Tipos de Objetos

No JavaScript, a forma fundamental de agrupar e passar dados é através de objetos. No TypeScript, representamos isso através de *tipos de objetos*.

Como vimos, eles podem ser anônimos:

```
function greet(person: { name: string; age: number }) {
  return "Hello " + person.name;
}
```

ou podem ser nomeados usando uma interface:

```
interface Person {
  name: string;
  age: number;
}

function greet(person: Person) {
  return "Hello " + person.name;
}
```

ou um alias de tipo:

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

function greet(person: Person) {
  return "Hello " + person.name;
}
```

Em todos os três exemplos acima, escrevemos funções que aceitam objetos que contêm a propriedade `name` (que deve ser um `string`) e `age` (que deve ser um `number`).

Referência Rápida

Temos folhas de dicas disponíveis para tipos e interfaces, se você quiser um rápido olhar sobre a sintaxe importante do dia a dia.

Modificadores de Propriedade

Cada propriedade em um tipo de objeto pode especificar algumas coisas: o tipo, se a propriedade é opcional e se a propriedade pode ser escrita.

Propriedades Opcionais

Na maior parte do tempo, nos deparamos com objetos que *podem* ter uma propriedade definida. Nesses casos, podemos marcar essas propriedades como *opcionais* adicionando um ponto de interrogação (`?`) ao final de seus nomes.

```
interface PaintOptions {
  shape: Shape;
  xPos?: number;
  yPos?: number;
```

```
}

function paintShape(opts: PaintOptions) {
  // ...
}

const shape = getShape();
paintShape({ shape });
paintShape({ shape, xPos: 100 });
paintShape({ shape, yPos: 100 });
paintShape({ shape, xPos: 100, yPos: 100 });
```

Neste exemplo, tanto `xPos` quanto `yPos` são considerados opcionais. Podemos escolher fornecer qualquer um deles, então todas as chamadas acima para `paintShape` são válidas. A opcionalidade realmente diz que, se a propriedade *estiver* definida, ela deve ter um tipo específico.

Também podemos ler dessas propriedades — mas quando fazemos isso sob strictNullChecks, o TypeScript nos informa que elas são potencialmente `undefined`.

```
function paintShape(opts: PaintOptions) {
  let xPos = opts.xPos;


(property) PaintOptions.xPos?: number | undefined
  let yPos = opts.yPos;


(property) PaintOptions.yPos?: number | undefined
  // ...
}
```

No JavaScript, mesmo que a propriedade nunca tenha sido definida, ainda podemos acessá-la — ela apenas nos dará o valor `undefined`. Podemos tratar `undefined` de forma especial verificando-o.

```
function paintShape(opts: PaintOptions) {
  let xPos = opts.xPos === undefined ? 0 : opts.xPos;

let xPos: number
  let yPos = opts.yPos === undefined ? 0 : opts.yPos;

let yPos: number
  // ...
}
```

Observe que esse padrão de definir valores padrão para valores não especificados é tão comum que o JavaScript tem sintaxe para suportá-lo.

```
function paintShape({ shape, xPos = 0, yPos = 0 }: PaintOptions) {
  console.log("x coordinate at", xPos);
  console.log("y coordinate at", yPos);
  // ...
}
```

Aqui usamos um padrão de destruturação para o parâmetro de `paintShape`, e fornecemos `valores padrão` para `xPos` e `yPos`. Agora, `xPos` e `yPos` estão definitivamente presentes no corpo de `paintShape`, mas são opcionais para qualquer um que chamar a função.

> Note que atualmente não há como colocar anotações de tipo dentro de padrões de destruturação. Isso ocorre porque a seguinte sintaxe já tem um significado diferente no JavaScript.

```
function draw({ shape: Shape, xPos: number = 100 /*...*/ }) {
  render(shape);
  // Erros de compilação...
}
```

Em um padrão de destruturação de objeto, `shape: Shape` significa “pegar a propriedade `shape` e redefini-la localmente como uma variável chamada `Shape`.” Da mesma forma, `xPos: number` cria uma variável chamada `number` cujo valor é baseado no parâmetro `xPos`.

Propriedades `readonly`

As propriedades também podem ser marcadas como `readonly` no TypeScript. Embora isso não mude nenhum comportamento em tempo de execução, uma propriedade marcada como `readonly` não pode ser reescrita durante a verificação de tipos.

