Tipos do Dia-a-Dia

Neste capítulo, abordaremos alguns dos tipos de valores mais comuns que você encontrará no código JavaScript e explicaremos as formas correspondentes de descrever esses tipos em TypeScript. Esta não é uma lista exaustiva, e capítulos futuros descreverão mais formas de nomear e usar outros tipos.

Os tipos também podem aparecer em muitos mais *lugares* do que apenas nas anotações de tipo. À medida que aprendemos sobre os tipos em si, também aprenderemos sobre os lugares onde podemos referenciar esses tipos para formar novos construtos.

Começaremos revisando os tipos mais básicos e comuns que você pode encontrar ao escrever código JavaScript ou TypeScript. Estes formarão mais tarde os blocos de construção fundamentais para tipos mais complexos.

Os primitivos: `string`, `number` e `boolean`

JavaScript tem três primitivos muito comuns: `string`, `number` e `boolean`. Cada um tem um tipo correspondente em TypeScript. Como você pode esperar, esses são os mesmos nomes que você veria se usasse o operador typeof do JavaScript em um valor desses tipos:

* `string` representa valores de string como `"Hello, world"`
* `number` é para números como `42`. O JavaScript não tem um valor de tempo de execução especial para inteiros, então não há equivalente a `int` ou `float` - tudo é simplesmente `number`
* `boolean` é para os dois valores `true` e `false`

> Os nomes de tipo String, Number e Boolean (com letras maiúsculas) são legais, mas se referem a alguns tipos internos especiais que raramente aparecerão no seu código. *Sempre* use string, number ou boolean para tipos.

Arrays

Para especificar o tipo de um array como `[1, 2, 3]`, você pode usar a sintaxe `number[]`; essa sintaxe funciona para qualquer tipo (por exemplo, `string[]` é um array de strings, e assim por diante). Você também pode ver isso escrito como `Array<number>`, que significa a mesma coisa. Aprenderemos mais sobre a sintaxe `T<U>` quando cobrirmos *generics*.

> Note que [number] é uma coisa diferente; consulte a seção sobre Tuples.

`any`

TypeScript também tem um tipo especial, `any`, que você pode usar sempre que não quiser que um valor específico cause erros de verificação de tipo.

Quando um valor é do tipo any, você pode acessar quaisquer propriedades dele (que serão, por sua vez, do tipo `any`), chamá-lo como uma função, atribuí-lo a (ou a partir de) um valor de qualquer tipo, ou praticamente qualquer outra coisa que seja sintaticamente legal:

```
let obj: any = { x: 0 };
// Nenhuma das linhas de código a seguir gerará erros de compilação.
// Usar `any` desativa toda a verificação de tipo subsequente, e assume-se
// que você conhece o ambiente melhor do que o TypeScript.
obj.foo();
```

```
obj();
obj.bar = 100;
obj = "hello";
const n: number = obj;
```

O tipo `any` é útil quando você não quer escrever um tipo longo apenas para convencer o TypeScript de que uma linha específica de código está ok.

`noImplicitAny`

Quando você não especifica um tipo, e o TypeScript não consegue inferi-lo a partir do contexto, o compilador geralmente assume `any` como padrão.

Você geralmente quer evitar isso, no entanto, porque `any` não é verificado quanto ao tipo. Use a flag do compilador noImplicitAny para sinalizar `any` implícito como um erro.

Anotações de Tipo em Variáveis

Quando você declara uma variável usando `const`, `var` ou `let`, você pode opcionalmente adicionar uma anotação de tipo para especificar explicitamente o tipo da variável:

```
let myName: string = "Alice";
```

> O TypeScript não usa declarações no estilo “tipos à esquerda”, como `int x = 0`; as anotações de tipo sempre vão *depois* da coisa que está sendo tipada.

Na maioria dos casos, no entanto, isso não é necessário. Sempre que possível, o TypeScript tenta *inferir* automaticamente os tipos no seu código. Por exemplo, o tipo de uma variável é inferido com base no tipo de seu inicializador:

```
// Nenhuma anotação de tipo necessária -- 'myName' inferido como tipo 'string'
let myName = "Alice";
```

Na maior parte, você não precisa aprender explicitamente as regras de inferência. Se você está começando, tente usar menos anotações de tipo do que você acha - você pode se surpreender com a quantidade de pouco que você precisa para o TypeScript entender completamente o que está acontecendo.

Funções

Funções são o meio principal de passar dados em JavaScript. O TypeScript permite que você especifique os tipos dos valores de entrada e saída das funções.

