

# TypeScript Master

Se torne um verdadeiro especialista em TypeScript com nosso ebook abrangente e prático. Aprenda desde os conceitos básicos até as técnicas avançadas.

 by Luiz Santiago



A large, abstract graphic on the left side of the slide features a series of concentric, wavy bands in shades of purple and blue. The bands are composed of fine, horizontal lines that create a sense of depth and motion. The overall effect is reminiscent of a digital or futuristic interface.

# Introdução

Descubra os benefícios que o TypeScript pode trazer para o desenvolvimento de sua aplicação JavaScript. Prepare-se para uma jornada incrível!

# Recursos do TypeScript

## Tipos de dados

Explore a forte tipagem do TypeScript e como ela ajuda a prevenir erros de programação.

## Inferência de tipo

Entenda como o TypeScript é capaz de inferir o tipo das variáveis, facilitando a escrita de código.

## Modularidade

Saiba como organizar seu código em módulos independentes e reutilizáveis.

## Autocomplete e refatoração

Aproveite as vantagens das ferramentas de desenvolvimento para agilizar o processo de escrita e refatoração de código TypeScript.

## Compilação e transpilação

Compreenda todo o fluxo de compilação e transpilação do TypeScript para JavaScript.

## Suporte para ES6+

Aproveite as capacidades avançadas do ECMAScript 6 e versões posteriores no seu código TypeScript.

## Decorators e Mixins

Aprenda sobre os poderosos recursos de decorators e mixins do TypeScript.

# Tipo de Dados em TypeScript

O TypeScript é um superconjunto do JavaScript que adiciona suporte a tipos estáticos opcionais. Isso significa que você pode declarar o tipo de cada variável e função em seu código. Ter tipos estáticos permite que o TypeScript faça verificações de tipo durante a compilação, ajudando a evitar erros comuns durante o desenvolvimento.

Alguns dos tipos de dados disponíveis em TypeScript incluem:

- **number**: para valores numéricos, como inteiros e números de ponto flutuante. Por exemplo: 10, 3.14
- **string**: para valores de texto. Por exemplo: "Olá, mundo!"
- **boolean**: para valores verdadeiro ou falso. Por exemplo: true, false
- **array**: para coleções de valores do mesmo tipo. Por exemplo: [1, 2, 3, 4]
- **object**: para estruturas de dados complexas. Por exemplo: { nome: "João", idade: 25 }

Além desses tipos básicos, TypeScript também suporta tipos mais avançados, como uniões, interseções e tipos genéricos. As uniões permitem que uma variável possa ter diferentes tipos de valor, enquanto as interseções permitem combinar vários tipos em um único tipo. Já os tipos genéricos permitem criar funções e classes que possam trabalhar com diferentes tipos de dados de forma flexível.

Exemplo:

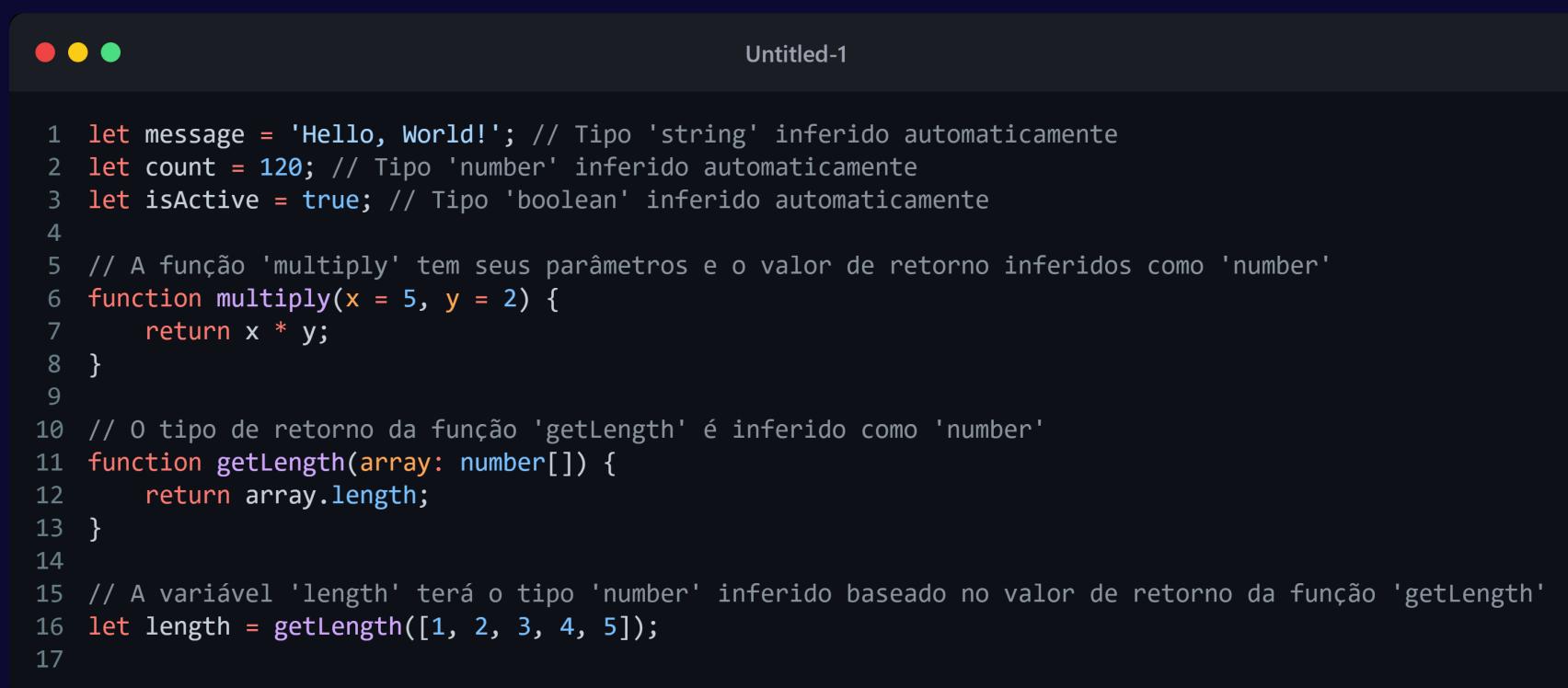
```
Untitled-1

1 // Exemplo de 'number'
2 let integer: number = 10;
3 let floatingPoint: number = 3.14;
4
5 // Exemplo de 'string'
6 let greeting: string = "Olá, mundo!";
7
8 // Exemplo de 'boolean'
9 let isCompleted: boolean = true;
10 let isValid: boolean = false;
11
12 // Exemplo de 'array'
13 let numbersArray: number[] = [1, 2, 3, 4];
14 let stringsArray: Array<string> = ["um", "dois", "três", "quatro"];
15
16 // Exemplo de 'object'
17 let person: { name: string; age: number } = {
18     name: "João",
19     age: 25
20 };
21
22 // Exemplo de 'união' (Union Type)
23 let id: number | string;
24 id = 10; // id pode ser um 'number'
25 id = "0020A"; // id também pode ser uma 'string'
26
27 // Exemplo de 'interseção' (Intersection Type)
28 type Worker = { name: string; employeeId: number };
29 type Manager = { stockOptions: number };
30 let executive: Worker & Manager = { // 'executive' deve ter todas as propriedades de 'Worker' e 'Manager'
31     name: "Maria",
32     employeeId: 1,
33     stockOptions: 100
34 };
35
36 // Exemplo de 'tipos genéricos' (Generic Types)
37 function identity<Type>(arg: Type): Type {
38     return arg;
39 }
40
41 let output1 = identity<string>("myString"); // output1 é do tipo 'string'
42 let output2 = identity<number>(100); // output2 é do tipo 'number'
43
```

# Interferências de tipo

- No TypeScript, a interferência de tipo ocorre quando você inicializa variáveis, define parâmetros de funções ou determina valores de retorno. O TypeScript é inteligente o suficiente para "inferir" o tipo de dado baseado no valor inicial que você atribui a uma variável. Aqui está um exemplo de interferência de tipo no TypeScript:
- No entanto, é importante ressaltar que as interferências de tipo não são sempre garantidas e pode haver casos em que é necessário fornecer explicitamente o tipo de uma variável.
- No geral, as interferências de tipo são uma ótima ferramenta no TypeScript para melhorar a legibilidade e manutenibilidade do código, além de proporcionar uma verificação de tipo mais rigorosa durante o processo de compilação.

