitem que valores como `null` sejam usados em locais que demandem um tipo diferente. Foi criado por Tony Hoare em referência ao volumoso prejuízo que ele parece ter causado. Consulte também *verificação estrita de nulos*.

### *estreitamento de tipos*

Ocorre quando o TypeScript pode inferir um tipo mais específico para um valor dentro de um bloco de código que foi inserido em um guarda

de tipos.

### *estruturalmente tipado*

Sistema de tipos no qual qualquer valor que atenda a um tipo pode ser usado como instância desse tipo. Consulte também *duck typed*.

### *extensão de uma interface*

Quando uma interface declara que estende outra interface. Essa operação copia todas as propriedades da interface original na nova interface. Consulte também *interface*.

### *genérico*

Permitir que um tipo diferente seja substituído por uma estrutura sempre que um novo uso for criado para a estrutura. Classes, interfaces e apelidos (aliases) de tipo podem ser tornados genéricos.

### *gerar, código gerado*

A saída de um compilador, como os arquivos .js geralmente produzidos pela execução do `tsc`. As gerações de arquivo JavaScript e/ou de declaração do compilador TypeScript podem ser controladas por suas opções de compilador.

### *IDE, Ambiente de Desenvolvimento Integrado*

Programa que fornece ferramentas de desenvolvedor baseadas em um editor de texto para código-fonte. Geralmente, os IDEs vêm com depuradores, realce de sintaxe e plugins que geram alertas das linguagens de programação como erros de tipo. Este livro está usando o VS Code para seus exemplos de IDE, mas existem outros, como o Atom, Emacs, Vim, Visual Studio e WebStorm.

### *interface*

Conjunto nomeado de propriedades. O TypeScript saberá que um valor que foi declarado como de um tipo de interface específico terá as propriedades declaradas dessa interface.

### *interface derivada*

Interface que estende pelo menos mais uma interface, chamada de interface base. Essa operação copia todas as propriedades da interface

base na interface derivada.

### *JSDoc*

Padrão para comentários de bloco `/** ... */` que descrevem trechos de código como classes, funções e variáveis. Geralmente usado em projetos JavaScript para descrever concisamente os tipos.

### *literal*

Valor que é sabidamente uma instância distinta de um primitivo.

### *mesclagem de interfaces*

Propriedade das interfaces que faz com que, quando múltiplas interfaces de mesmo nome são declaradas no mesmo escopo, elas se combinem em uma única interface em vez de causar um type error sobre nomes conflitantes. Geralmente, é mais usada por criadores de definições para ampliar interfaces globais como `window`.

### *modo estrito*

Conjunto de opções do compilador que aumenta o nível de rigidez e o número de verificações que o verificador de tipos do TypeScript executa. Pode ser ativado para o `tsc` com a flag `--strict` e em arquivos TSConfiguration com `"strict": true` `compilerOption`.

### *módulo*

Arquivo que começa com uma instrução `export` ou `import`. Geralmente, são arquivos de código-fonte ou de pacotes `node_modules/`. Consulte também `script`.

### *namespace*

Estrutura antiga do TypeScript que cria um objeto globalmente disponível com conteúdos “exportados” que podem ser chamados como membros desse objeto. Os namespaces são um exemplo raro de extensão da sintaxe específica do TypeScript para JavaScript vanilla. Atualmente, eles são mais usados em arquivos de declaração `.d.ts`.

### `never`

Tipo do TypeScript que representa o tipo vazio: um tipo que não pode ter valores possíveis. Consulte também tipo `vazio`.

## `null`

Um dos dois tipos primitivos do JavaScript que representam ausência de valor. `null` representa uma ausência de valor intencional, enquanto `undefined` representa uma ausência de valor mais genérica. Consulte também `undefined`.

## *opcional*

Parâmetro de função, propriedade de classe, ou membro de um tipo interface ou de objeto que não precisa ser fornecido. Indicado pela inserção de um `?` após o seu nome, ou para parâmetros de função e propriedades de classe, alternativamente indicado pelo fornecimento de um valor padrão com um `=`.

## *parâmetro*

Uma entrada recebida, geralmente referenciando o que uma função declara. Para as funções, um argumento é o valor que está sendo passado para uma chamada, enquanto um parâmetro é o valor existente dentro da função. Consulte também *argumento*.

## *parâmetro de tipo genérico, parâmetro de tipo*

Tipo substituído por um genérico. Os parâmetros de tipo genéricos podem receber diferentes argumentos de tipo para cada instância da estrutura, mas permanecerão consistentes dentro dessa instância.