Anotações de Tipo de Parâmetro

Quando você declara uma função, pode adicionar anotações de tipo após cada parâmetro para declarar que tipos de parâmetros a função aceita. Anotações de tipo de parâmetro vão após o nome do parâmetro:

```
// Anotação de tipo do parâmetro
function greet(name: string) {
  console.log("Olá, " + name.toUpperCase() + "!!");
}
```

Quando um parâmetro tem uma anotação de tipo, os argumentos para essa função serão verificados:

```
// Seria um erro em tempo de execução se executado!
greet(42);
Argumento do tipo 'number' não é atribuível ao parâmetro do tipo 'string'.
```

> Mesmo que você não tenha anotações de tipo nos seus parâmetros, o TypeScript ainda verificará se você passou o número certo de argumentos.

Anotações de Tipo de Retorno

Você também pode adicionar anotações de tipo de retorno. As anotações de tipo de retorno aparecem após a lista de parâmetros:

```
function getFavoriteNumber(): number {
  return 26;
}
```

Assim como as anotações de tipo de variável, você geralmente não precisa de uma anotação de tipo de retorno porque o TypeScript inferirá o tipo de retorno da função com base em suas instruções `return`. A anotação de tipo no exemplo acima não muda nada. Alguns códigos explicitarão um tipo de retorno para fins de documentação, para evitar mudanças acidentais, ou apenas por preferência pessoal.

Funções Que Retornam Promises

Se você quiser anotar o tipo de retorno de uma função que retorna uma promessa, você deve usar o tipo `Promise`:

```
async function getFavoriteNumber(): Promise<number> {
  return 26;
}
```

Funções Anônimas

Funções anônimas são um pouco diferentes das declarações de função. Quando uma função aparece em um lugar onde o TypeScript pode determinar como ela será chamada, os parâmetros dessa função recebem tipos automaticamente.

Aqui está um exemplo:

```
const names = ["Alice", "Bob", "Eve"];

// Tipagem contextual para função - parâmetro s inferido como tipo string
names.forEach(function (s) {
  console.log(s.toUpperCase());
});

// A tipagem contextual também se aplica a funções de seta
names.forEach((s) => {
  console.log(s.toUpperCase());
});
```

Embora o parâmetro `s` não tenha uma anotação de tipo, o TypeScript usou os tipos da função `forEach`, juntamente com o tipo inferido do array, para determinar o tipo que `s` terá.

Esse processo é chamado de *tipagem contextual* porque o *contexto* em que a função ocorre informa qual tipo ela deve ter.

Semelhante às regras de inferência, você não precisa aprender explicitamente como isso acontece, mas entender que *isso* acontece pode ajudá-lo a notar quando as anotações de tipo não são necessárias. Mais adiante, veremos mais exemplos de como o contexto em que um valor ocorre pode afetar seu tipo.

Tipos de Objetos

Além dos primitivos, o tipo mais comum que você encontrará é um *tipo de objeto*. Isso se refere a qualquer valor JavaScript com propriedades, que é quase todos eles! Para definir um tipo de objeto, simplesmente listamos suas propriedades e seus tipos.

Por exemplo, aqui está uma função que aceita um objeto semelhante a um ponto:

```
// A anotação de tipo do parâmetro é um tipo de objeto
function printCoord(pt: { x: number; y: number }) {
  console.log("O valor de x da coordenada é " + pt.x);
  console.log("O valor de y da coordenada é " + pt.y);
}
printCoord({ x: 3, y: 7 });
```

Aqui, anotamos o parâmetro com um tipo com duas propriedades - `x` e `y` - que são ambas do tipo `number`. Você pode usar `,` ou `;` para separar as propriedades, e o último separador é opcional de qualquer forma.

A parte do tipo de cada propriedade também é opcional. Se você não especificar um tipo, será assumido como `any`.