Exemplo:

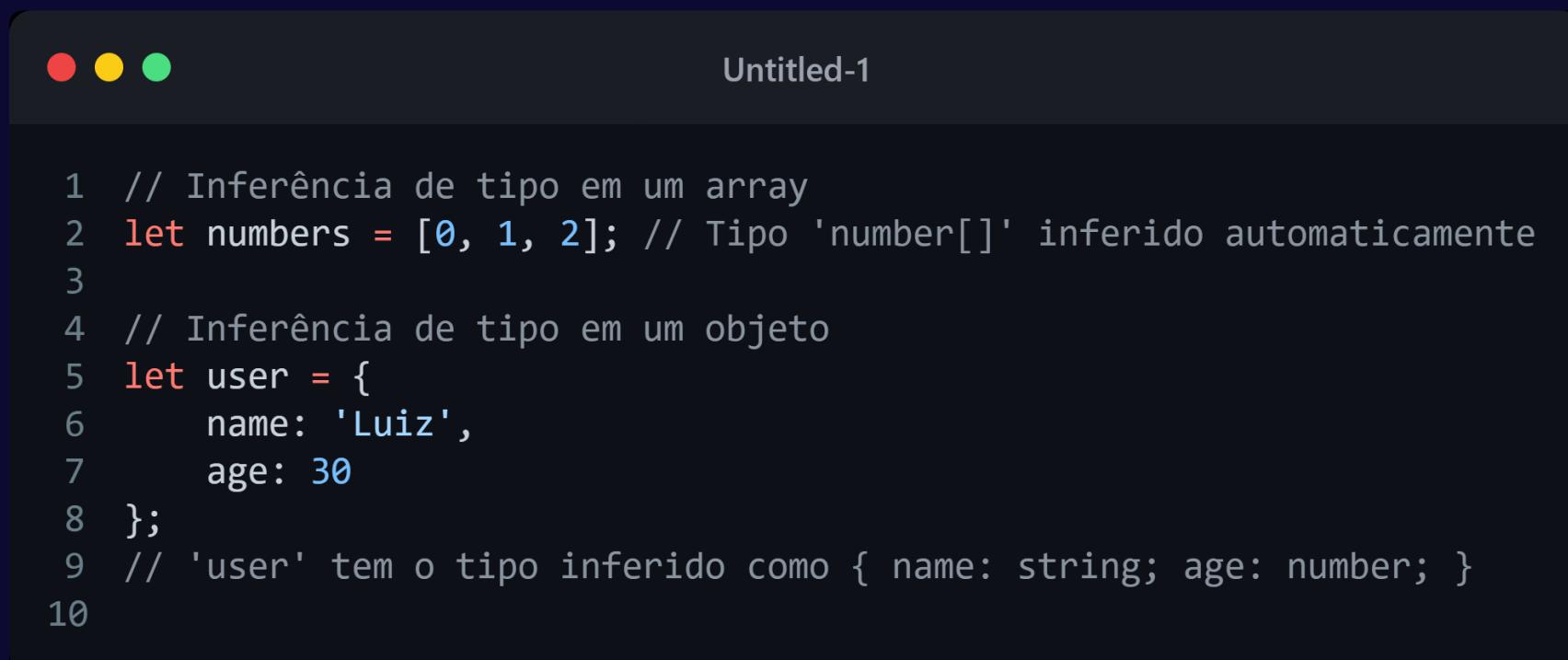


```
Untitled-1

1 let message = 'Hello, World!'; // Tipo 'string' inferido automaticamente
2 let count = 120; // Tipo 'number' inferido automaticamente
3 let isActive = true; // Tipo 'boolean' inferido automaticamente
4
5 // A função 'multiply' tem seus parâmetros e o valor de retorno inferidos como 'number'
6 function multiply(x = 5, y = 2) {
7     return x * y;
8 }
9
10 // O tipo de retorno da função 'getLength' é inferido como 'number'
11 function getLength(array: number[]) {
12     return array.length;
13 }
14
15 // A variável 'length' terá o tipo 'number' inferido baseado no valor de retorno da função 'getLength'
16 let length = getLength([1, 2, 3, 4, 5]);
17
```

No exemplo acima, o TypeScript infere o tipo de `message` como `string`, `count` como `number`, e `isActive` como `boolean` baseado nos valores que lhes foram atribuídos. Da mesma forma, para a função `multiply`, TypeScript infere que os parâmetros `x` e `y` e o valor de retorno são do tipo `number`.