```
interface SomeType {
  readonly prop: string;
}

function doSomething(obj: SomeType) {
  // Podemos ler de 'obj.prop'.
  console.log(`prop has the value '${obj.prop}'.`);

  // Mas não podemos reatribuir.
  obj.prop = "hello"; // Erro
}
```

Usar o modificador `readonly` não implica que um valor seja totalmente imutável — em outras palavras, que seu conteúdo interno não possa ser alterado. Isso apenas significa que a propriedade em si não pode ser reescrita.

```
interface Home {
  readonly resident: { name: string; age: number };
}

function visitForBirthday(home: Home) {
  // Podemos ler e atualizar propriedades de 'home.resident'.
  console.log(`Happy birthday ${home.resident.name}!`);
  home.resident.age++;
}

function evict(home: Home) {
  // Mas não podemos escrever na propriedade 'resident' em um 'Home'.
  home.resident = { // Erro
    name: "Victor the Evictor",
    age: 42,
  };
}
```

É importante gerenciar as expectativas sobre o que `readonly` implica. É útil para sinalizar a intenção durante o tempo de desenvolvimento sobre como um objeto deve ser utilizado. O TypeScript não

considera se as propriedades de dois tipos são `readonly` ao verificar se esses tipos são compatíveis, então propriedades `readonly` também podem mudar via aliasing.

```
interface Person {
  name: string;
  age: number;
}

interface ReadonlyPerson {
  readonly name: string;
  readonly age: number;
}

let writablePerson: Person = {
  name: "Person McPersonface",
  age: 42,
};

// funciona
let readonlyPerson: ReadonlyPerson = writablePerson;

console.log(readonlyPerson.age); // imprime '42'
writablePerson.age++;
console.log(readonlyPerson.age); // imprime '43'
```

Usando modificadores de mapeamento, você pode remover atributos `readonly`.

Assinaturas de Índice

Às vezes, você não conhece todos os nomes das propriedades de um tipo com antecedência, mas sabe a forma dos valores.

Nesses casos, você pode usar uma assinatura de índice para descrever os tipos de valores possíveis, por exemplo:

```
interface StringArray {
  [index: number]: string;
}

const myArray: StringArray = getStringArray();
const secondItem = myArray[1];

const secondItem: string
```

Acima, temos uma interface `StringArray` que possui uma assinatura de índice. Essa assinatura de índice afirma que, quando um `StringArray` é indexado com um `número`, retornará um `string`.

Apenas alguns tipos são permitidos para propriedades de assinaturas de índice: `string`, `number`, `symbol`, padrões de strings de template e tipos de união consistindo apenas destes.

É possível suportar múltiplos tipos de indexadores...
Enquanto as assinaturas de índice de string são uma maneira poderosa de descrever o padrão “dicionário”, elas também impõem que todas as propriedades correspondam ao seu tipo de retorno. Isso ocorre porque um índice de string declara que `obj.property` também está disponível como

`obj["property"]`. No exemplo a seguir, o tipo de `name` não corresponde ao tipo do índice de string, e o verificador de tipo gera um erro:

```
interface NumberDictionary {
  [index: string]: number;

  length: number; // ok
  name: string;
Property 'name' of type 'string' is not assignable to 'string' index type 'number'.
}
```

No entanto, propriedades de tipos diferentes são aceitáveis se a assinatura de índice for uma união dos tipos das propriedades:

```
interface NumberOrStringDictionary {
  [index: string]: number | string;
  length: number; // ok, length é um número
  name: string; // ok, name é uma string
}
```

Finalmente, você pode tornar as assinaturas de índice `somente leitura` para impedir a atribuição a seus índices:

```
interface ReadonlyStringArray {
  readonly [index: number]: string;
}

let myArray: ReadonlyStringArray = getReadOnlyStringArray();
myArray[2] = "Mallory";
Index signature in type 'ReadonlyStringArray' only permits reading.
```

Você não pode definir `myArray[2]` porque a assinatura de índice é `somente leitura`.

Verificações de Propriedade Excessiva

Onde e como um objeto é atribuído a um tipo pode fazer diferença no sistema de tipos. Um dos exemplos chave disso é nas verificações de propriedades excessivas, que valida o objeto mais minuciosamente quando é criado e atribuído a um tipo de objeto durante a criação.