Propriedades Opcionais

Os tipos de objeto também podem especificar que algumas ou todas as suas propriedades são *opcionais*. Para fazer isso, adicione um `?` após o nome da propriedade:

```
function printName(obj: { first: string; last?: string }) {
  // ...
}
// Ambos estão OK
printName({ first: "Bob" });
printName({ first: "Alice", last: "Alisson" });
```

No JavaScript, se você acessar uma propriedade que não existe, você obterá o valor `undefined` em vez de um erro em tempo de execução. Por causa disso, quando você *lê* de uma propriedade opcional, terá que verificar se é `undefined` antes de usá-la.

```
function printName(obj: { first: string; last?: string }) {
  // Erro - pode falhar se 'obj.last' não foi fornecido!
  console.log(obj.last.toUpperCase());
  // 'obj.last' pode ser 'undefined'.
  if (obj.last !== undefined) {
    // OK
    console.log(obj.last.toUpperCase());
  }

  // Uma alternativa segura usando a sintaxe moderna do JavaScript:
  console.log(obj.last?.toUpperCase());
}
```

Tipos de União

O sistema de tipos do TypeScript permite que você construa novos tipos a partir de tipos existentes usando uma grande variedade de operadores. Agora que sabemos como escrever alguns tipos, é hora de começar a *combiná-los* de maneiras interessantes.

Definindo um Tipo de União

A primeira maneira de combinar tipos que você pode ver é um *tipo de união*. Um tipo de união é um tipo formado a partir de dois ou mais tipos, representando valores que podem ser *qualquer* um desses tipos. Chamamos cada um desses tipos de *membros* da união.

Vamos escrever uma função que pode operar em strings ou números:

```
function printId(id: number | string) {
  console.log("Seu ID é: " + id);
}
// OK
printId(101);
// OK
printId("202");
// Erro
printId({ myID: 22342 });
Argumento do tipo '{ myID: number; }' não é atribuível ao parâmetro do tipo 'string |
number'.
```

Trabalhando com Tipos de União

É fácil *fornecer* um valor que corresponde a um tipo de união - basta fornecer um tipo que corresponda a qualquer um dos membros da união. Mas se você *tem* um valor de um tipo de união, como você trabalha com ele?

O TypeScript só permitirá uma operação se ela for válida para *todos* os membros da união. Por exemplo, se você tem a união `string | number`, não pode usar métodos que estão disponíveis apenas para `string`:

```
function printId(id: number | string) {
  console.log(id.toUpperCase());
  // Propriedade 'toUpperCase' não existe no tipo 'string | number'.
  // Propriedade 'toUpperCase' não existe no tipo 'number'.
}
```

A solução é *reduzir* a união com código, da mesma forma que você faria no JavaScript sem anotações de tipo. *Reduzir* ocorre quando o TypeScript pode deduzir um tipo mais específico para um valor com base na estrutura do código.

Por exemplo, o TypeScript sabe que somente um valor `string` terá um valor `typeof` igual a `"string"`:

```
function printId(id: number | string) {
  if (typeof id === "string") {
    // Neste ramo, id é do tipo 'string'
    console.log(id.toUpperCase());
  } else {
    // Aqui, id é do tipo 'number'
    console.log(id);
```

```
  }
}
```

Outro exemplo é usar uma função como `Array.isArray`:

```
function welcomePeople(x: string[] | string) {
  if (Array.isArray(x)) {
    // Aqui: 'x' é 'string[]'
    console.log("Olá, " + x.join(" e "));
  } else {
    // Aqui: 'x' é 'string'
    console.log("Bem-vindo viajante solitário " + x);
  }
}
```

Note que no ramo `else`, não precisamos fazer nada de especial - se `x` não era um `string[]`, então deve ter sido um `string`.

Às vezes, você terá uma união onde todos os membros têm algo em comum. Por exemplo, tanto arrays quanto strings têm um método `slice`. Se cada membro de uma união tem uma propriedade em comum, você pode usar essa propriedade sem reduzir:

```
// Tipo de retorno inferido como number[] | string
function getFirstThree(x: number[] | string) {
  return x.slice(0, 3);
}
```

> Pode ser confuso que uma *união* de tipos parece ter a *interseção* das propriedades desses tipos. Isso não é um acidente - o nome *união* vem da teoria dos tipos. A *união* `number | string` é composta pela união *dos valores* de cada tipo. Note que, dado dois conjuntos com fatos correspondentes sobre cada conjunto, apenas a *interseção* desses fatos se aplica à *união* dos próprios conjuntos. Por exemplo, se tivermos uma sala de pessoas altas usando chapéus, e outra sala de falantes de espanhol usando chapéus, depois de combinar essas salas, a única coisa que sabemos sobre *cada* pessoa é que elas devem estar usando um chapéu.

Aliases de Tipo

Temos usado tipos de objeto e tipos de união escrevendo-os diretamente nas anotações de tipo. Isso é conveniente, mas é comum querer usar o mesmo tipo mais de uma vez e se referir a ele por um único nome.