Quando você usa arrays ou objetos, o TypeScript também é capaz de inferir os tipos de seus elementos ou propriedades:



```
Untitled-1

1 // Inferência de tipo em um array
2 let numbers = [0, 1, 2]; // Tipo 'number[]' inferido automaticamente
3
4 // Inferência de tipo em um objeto
5 let user = {
6     name: 'Luiz',
7     age: 30
8 };
9 // 'user' tem o tipo inferido como { name: string; age: number; }
10
```

# Modularidade

Modularidade é um conceito fundamental na engenharia de software, especialmente importante quando você está lidando com projetos de grande escala. Em TypeScript, modularidade refere-se à capacidade de organizar o código em pequenas unidades independentes, conhecidas como módulos, que podem ser desenvolvidas, testadas, depuradas e mantidas de forma isolada.

Como um programador sênior, gostaria de lhe passar a sabedoria de que "dividir para conquistar" não é apenas uma estratégia de batalha, mas também um princípio sólido de programação. Isso significa dividir um grande problema (ou aplicação) em partes menores e mais gerenciáveis que são mais fáceis de entender e manter.

## Vantagens da Modularidade

- **Manutenção:** Cada módulo é independente, então mudanças em um módulo específico são menos propensas a afetar outros.
- **Reusabilidade:** Módulos podem ser reutilizados em diferentes partes do seu projeto ou em projetos futuros.
- **Escopo:** Variáveis e funções definidas em um módulo não poluem o escopo global. Elas só são acessíveis onde você explicitamente as importa.

## Como Criar Módulos em TypeScript

Em TypeScript, cada arquivo é considerado um módulo quando contém pelo menos uma importação ou exportação. Aqui está um exemplo de como podemos criar um módulo para uma aplicação de e-commerce:

```
Untitled-1

1 // product.ts
2 export interface Product {
3   id: number;
4   name: string;
5   price: number;
6 }
7
8 export function calculateDiscount(price: number, discount: number): number {
9   return price * (1 - discount);
10}
11
```

No arquivo `product.ts`, definimos uma interface `Product` e uma função `calculateDiscount`. Estamos exportando ambos, o que os torna acessíveis em outros módulos que importam este.

Agora, vamos usar estes membros exportados em outro módulo:

```
Untitled-1

1 // cart.ts
2 import { Product, calculateDiscount } from './product';
3
4 export class Cart {
5   private items: Product[] = [];
6
7   addItem(item: Product) {
8     this.items.push(item);
9   }
10
11  calculateTotal(): number {
12    return this.items.reduce((total, item) => {
13      return total + calculateDiscount(item.price, 0.1); // supondo um desconto fixo de 10%
14    }, 0);
15  }
16}
17
```

No `cart.ts`, importamos a interface `Product` e a função `calculateDiscount` do módulo `product.ts`. Em seguida, criamos e exportamos uma classe `Cart` que utiliza essas importações.

## Importando Módulos

Você pode importar um módulo inteiro ou partes específicas de um módulo. Vamos dizer que temos um módulo `utils.ts` e queremos importar apenas uma função específica dele:

```
Untitled-1

1 // utils.ts
2 export function calculateTax(amount: number): number {
3   return amount * 0.15; // taxa hipotética de 15%
4 }
5
6 // order.ts
7 import { calculateTax } from './utils';
8
9 export function finalizeOrder(amount: number): number {
10   return amount + calculateTax(amount);
11 }
```

Aqui, apenas a função `calculateTax` foi importada e utilizada para calcular o valor final do pedido, demonstrando como você pode importar apenas o que é necessário.

# Autocomplete no TypeScript

Autocomplete, ou autocompletar, é uma característica das modernas ferramentas de desenvolvimento que facilita e acelera a escrita de código ao prever e completar automaticamente o que você está digitando. No TypeScript, isso é especialmente poderoso devido ao sistema de tipos do idioma.

Ao trabalhar com TypeScript, você se beneficia imediatamente do autocompletar devido ao tipo de análises realizadas pelo compilador. Por exemplo, se você tem uma interface ou tipo definido, o editor pode sugerir propriedades ou métodos assim que você começar a digitar. Isso não só economiza tempo, mas também reduz a possibilidade de erros de digitação e ajuda na descoberta de APIs.