```
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
  return {
    color: config.color || "red",
    area: config.width ? config.width * config.width : 20,
  };
}

let mySquare = createSquare({ colour: "red", width: 100 });
Object literal may only specify known properties, but 'colour' does not exist in type
'SquareConfig'. Did you mean to write 'color'?
```

Note que o argumento dado a `createSquare` está escrito *colour* em vez de `color`. Em JavaScript simples, esse tipo de coisa falha silenciosamente.

Você poderia argumentar que este programa está tipado corretamente, já que as propriedades de `width` são compatíveis, não há uma propriedade `color` presente, e a propriedade extra `colour` é insignificante.

No entanto, o TypeScript assume que há provavelmente um bug neste código. Literais de objeto recebem tratamento especial e passam por *verificações de propriedades excessivas* ao serem atribuídos a outras variáveis ou passados como argumentos. Se um literal de objeto tiver quaisquer propriedades que o "tipo alvo" não possui, você receberá um erro:

```
let mySquare = createSquare({ colour: "red", width: 100 });
Object literal may only specify known properties, but 'colour' does not exist in type
'SquareConfig'. Did you mean to write 'color'?
```

Contornar essas verificações é realmente simples. O método mais fácil é apenas usar uma asserção de tipo:

```
let mySquare = createSquare({ width: 100, opacity: 0.5 } as SquareConfig);
```

No entanto, uma abordagem melhor pode ser adicionar uma assinatura de índice de string se você tiver certeza de que o objeto pode ter algumas propriedades extras que são usadas de alguma maneira especial. Se `SquareConfig` pode ter propriedades `color` e `width` com os tipos acima, mas também poderia *também* ter qualquer número de outras propriedades, então poderíamos defini-lo assim:

```
interface SquareConfig {
  color?: string;
  width?: number;
  [propName: string]: any;
}
```

Aqui estamos dizendo que `SquareConfig` pode ter qualquer número de propriedades, e desde que não sejam `color` ou `width`, seus tipos não importam.

Uma maneira final de contornar essas verificações, que pode ser um pouco surpreendente, é atribuir o objeto a outra variável: como atribuir `squareOptions` não passará por verificações de propriedades excessivas, o compilador não lhe dará um erro:

```
let squareOptions = { colour: "red", width: 100 };
let mySquare = createSquare(squareOptions);
```

A solução acima funcionará enquanto você tiver uma propriedade comum entre `squareOptions` e `SquareConfig`. Neste exemplo, a propriedade era `width`. No entanto, falhará se a variável não tiver nenhuma propriedade de objeto em comum. Por exemplo:

```
let squareOptions = { colour: "red" };
let mySquare = createSquare(squareOptions);
Type '{ colour: string; }' has no properties in common with type 'SquareConfig'.
```

Lembre-se de que, para um código simples como o acima, você provavelmente não deve tentar "contornar" essas verificações. Para literais de objeto mais complexos que possuem métodos e mantêm estado, você pode precisar ter essas técnicas em mente, mas a maioria dos erros de propriedade excessiva é, na verdade, um bug.

Isso significa que, se você estiver enfrentando problemas de verificação de propriedades excessivas para algo como sacolas de opções, pode ser necessário revisar algumas de suas declarações de tipo. Nesse caso, se for aceitável passar um objeto com uma propriedade `color` ou `colour` para `createSquare`, você deve ajustar a definição de `SquareConfig` para refletir isso.

Estendendo Tipos

É bastante comum ter tipos que possam ser versões mais específicas de outros tipos. Por exemplo, podemos ter um tipo `BasicAddress` que descreve os campos necessários para enviar cartas e pacotes nos EUA.

```
interface BasicAddress {
  name?: string;
  street: string;
  city: string;
  country: string;
  postalCode: string;
}
```

Em algumas situações, isso é suficiente, mas endereços frequentemente têm um número de unidade associado se o edifício em um endereço tiver várias unidades. Podemos então descrever um `AddressWithUnit`.