## *Pascal case*

Convenção de nomeação na qual a primeira letra de cada palavra que compõe um nome é maiúscula, como em PascalCase. É a convenção usada para os nomes de muitas estruturas do sistema de tipos do TypeScript, incluindo genéricos, interfaces, e aliases de tipo.

## *predicado de tipo*

Função com um tipo de retorno anotado para agir como um *type guard*. As funções de predicado de tipo retornam um valor `boolean` que indica se um valor é de um tipo.

## *primitivo*

Um tipo de dado imutável interno do JavaScript que não é um objeto.

São eles: `null`, `undefined`, `boolean`, `string`, `number`, `bignum` e `symbol`.

### *privacidade, campo privado*

Recurso do JavaScript no qual propriedades da classe cujos nomes comecem com `#` só podem ser acessados dentro dessa mesma classe.

### *propriedade de parâmetro*

Extensão de sintaxe do TypeScript para a declaração de uma propriedade atribuída a uma propriedade membro do mesmo tipo no começo de um construtor de classe.

### *refatoração*

Alteração no código que mantém intactos todos os seus comportamentos ou a maioria deles. O serviço de linguagem do TypeScript executa algumas refatorações no código-fonte quando solicitado, como mover linhas de código complexas para uma variável `const`.

### *referências de projeto*

Recurso dos arquivos de configuração do TypeScript no qual eles podem referenciar os projetos de outros arquivos de configuração como dependências. Isso permite usar o TypeScript como um coordenador de build para que ele imponha a árvore de dependências de um projeto.

### *Resolução de módulo*

Conjunto de etapas usadas para determinar para qual arquivo uma importação de módulo apontará. O compilador TypeScript pode receber essa especificação em sua opção de compilador `moduleResolution`.

### *Rick Roll*

Um meme da internet em que os usuários são obrigados a ouvir e/ou assistir a um vídeo musical do seminal clássico de Rick Astley “Never Gonna Give You Up”. Ocultei vários neste livro. Consulte também <https://oreil.ly/rickroll>.

### *script*

Qualquer arquivo de código-fonte que não é um módulo. Consulte também *módulo*.

## *sobrecarga de funções, função sobrecarregada*

Maneira de descrever uma função que pode ser chamada com conjuntos de parâmetros muito diferentes.

## *sobreescrivendo*

Redeclarar uma propriedade, que já exista na base, em um objeto de subclasse-interface derivada.

## *somente leitura*

Recurso da tipagem TypeScript no qual a inclusão da palavra-chave `readonly` na frente da propriedade de uma classe ou objeto indica que ela não pode ser reatribuída.

## *subclasse*

Classe que estende outra classe, chamada de classe base. Essa operação copia as propriedades do protótipo da classe base para o protótipo da classe filha.

## *target*

Opção de compilador do TypeScript para a especificação de quanto será necessário retroceder no suporte à sintaxe para o código JavaScript ser transpilado, como em "`es5`" ou "`es2017`". Embora o padrão de `target` seja "`es3`" por razões de compatibilidade regressiva, é aconselhável usar a sintaxe JavaScript mais recente possível, conforme a plataforma de destino, já que o suporte a recursos mais novos do JavaScript em ambientes mais antigos demanda a criação de mais código JavaScript.

## *Thenable*

Objeto JavaScript com um método `.then` que recebe até duas funções de callback e retorna outro Thenable. Geralmente, é implementado pela classe interna `Promise`, mas classes e objetos definidos pelo usuário também podem funcionar como um Thenable.

## *tipagem*

Conjunto de regras que permite que uma linguagem de programação saiba que tipos as estruturas de um programa podem ter.

## *tipo*

Conhecimento dos membros e recursos que estão disponíveis para um valor. Estes podem ser os tipos primitivos, como `string`, literais, como `123`, ou ter formas mais complexas, como as funções e objetos.

### *tipo condicional*

Tipo que pode ser um entre dois tipos possíveis, baseado em um tipo existente.

### *tipo de interseção*

Tipo que usa o operador `&` para indicar que ele tem todas as propriedades dos seus dois constituintes.

### *tipo de retorno*

Tipo que deve ser retornado por uma função. Se existirem várias instruções `return` com tipos diferentes na função, o tipo de retorno será uma união de todos os tipos possíveis. Se a função não puder retornar, ele será `never`.

### *tipo universal*

Tipo que pode representar qualquer tipo possível em um sistema. Consulte também `any`, `unknown`.

### *tipo vazio*

Tipo que não tem valores possíveis – é o conjunto de tipos vazio. Nenhum tipo é atribuível ao tipo vazio. O TypeScript fornece a palavra-chave `never` para indicar um tipo vazio. Consulte também `never`.