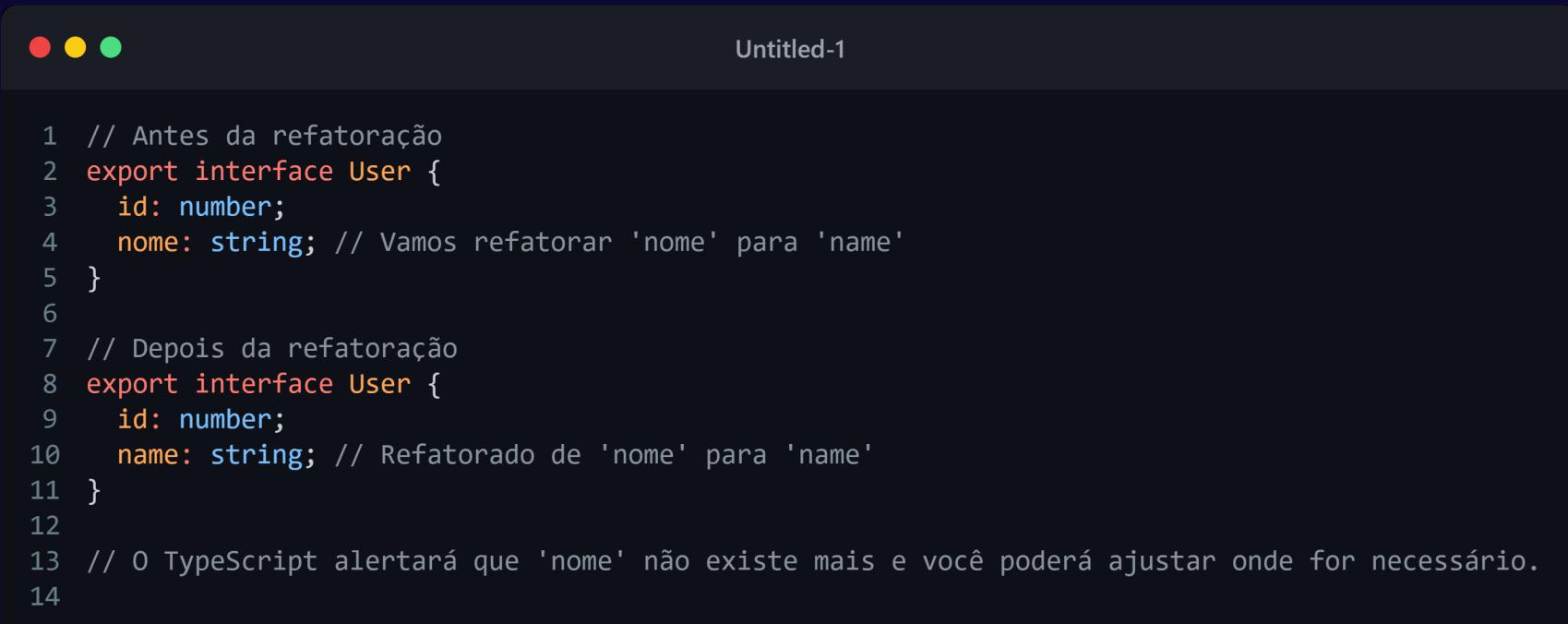
The screenshot shows a dark-themed code editor window titled "Untitled-1". In the code area, line 11 shows the completion of a variable name. The code is as follows:

```
1 // user.ts
2 export interface User {
3   id: number;
4   name: string;
5   login(): void;
6 }
7
8 // Ao instanciar um objeto 'User', o TypeScript sugere propriedades e métodos disponíveis.
9 let user: User = { id: 1, name: 'Luiz', login: () => /* ... */ };
10 user. // Ao digitar 'user.' o editor sugere 'id', 'name', e 'login'.
11
```

## Refatoração

Refatoração é o processo de reestruturar o código existente sem alterar seu comportamento externo. É uma prática essencial para manter o código limpo, legível e eficiente. No TypeScript, a refatoração é facilitada devido ao sistema de tipos estáticos.