```
interface AddressWithUnit {
  name?: string;
  unit: string;
  street: string;
  city: string;
  country: string;
  postalCode: string;
}
```

Isso resolve o problema, mas a desvantagem aqui é que tivemos que repetir todos os outros campos de `BasicAddress` quando nossas mudanças eram puramente aditivas. Em vez disso, podemos estender o tipo original `BasicAddress` e apenas adicionar os novos campos que são exclusivos de `AddressWithUnit`.

```
interface BasicAddress {
  name?: string;
  street: string;
  city: string;
  country: string;
  postalCode: string;
}

interface AddressWithUnit extends BasicAddress {
  unit: string;
}
```

A palavra-chave `extends` em uma `interface` nos permite copiar efetivamente membros de outros tipos nomeados e adicionar os novos membros que quisermos. Isso pode ser útil para reduzir a quantidade de código repetitivo que temos que escrever e para sinalizar a intenção de que várias declarações da mesma propriedade podem estar relacionadas. Por exemplo, `AddressWithUnit` não precisou repetir a

propriedade `street`, e como `street` se origina de `BasicAddress`, um leitor saberá que esses dois tipos estão relacionados de alguma forma.

As `interfaces` também podem estender de múltiplos tipos.

```
interface Colorful {
  color: string;
}

interface Circle {
  radius: number;
}

interface ColorfulCircle extends Colorful, Circle {}

const cc: ColorfulCircle = {
  color: "red",
  radius: 42,
};
```

Tipos de Interseção

As `interfaces` nos permitem construir novos tipos a partir de outros tipos estendendo-os. O TypeScript fornece outra construção chamada *tipos de interseção*, que é usada principalmente para combinar tipos de objetos existentes.

Um tipo de interseção é definido usando o operador `&`.

```
interface Colorful {
  color: string;
}
interface Circle {
  radius: number;
}

type ColorfulCircle = Colorful & Circle;
```

Aqui, nós interseccionamos `Colorful` e `Circle` para produzir um novo tipo que tem todos os membros de `Colorful` *e* `Circle`.

```
function draw(circle: Colorful & Circle) {
  console.log(`Color was ${circle.color}`);
  console.log(`Radius was ${circle.radius}`);
}

// ok
draw({ color: "blue", radius: 42 });

// opa
draw({ color: "red", raidus: 42 });
Object literal may only specify known properties, but 'raidus' does not exist in type
'Colorful & Circle'. Did you mean to write 'radius'?
```

Interfaces vs. Interseções

Acabamos de ver duas maneiras de combinar tipos que são semelhantes, mas na verdade são sutilmente diferentes. Com interfaces, poderíamos usar uma cláusula `extends` para estender de outros tipos, e conseguimos fazer algo similar com interseções e nomear o resultado com um alias de tipo. A principal diferença entre os dois é como os conflitos são tratados, e essa diferença é geralmente uma das principais razões pelas quais você escolheria um em vez do outro entre uma interface e um alias de tipo de interseção.

Se interfaces forem definidas com o mesmo nome, o TypeScript tentará mesclá-las se as propriedades forem compatíveis. Se as propriedades não forem compatíveis (ou seja, tiverem o mesmo nome de propriedade, mas tipos diferentes), o TypeScript levantará um erro.

No caso de tipos de interseção, propriedades com tipos diferentes serão mescladas automaticamente. Quando o tipo é usado posteriormente, o TypeScript esperará que a propriedade satisfaça ambos os tipos simultaneamente, o que pode produzir resultados inesperados.

Por exemplo, o seguinte código gerará um erro porque as propriedades são incompatíveis:

```
interface Person {
  name: string;
}
interface Person {
  name: number;
}
```

Em contraste, o seguinte código será compilado, mas resulta em um tipo `never`:

```
interface Person1 {
  name: string;
}

interface Person2 {
  name: number;
}

type Staff = Person1 & Person2;

declare const staffer: Staff;
staffer.name;

(property) name: never
```

Neste caso, `Staff` exigiria que a propriedade `name` fosse tanto uma string quanto um número, o que resulta na propriedade sendo do tipo `never`.

Tipos de Objetos Genéricos

Vamos imaginar um tipo `Box` que pode conter qualquer valor — `string`s, `number`s, `Giraffe`s, o que quer que seja.