Um *alias de tipo* é exatamente isso - um *nome* para qualquer *tipo*. A sintaxe para um alias de tipo é:

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

// Exatamente o mesmo que o exemplo anterior
function printCoord(pt: Point) {
  console.log("O valor de x da coordenada é " + pt.x);
  console.log("O valor de y da coordenada é " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

Na verdade, você pode usar um alias de tipo para dar um nome a qualquer tipo, não apenas a um tipo de objeto. Por exemplo, um alias de tipo pode nomear um tipo de união:

```
type ID = number | string;
```

Note que aliases são *somente* aliases - você não pode usar aliases de tipo para criar "versões" diferentes/distintas do mesmo tipo. Quando você usa o alias, é exatamente como se você tivesse escrito o tipo aliasado. Em outras palavras, este código pode *parecer* ilegal, mas está OK de acordo com o TypeScript porque ambos os tipos são aliases para o mesmo tipo:

```
type UserInputSanitizedString = string;

function sanitizeInput(str: string): UserInputSanitizedString {
  return sanitize(str);
}

// Crie uma entrada sanitizada
let userInput = sanitizeInput(getInput());

// Ainda pode ser reatribuído com uma string, no entanto
userInput = "new input";
```

Interfaces

Uma *declaração de interface* é outra forma de nomear um tipo de objeto:

Exemplo com Interface

```
interface Point {
  x: number;
  y: number;
}

function printCoord(pt: Point) {
  console.log("O valor de x da coordenada é " + pt.x);
  console.log("O valor de y da coordenada é " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

Assim como no exemplo com um alias de tipo, o TypeScript não se importa com o nome da interface quando você passa um valor para `printCoord`. O que importa é a *estrutura* do valor - ele precisa ter as propriedades esperadas. É por isso que chamamos o TypeScript de sistema de tipos *estrutural*.

Diferenças Entre Alias de Tipo e Interface

Embora `type` e `interface` sejam bastante similares e muitos dos recursos de uma `interface` estejam disponíveis em `type`, há algumas diferenças importantes:

Interface

```
interface Animal {
  name: string;
}

interface Bear extends Animal {
```

```
  honey: boolean;
}

const bear = getBear();
bear.name;
bear.honey;
```

Tipo

```
type Animal = {
  name: string;
}

type Bear = Animal & {
  honey: boolean;
}

const bear = getBear();
bear.name;
bear.honey;
```

Adicionando Novos Campos a uma Interface Existente

```
interface Window {
  title: string;
}

interface Window {
  ts: TypeScriptAPI;
}

const src = 'const a = "Hello World"';
window.ts.transpileModule(src, {});
```

Um Tipo Não Pode Ser Alterado Após Ser Criado

```
type Window = {
  title: string;
}

type Window = {
  ts: TypeScriptAPI;
}

// Erro: Identificador duplicado 'Window'.
```

Você aprenderá mais sobre esses conceitos em capítulos futuros, então não se preocupe se não entender todos esses detalhes imediatamente.

* **Mensagens de Erro**: Antes da versão 4.2 do TypeScript, os nomes dos aliases de tipo podem aparecer nas mensagens de erro, às vezes no lugar do tipo anônimo equivalente (o que pode ou não ser desejável). Interfaces sempre serão nomeadas nas mensagens de erro.

* **Mesclagem de Declarações**: Aliases de tipo podem não participar da mesclagem de declarações, mas interfaces podem.

* **Renomear Primitivos**: Interfaces só podem ser usadas para declarar formas de objetos, não para renomear primitivos.
* **Nomes de Interface**: Nomes de interfaces *sempre* aparecem em sua forma original nas mensagens de erro, mas *somente* quando são usadas pelo nome.
* **Performance de Compilação**: Usar interfaces com `extends` [pode ser

mais eficiente para o compilador](https://github.com/microsoft/TypeScript/wiki/Performance#preferring-interfaces-over-intersections) do que usar aliases de tipo com interseções.

Na maior parte dos casos, você pode escolher com base na preferência pessoal e o TypeScript informará se é necessário usar o outro tipo de declaração. Se você precisar de uma diretriz, use `interface` até precisar de recursos específicos de `type`.

Aserções de Tipo

Às vezes, você tem informações sobre o tipo de um valor que o TypeScript não consegue inferir automaticamente.

Por exemplo, se você estiver usando `document.getElementById`, o TypeScript só sabe que isso retornará algum tipo de `HTMLElement`, mas você pode saber que sua página sempre terá um `HTMLCanvasElement` com um ID específico.

Nesse caso, você pode usar uma *asserção de tipo* para especificar um tipo mais específico:

```
const myCanvas = document.getElementById("main_canvas") as HTMLCanvasElement;
```

Assim como uma anotação de tipo, asserções de tipo são removidas pelo compilador e não afetam o comportamento de tempo de execução do seu código.

Você também pode usar a sintaxe de colchetes angulares (exceto se o código estiver em um arquivo `.tsx`), que é equivalente:

```
const myCanvas = <HTMLCanvasElement>document.getElementById("main_canvas");
```

> Lembre-se: Como asserções de tipo são removidas na compilação, não há verificação em tempo de execução associada a uma asserção de tipo. Não haverá exceção ou `null` gerado se a asserção de tipo estiver errada.

O TypeScript só permite asserções de tipo que convertem para uma versão *mais específica* ou *menos específica* de um tipo. Essa regra evita coerções "impossíveis" como:

```
const x = "hello" as number;
// Conversão do tipo 'string' para o tipo 'number' pode ser um erro porque nenhum dos tipos se sobrepõe ao outro.
```

Às vezes, essa regra pode ser muito conservadora e impedir coerções mais complexas que podem ser válidas. Se isso acontecer, você pode usar duas asserções, primeiro para `any` (ou `unknown`, que será introduzido mais tarde), e então para o tipo desejado:

```
const a = expr as any as T;
```

Tipos Literais

Além dos tipos gerais `string` e `number`, podemos nos referir a *strings* e *números* específicos em posições de tipo.

Uma maneira de pensar sobre isso é considerar como o JavaScript vem com diferentes maneiras de declarar uma variável. Tanto `var` quanto `let` permitem mudar o que é armazenado dentro da variável, e `const` não. Isso é refletido em como o TypeScript cria tipos para literais.

```
let changingString = "Hello World";
changingString = "Olá Mundo";
// Como `changingString` pode representar qualquer string possível, esse é o tipo
descrito pelo TypeScript no sistema de tipos
changingString;

let changingString: string;

const constantString = "Hello World";
// Como `constantString` só pode representar 1 string possível, ele tem uma
representação de tipo literal
constantString;

const constantString: "Hello World";
```

Por si só, os tipos literais não são muito valiosos:

```
let x: "hello" = "hello";
// OK
x = "hello";
// ...
x = "howdy";
// Tipo '"howdy"' não é atribuível ao tipo '"hello"'.
```

Não adianta muito ter uma variável que só pode ter um valor!

Mas ao *combinar* literais em uniões, você pode expressar um conceito muito mais útil - por exemplo, funções que aceitam apenas um conjunto de valores conhecidos:

```
function printText(s: string, alignment: "left" | "right" | "center") {
  // ...
}
printText("Hello, world", "left");
printText("G'day, mate", "centre");
Argument of type '"centre"' is not assignable to parameter of type '"left" | "right"
| "center"'.
```

Tipos literais numéricos funcionam da mesma maneira:

```
function compare(a: string, b: string): -1 | 0 | 1 {
  return a === b ? 0 : a > b ? 1 : -1;
}
```

Claro, você pode combinar esses com tipos não literais:

```
interface Options {
  width: number;
}
```

```
function configure(x: Options | "auto") {
  // ...
}
configure({ width: 100 });
configure("auto");
configure("automatic");
Argument of type '"automatic"' is not assignable to parameter of type 'Options |
"auto"'.
```

Há um tipo literal mais: literais booleanos. Existem apenas dois tipos literais booleanos, e como você pode adivinhar, são os tipos `true` e `false`. O tipo `boolean` em si é na verdade apenas um alias para a união `true | false`.

Inferência Literal

Quando você inicializa uma variável com um objeto, o TypeScript assume que as propriedades desse objeto podem mudar de valor mais tarde. Por exemplo, se você escrever um código assim:

```
const obj = { counter: 0 };
if (someCondition) {
  obj.counter = 1;
}
```

O TypeScript não assume que a atribuição de `1` a um campo que tinha anteriormente `0` é um erro. Outra maneira de dizer isso é que `obj.counter` deve ter o tipo `number`, não `0`, porque os tipos são usados para determinar tanto o comportamento de *leitura* quanto o de *escrita*.

O mesmo se aplica a strings:

```
declare function handleRequest(url: string, method: "GET" | "POST"): void;

const req = { url: "https://example.com", method: "GET" };
handleRequest(req.url, req.method);
Argument of type 'string' is not assignable to parameter of type '"GET" | "POST"'.
```

No exemplo acima, `req.method` é inferido como `string`, não `"GET"`. Como o código pode ser avaliado entre a criação de `req` e a chamada de `handleRequest`, o que poderia atribuir uma nova string como `"GUESS"` a `req.method`, o TypeScript considera que esse código tem um erro.

Existem duas maneiras de contornar isso.

1. Você pode alterar a inferência adicionando uma asserção de tipo em qualquer uma das localizações:

```
// Mudança 1:
const req = { url: "https://example.com", method: "GET" as "GET" };
// Mudança 2
handleRequest(req.url, req.method as "GET");
```

A Mudança 1 significa “Eu pretendo que `req.method` sempre tenha o *tipo literal* `"GET"`”, evitando a possível atribuição de `"GUESS"` a esse campo depois. A Mudança 2 significa “Eu sei por outros motivos que `req.method` tem o valor `"GET"`“.

2. Você pode usar `as const` para converter o objeto inteiro para tipos literais:

```
const req = { url: "https://example.com", method: "GET" } as const;
handleRequest(req.url, req.method);
```

O sufixo `as const` atua como `const` mas para o sistema de tipos, garantindo que todas as propriedades sejam atribuídas ao tipo literal em vez de uma versão mais geral como `string` ou `number`.

`null` e `undefined`

JavaScript tem dois valores primitivos usados para sinalizar valor ausente ou não inicializado: `null` e `undefined`.

O TypeScript tem dois *tipos* correspondentes com os mesmos nomes. Como esses tipos se comportam depende de você ter a opção strictNullChecks ativada.

strictNullChecks desativado

Com strictNullChecks *desativado*, valores que podem ser `null` ou `undefined` ainda podem ser acessados normalmente, e os valores `null` e `undefined` podem ser atribuídos a uma propriedade de qualquer tipo. Isso é semelhante ao comportamento de linguagens sem verificações de nulo (por exemplo, C#, Java). A falta de verificação para esses valores tende a ser uma grande fonte de erros; sempre recomendamos que as pessoas ativem strictNullChecks se for prático fazê-lo em seu código.

strictNullChecks ativado

Com strictNullChecks *ativado*, quando um valor é `null` ou `undefined`, você precisará testar esses valores antes de usar métodos ou propriedades nesse valor. Assim como verificar `undefined` antes de usar uma propriedade opcional, podemos usar *narrowing* para verificar valores que podem ser `null`:

```
function doSomething(x: string | null) {
  if (x === null) {
    // não faz nada
  } else {
    console.log("Olá, " + x.toUpperCase());
  }
}
```

Operador de Asserção Não-Nulo (Postfix `!`)

O TypeScript também tem uma sintaxe especial para remover `null` e `undefined` de um tipo sem fazer nenhuma verificação explícita. Escrever `!` após qualquer expressão é efetivamente uma asserção de tipo de que o valor não é `null` ou `undefined`:

```
function liveDangerously(x?: number | null) {
  // Sem erro
  console.log(x!.toFixed());
}
```

Assim como outras asserções de tipo, isso não altera o comportamento em tempo de execução do seu código, então é importante usar `!` apenas quando você sabe que o valor *não pode ser* nulo ou indefinido.

Enums

Enums são um recurso adicionado ao JavaScript pelo TypeScript que permite descrever um valor que pode ser um de um conjunto de constantes nomeadas. Diferente da maioria dos recursos do TypeScript, isso *não* é uma adição ao nível de tipo do JavaScript, mas algo adicionado à linguagem e ao runtime. Por

causa disso, é um recurso que você deve saber que existe, mas talvez evitar usar a menos que esteja certo. Você pode ler mais sobre enums na página de referência de Enums.

Primitivos Menos Comuns

Vale mencionar o resto dos primitivos em JavaScript que são representados no sistema de tipos. Embora não entraremos em detalhes aqui.

bigint

A partir do ES2020, há um primitivo em JavaScript usado para inteiros muito grandes, `BigInt`:

```
// Criando um bigint via a função BigInt
const oneHundred: bigint = BigInt(100);

// Criando um BigInt via a sintaxe literal
const anotherHundred: bigint = 100n;
```

Você pode aprender mais sobre BigInt nas notas de lançamento do TypeScript 3.2.

symbol

Há um primitivo em JavaScript usado para criar uma referência globalmente única via a função `Symbol()`:

```
const firstName = Symbol("name");
const secondName = Symbol("name");

if (firstName === secondName) {
  // Isso nunca acontecerá
}
```

Você pode aprender mais sobre eles na página de referência de Symbols.