Por exemplo, se você decidir mudar o nome de uma propriedade de uma interface, o TypeScript pode automaticamente atualizar todas as referências a essa propriedade em todo o código. Isso minimiza os riscos associados a essas alterações, pois o compilador alertará sobre qualquer parte do código que não esteja em conformidade com a nova estrutura.



The screenshot shows a dark-themed code editor window titled "Untitled-1". It illustrates a refactoring process across two code blocks. The first block shows the state before refactoring, and the second block shows it after. Lines 13 and 14 are comments explaining the TypeScript alert.

```
1 // Antes da refatoração
2 export interface User {
3   id: number;
4   nome: string; // Vamos refatorar 'nome' para 'name'
5 }
6
7 // Depois da refatoração
8 export interface User {
9   id: number;
10  name: string; // Refatorado de 'nome' para 'name'
11 }
12
13 // O TypeScript alertará que 'nome' não existe mais e você poderá ajustar onde for necessário.
14
```

A refatoração no TypeScript não se limita à renomeação. Você pode mover classes e interfaces para novos módulos, encapsular campos, converter tipos de dados e muito mais, tudo com o apoio do compilador que verifica a consistência do tipo em tempo real.

# Compilação e Transpilação no TypeScript

Compilação é o processo de transformar código escrito em uma linguagem de programação (o código fonte) em outra forma, geralmente código de máquina ou código em uma linguagem de nível mais baixo, para que possa ser executado por um computador. No entanto, no contexto de linguagens como TypeScript, o termo "transpilação" é frequentemente mais apropriado.

## Transpilação

Transpilação é um tipo específico de compilação onde o código fonte é convertido de uma linguagem para outra no mesmo nível de abstração. TypeScript é um superconjunto de JavaScript, o que significa que todo código JavaScript válido é também código TypeScript válido. O processo de transpilação no TypeScript envolve converter código TypeScript em JavaScript equivalente que pode ser executado em qualquer ambiente que suporte JavaScript.

O TypeScript adiciona tipos estáticos e recursos de linguagem que não estão presentes no JavaScript padrão. Durante a transpilação, o compilador TypeScript verifica o código quanto a erros de tipo e então produz um código JavaScript que reflete a lógica do código TypeScript original, mas sem tipos estáticos.

## Exemplo de Transpilação

Aqui está um exemplo simples de um código TypeScript e seu equivalente em JavaScript após a transpilação:



```
● ● ● Untitled-1

1 // TypeScript - Antes da transpilação
2 function greet(name: string): string {
3     return `Olá, ${name}!`;
4 }
5
6 const message: string = greet('Luiz');
7
```

Após a transpilação, o código acima pode se parecer com isso em JavaScript:



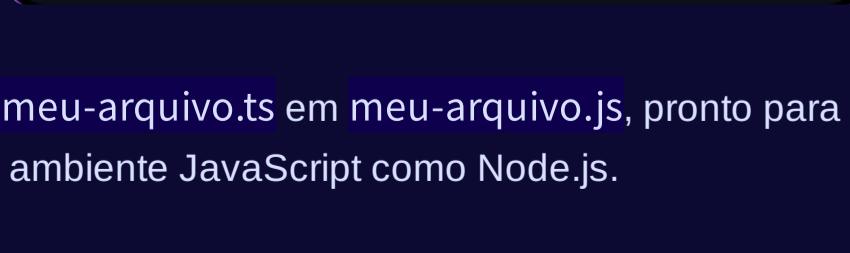
```
● ● ● Untitled-1

1 // JavaScript - Após a transpilação
2 function greet(name) {
3     return `Olá, ${name}!`;
4 }
5
6 const message = greet('Luiz');
7
```

Observe que os tipos foram removidos, pois o JavaScript não tem tipos estáticos.

## Processo de Compilação/Transpilação

Quando você compila um projeto TypeScript, você geralmente executará o compilador TypeScript (tsc) através de linha de comando ou um script de construção. O compilador verifica o código quanto a erros, e se tudo estiver correto, ele produz arquivos JavaScript correspondentes.



```
● ● ● Untitled-1

1 tsc meu-arquivo.ts
2
```

Este comando compila `meu-arquivo.ts` em `meu-arquivo.js`, pronto para ser executado em qualquer navegador ou ambiente JavaScript como Node.js.