```
interface Box {
  contents: any;
}
```

Neste momento, a propriedade `contents` é tipada como `any`, o que funciona, mas pode levar a acidentes no futuro.

Poderíamos usar `unknown`, mas isso significaria que, em casos onde já conhecemos o tipo de `contents`, precisaríamos fazer verificações de precaução ou usar asserções de tipo propensas a erros.

```
interface Box {
  contents: unknown;
}

let x: Box = {
  contents: "hello world",
};

// poderíamos checar 'x.contents'
if (typeof x.contents === "string") {
  console.log(x.contents.toLowerCase());
}

// ou poderíamos usar uma asserção de tipo
console.log((x.contents as string).toLowerCase());
```

Uma abordagem segura seria criar diferentes tipos de `Box` para cada tipo de `contents`.

```
interface NumberBox {
  contents: number;
}

interface StringBox {
  contents: string;
}

interface BooleanBox {
  contents: boolean;
}
```

Mas isso significa que teremos que criar diferentes funções ou sobrecargas de funções para operar nesses tipos.

```
function setContents(box: StringBox, newContents: string): void;
function setContents(box: NumberBox, newContents: number): void;
function setContents(box: BooleanBox, newContents: boolean): void;
function setContents(box: { contents: any }, newContents: any) {
  box.contents = newContents;
}
```

Isso é muito boilerplate. Além disso, podemos precisar introduzir novos tipos e sobrecargas mais tarde. Isso é frustrante, já que nossos tipos de caixa e sobrecargas são todos efetivamente os mesmos.

Em vez disso, podemos fazer um tipo `Box` *genérico* que declara um *parâmetro de tipo*.

```
interface Box<Type> {
  contents: Type;
}
```

Você pode ler isso como “Uma `Box` de `Type` é algo cujos `contents` têm tipo `Type`”. Mais tarde, ao nos referirmos a `Box`, precisamos fornecer um *argumento de tipo* em lugar de `Type`.

```
let box: Box<string>;
```

Pense em `Box` como um template para um tipo real, onde `Type` é um espaço reservado que será substituído por algum outro tipo. Quando o TypeScript vê `Box<string>`, ele substituirá cada instância de `Type` em `Box<Type>` por `string`, e acabará trabalhando com algo como `{ contents: string }`. Em outras palavras, `Box<string>` e nosso anterior `StringBox` funcionam de forma idêntica.

```
interface Box<Type> {
  contents: Type;
}
interface StringBox {
  contents: string;
}

let boxA: Box<string> = { contents: "hello" };
boxA.contents;

(property) Box<string>.contents: string

let boxB: StringBox = { contents: "world" };
boxB.contents;

(property) StringBox.contents: string
```

`Box` é reutilizável na medida em que `Type` pode ser substituído por qualquer coisa. Isso significa que, quando precisamos de uma caixa para um novo tipo, não precisamos declarar um novo tipo de `Box` (embora certamente pudéssemos, se quiséssemos).

```
interface Box<Type> {
  contents: Type;
}

interface Apple {
  // ....
}

// O mesmo que '{ contents: Apple }'.
type AppleBox = Box<Apple>;
```

Isso também significa que podemos evitar sobrecargas completamente, usando em vez disso funções genéricas.

```
function setContents<Type>(box: Box<Type>, newContents: Type) {
  box.contents = newContents;
}
```

Vale a pena notar que aliases de tipo também podem ser genéricos. Poderíamos ter definido nossa nova interface `Box<Type>` como:

```
interface Box<Type> {
  contents: Type;
}
```

usando um alias de tipo em vez disso:

```
type Box<Type> = {
  contents: Type;
};
```

Como aliases de tipo, ao contrário de interfaces, podem descrever mais do que apenas tipos de objeto, também podemos usá-los para escrever outros tipos de ajuda genéricos.