### *tipos mapeados*

Tipo que recebe outro tipo e executa alguma operação em cada membro desse tipo. Em outras palavras, ele faz o mapeamento de membros de um tipo para um novo conjunto de membros.

### *transpilar*

Termo para a compilação que converte o código-fonte de uma linguagem de programação legível por humanos para outra linguagem. O TypeScript inclui um compilador que converte código-fonte TypeScript `.ts/.tsx` para arquivos `.js`, o que às vezes é chamado de transpilação. Consulte também `compilar`.

## *TSConfig*

Arquivo de configuração JSON para TypeScript. Mais comumente chamado de `tsconfig.json`, ou, no padrão, `tsconfig.*.json`. Editores como o VS Code fazem a leitura em um arquivo `tsconfig.json` em um diretório para determinar opções de configuração do serviço de linguagem do TypeScript.

## *tupla*

Array de tamanho fixo no qual cada elemento recebe um tipo explícito. Por exemplo, `[number, string | undefined]` é uma tupla de tamanho dois na qual o primeiro elemento é de tipo `number` e o segundo é de tipo `string | undefined`.

## *type guard*

Lógica de runtime que é entendida no sistema de tipos como só permitindo a execução de determinada lógica se um valor for de um tipo específico.

## `undefined`

Um dos dois tipos primitivos do JavaScript que representa ausência de valor. `null` representa ausência de valor intencional, enquanto `undefined` representa uma ausência de valor mais genérica. Consulte também `null`.

## *união*

Tipo que descreve um valor que pode ter dois ou mais tipos possíveis. É representado pelo pipe (`|`) entre cada tipo possível.

## *união discriminada, união de tipos discriminada*

União de tipos na qual um existe um membro “discriminante” com o mesmo nome, mas com um valor diferente em cada tipo constituinte. A verificação do valor do discriminante atua como uma espécie de estreitamento de tipo.

## `unknown`

Conceito do TypeScript que representa o tipo universal. `unknown` não permite o acesso arbitrário a propriedades sem o estreitamento de tipo. Consulte também `any`, *tipo universal*.

### *variável global*

Variável que existe no escopo global, como `setTimeout` em ambientes como o de navegadores, do Deno e do Node.

### *verificação estrita de nulos*

Modo estrito do TypeScript no qual `null` e `undefined` não podem mais ser fornecidos para tipos que não os incluem explicitamente. Consulte também *erro de um bilhão de dólares*.

### *visibilidade*

Especificar se uma propriedade da classe ficará visível para códigos de fora da classe. Indicado antes da declaração da propriedade com as palavras-chave `public`, `protected` e `private`. A visibilidade e suas palavras-chave precedem a privacidade real de propriedades com `#` do JavaScript e só existem na tipagem do TypeScript. Consulte também *privacidade*.

### *void*

Tipo que indica a ausência de valor retornado por uma função e que é representado pela palavra-chave `void` no TypeScript. As funções são consideradas como retornando `void` quando não têm instruções `return` que retornem um valor.

# Sobre o autor

**Josh Goldberg** é um desenvolvedor front-end de Nova York apaixonado pelo movimento open source, por análise estática e pela web. Ele trabalha em tempo integral como mantenedor de códigos open source e faz contribuições regulares para o TypeScript e para projetos open source deste ecossistema, como o typescript-eslint e o TypeStat. Em trabalhos anteriores foi responsável pelo uso do TypeScript na Codecademy, ajudou a criar o curso Learn TypeScript e projetou aplicações rich client na Microsoft. Seus projetos vão da análise estática a metalinguagens para a recriação de retro games no navegador. Não podemos nos esquecer dos gatos.

# Colofão

O animal da capa de *Aprendendo Typescript* é uma jandaia-amarela (*Aratinga solstitialis*), um papagaio colorido nativo da região nordeste da América do Sul.

As jandaias-amarelas, também conhecidas como periquitos solares, são quase sempre amarelas com as pontas das asas verdes e a face e o peito amarelos. Elas são verde-oliva quando nascem, com as cores brilhantes se desenvolvendo gradualmente com o tempo tanto nos machos quanto nas fêmeas. São monogâmicas e as fêmeas põem de três a quatro ovos em uma ninhada com 23 a 27 dias de incubação. Sua dieta típica é composta de frutas, flores, sementes, nozes e insetos.

As jandaias-amarelas são populares como animais domésticos por causa de sua linda plumagem e personalidade cativante. Elas são aves curiosas, mas também podem ser muito barulhentas.

Muitos dos animais das capas da O'Reilly estão em perigo; todos são importantes para o mundo.

A ilustração da capa é de Karen Montgomery, baseada em uma antiga gravura linear de Zoology de George Shaw.