Como programador é crucial entender a diferença entre compilação e transpilação. No desenvolvimento TypeScript, a transpilação é o processo-chave que permite escrever código em uma linguagem de alto nível com recursos poderosos e, em seguida, executá-lo em ambientes JavaScript comuns.

A habilidade de transpilar eficientemente seu código TypeScript para JavaScript é uma pedra angular para garantir que suas aplicações sejam portáveis e acessíveis em diversos ambientes e plataformas.

# Supor te para ES6+ no TypeScript

O TypeScript não só fortalece seu código JavaScript com tipos estáticos, mas também expande suas capacidades ao suportar recursos modernos do ECMAScript 6 (ES6) e posteriores. Isso significa que você pode utilizar as funcionalidades mais recentes do JavaScript enquanto beneficia-se do sistema de tipos do TypeScript.

## Recursos do ES6+ no TypeScript

Aqui estão alguns dos recursos do ES6+ que você pode usar no TypeScript:

- **Let e Const:** Controle mais refinado sobre a mutabilidade e escopo de variáveis.
- **Arrow Functions:** Sintaxe mais curta para funções e compartilhamento do contexto `this`.
- **Classes:** Uma maneira orientada a objetos de estruturar seu código.
- **Modules:** Importe e exporte funções, objetos ou tipos entre diferentes arquivos.
- **Spread Operator e Rest Parameters:** Trabalhe com elementos de forma mais flexível.
- **Destructuring:** Extraia elementos de arrays ou propriedades de objetos de forma mais legível.
- **Template Literals:** Crie strings com variáveis e expressões incorporadas.
- **Promises e Async/Await:** Melhore a escrita de código assíncrono.
- **Generators e Iterators:** Trabalhe com coleções de dados e fluxos de operações customizadas.

## Exemplo de Código

Como um desenvolvedor júnior, adotar esses recursos modernos pode aumentar a qualidade e a legibilidade do seu código. Aqui está um exemplo de como você pode integrar recursos ES6+ em seu código TypeScript:

```
Untitled-1

1 // Uso de let e const
2 let mutableValue = "Pode mudar";
3 const immutableValue = "Não pode mudar";
4
5 // Arrow function com template literal
6 const greet = (name: string) => `Olá, ${name}!`;
7
8 // Classes e módulos
9 export class Person {
10     constructor(public name: string, private age: number) {}
11
12     greet() {
13         console.log(greet(this.name));
14     }
15 }
16
17 // Promises e async/await
18 async function checkStatus(url: string): Promise<boolean> {
19     try {
20         const response = await fetch(url);
21         return response.ok;
22     } catch (error) {
23         console.error("Erro ao verificar o status:", error);
24         return false;
25     }
26 }
27
28 // Destructuring e rest parameters
29 function calculateScores(...scores: number[]) {
30     const [first, second, ...others] = scores.sort((a, b) => b - a);
31     console.log(`Primeiro: ${first}, Segundo: ${second}, Outros: ${others}`);
32 }
33
```