```
type OrNull<Type> = Type | null;

type OneOrMany<Type> = Type | Type[];

type OneOrManyOrNull<Type> = OrNull<OneOrMany<Type>>;

type OneOrManyOrNull<Type> = OneOrMany<Type> | null;

type OneOrManyOrNullStrings = OneOrManyOrNull<string>;

type OneOrManyOrNullStrings = OneOrMany<string> | null;
```

Voltaremos a aliases de tipo em breve.

O Tipo `Array`

Tipos de objetos genéricos são frequentemente algum tipo de contêiner que funciona de forma independente do tipo de elementos que contêm. É ideal que estruturas de dados funcionem assim para que sejam reutilizáveis entre diferentes tipos de dados.

Na verdade, temos trabalhado com um tipo assim ao longo deste manual: o tipo `Array`. Sempre que escrevemos tipos como `number[]` ou `string[]`, isso é apenas uma forma abreviada de `Array<number>` e `Array<string>`.

```
function doSomething(value: Array<string>) {
  // ...
}

let myArray: string[] = ["hello", "world"];

// qualquer um desses funciona!
doSomething(myArray);
doSomething(new Array("hello", "world"));
```

Assim como o tipo `Box` acima, `Array` em si é um tipo genérico.

```
interface Array<Type> {
  length: number;
  pop(): Type | undefined;
  push(...items: Type[]): number;
  // ...
}
```

JavaScript moderno também fornece outras estruturas de dados que são genéricas, como `Map<K, V>`, `Set<T>`, e `Promise<T>`. Tudo isso realmente significa que, devido a como `Map`, `Set`, e `Promise` se comportam, eles podem trabalhar com qualquer conjunto de tipos.

O Tipo `ReadonlyArray`

O `ReadonlyArray` é um tipo especial que descreve arrays que não devem ser alterados.

```
function doStuff(values: ReadonlyArray<string>) {
  // Podemos ler de 'values'...
  const copy = values.slice();
  console.log(`The first value is ${values[0]}`);

  // ...mas não podemos mutar 'values'.
  values.push("hello!");
  // Property 'push' does not exist on type 'readonly string[]'.
}
```

Assim como o modificador `readonly` para propriedades, é principalmente uma ferramenta que podemos usar para indicar intenção. Quando vemos uma função que retorna `ReadonlyArray`s, isso nos diz que não devemos mudar o conteúdo, e quando vemos uma função que consome `ReadonlyArray`s, isso nos diz que podemos passar qualquer array para essa função sem nos preocupar que seu conteúdo será alterado.

Diferente de `Array`, não existe um construtor `ReadonlyArray` que possamos usar.

```
new ReadonlyArray("red", "green", "blue");
// 'ReadonlyArray' only refers to a type, but is being used as a value here.
```

Em vez disso, podemos atribuir arrays regulares a `ReadonlyArray`s.

```
const roArray: ReadonlyArray<string> = ["red", "green", "blue"];
```

Assim como TypeScript fornece uma sintaxe abreviada para `Array<Type>` com `Type[]`, também fornece uma sintaxe abreviada para `ReadonlyArray<Type>` com `readonly Type[]`.

```
function doStuff(values: readonly string[]) {
  // Podemos ler de 'values'...
  const copy = values.slice();
  console.log(`The first value is ${values[0]}`);

  // ...mas não podemos mutar 'values'.
  values.push("hello!");
  // Property 'push' does not exist on type 'readonly string[]'.
}
```

Uma última coisa a notar é que, ao contrário do modificador de propriedade `readonly`, a atribuição não é bidirecional entre arrays regulares e `ReadonlyArray`s.

```
let x: readonly string[] = [];
let y: string[] = [];

x = y; // ok
y = x; // erro: The type 'readonly string[]' is 'readonly' and cannot be assigned to
the mutable type 'string[]'.
```

Tipos de Tupla

`Um tipo de tupla é outro tipo de` Array `que sabe exatamente quantos elementos contém e exatamente quais tipos contém em posições específicas.`

```
type StringNumberPair = [string, number];
```

Aqui, `StringNumberPair` é um tipo de tupla de `string` e `number`. Assim como `ReadonlyArray`, não tem representação em tempo de execução, mas é significativo para o TypeScript. Para o sistema de tipos, `StringNumberPair` descreve arrays cujo índice `0` contém um `string` e cujo índice `1` contém um `number`.

```
function doSomething(pair: [string, number]) {
  const a = pair[0];

const a: string
  const b = pair[1];

const b: number
  // ...
}

doSomething(["hello", 42]);
```

Se tentarmos indexar além do número de elementos, teremos um erro.