## Transpilação de ES6+ para ES5

Mesmo se você estiver visando ambientes que ainda não suportam ES6+, o TypeScript pode transpilar seu código para versões mais antigas do JavaScript, como ES5. Isso é feito facilmente através da configuração do `target` no seu arquivo `tsconfig.json`, permitindo que você escreva código moderno e ainda assim atinja ampla compatibilidade.

```
Untitled-1

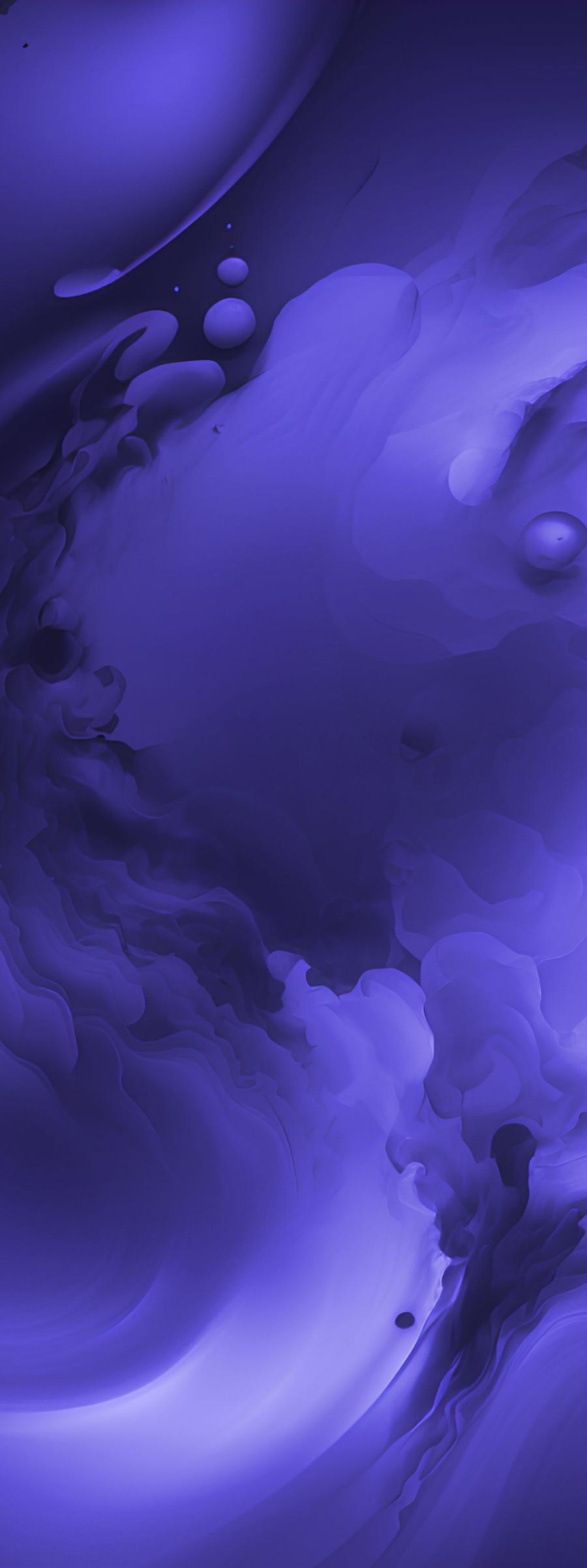
1 {
2     "compilerOptions": {
3         "target": "es5",
4         "module": "commonjs",
5         ...
6     }
7 }
```

Entender e utilizar os recursos do ES6+ no TypeScript pode transformar a maneira como você escreve e organiza seu código. Não se limite apenas aos recursos básicos do JavaScript; explore as possibilidades e melhore suas habilidades de desenvolvimento com essas poderosas ferramentas linguísticas. Como resultado, seu código será mais expressivo, mais fácil de manter e estará alinhado com as melhores práticas da moderna engenharia de software.

# Conceitos avançados para você pesquisar e evoluir



# Conclusão



Chegamos ao fim de nossa jornada exploratória pelo TypeScript, e espero que você esteja tão empolgado quanto eu estou com as possibilidades que essa poderosa linguagem traz. Ao longo deste ebook, mergulhamos nos fundamentos do TypeScript, desde tipos básicos até recursos avançados como genéricos e decoradores. Discutimos como o TypeScript facilita a escrita de código robusto e manutenível, graças ao seu sistema de tipos estáticos e suporte para as mais recentes funcionalidades do ECMAScript.

Como desenvolvedor, você está agora equipado para começar a construir aplicações complexas e de alto desempenho com maior confiança e eficiência. O TypeScript não é apenas uma ferramenta para prevenir erros; é um aliado na arquitetura de software, um mentor no design de código e um mapa que guia você através do complexo ecossistema do desenvolvimento moderno.

À medida que você continua a escrever mais TypeScript, lembre-se de que a aprendizagem é um processo contínuo. O ecossistema JavaScript está sempre evoluindo, e o TypeScript com ele. Mantenha-se curioso, continue praticando e não tenha medo de experimentar novas técnicas ou padrões de design.

Este ebook deve servir como um ponto de partida para sua aventura com TypeScript. Com os conceitos e exemplos que cobrimos aqui, você está bem preparado para se aprofundar ainda mais e explorar as vastas oportunidades que o TypeScript oferece. Seja estendendo seu conhecimento em tipagem avançada, contribuindo para projetos de código aberto, ou simplesmente construindo algo incrível, o futuro é brilhante para desenvolvedores TypeScript.

Obrigado por ter acompanhado até aqui. Agora, pegue o seu teclado e comece a tecer o seu próprio legado com TypeScript.