```
function doSomething(pair: [string, number]) {
  // ...

  const c = pair[2];
Tipo de tupla '[string, number]' de comprimento '2' não tem elemento no índice '2'.
}
```

Podemos também desestruturar tuplas usando a desestruturação de arrays do JavaScript.

```
function doSomething(stringHash: [string, number]) {
  const [inputString, hash] = stringHash;

  console.log(inputString);

const inputString: string

  console.log(hash);

const hash: number
}
```

> Tipos de tupla são úteis em APIs baseadas em convenções, onde o significado de cada elemento é "óbvio". Isso nos dá flexibilidade em como queremos nomear nossas variáveis ao desestruturá-las. No exemplo acima, pudemos nomear os elementos 0 e 1 como quiséssemos.
> No entanto, como nem todos os usuários têm a mesma visão do que é óbvio, pode valer a pena reconsiderar se usar objetos com nomes de propriedades descritivos pode ser melhor para sua API.

Além dessas verificações de comprimento, tipos de tupla simples como esses são equivalentes a tipos que são versões de `Array`s que declaram propriedades para índices específicos e que declaram `length` com um tipo literal numérico.

```
interface StringNumberPair {
  // propriedades especializadas
  length: 2;
  0: string;
  1: number;

  // Outros membros de 'Array<string | number>'...
```

```
  slice(start?: number, end?: number): Array<string | number>;
}
```

Outra coisa que pode interessar é que tuplas podem ter propriedades opcionais ao escrever um ponto de interrogação (`?`) após o tipo de um elemento. Elementos de tupla opcionais só podem vir no final e também afetam o tipo de `length`.

```
type Either2dOr3d = [number, number, number?];

function setCoordinate(coord: Either2dOr3d) {
  const [x, y, z] = coord;

const z: number | undefined

  console.log(`As coordenadas fornecidas tinham ${coord.length} dimensões`);

(propriedade) length: 2 | 3
}
```

Tuplas também podem ter elementos de resto, que precisam ser um tipo de array/tupla.

```
type StringNumberBooleans = [string, number, ...boolean[]];
type StringBooleansNumber = [string, ...boolean[], number];
type BooleansStringNumber = [...boolean[], string, number];
```

- `StringNumberBooleans` descreve uma tupla cujos dois primeiros elementos são `string` e `number`, respectivamente, mas que pode ter qualquer número de `boolean`s a seguir.
- `StringBooleansNumber` descreve uma tupla cujo primeiro elemento é `string` e depois qualquer número de `boolean`s, terminando com um número.
- `BooleansStringNumber` descreve uma tupla cujos elementos iniciais são qualquer número de `boolean`s, terminando com um `string` e depois um `number`.
  Uma tupla com um elemento de resto não tem um “comprimento” definido - só tem um conjunto de elementos bem conhecidos em posições diferentes.

```
const a: StringNumberBooleans = ["hello", 1];
const b: StringNumberBooleans = ["beautiful", 2, true];
const c: StringNumberBooleans = ["world", 3, true, false, true, false, true];
```

Por que elementos opcionais e de resto podem ser úteis? Bem, isso permite que o TypeScript corresponda tuplas com listas de parâmetros. Tipos de tupla podem ser usados em [parâmetros e argumentos de resto](), para que o seguinte:

```
function readButtonInput(...args: [string, number, ...boolean[]]) {
  const [name, version, ...input] = args;
  // ...
}
```

é basicamente equivalente a:

```
function readButtonInput(name: string, version: number, ...input: boolean[]) {
  // ...
}
```

Isso é útil quando você quer aceitar um número variável de argumentos com um parâmetro de resto, e precisa de um número mínimo de elementos, mas não quer introduzir variáveis intermediárias.

Tipos de Tupla `readonly`

Uma nota final sobre tipos de tupla - tipos de tupla têm variantes `readonly`, e podem ser especificados ao adicionar um modificador `readonly` na frente deles - assim como com a sintaxe abreviada de arrays.

```
function doSomething(pair: readonly [string, number]) {
  // ...
}
```

Como você pode esperar, escrever em qualquer propriedade de uma tupla `readonly` não é permitido no TypeScript.

```
function doSomething(pair: readonly [string, number]) {
  pair[0] = "hello!";
Não é possível atribuir a '0' porque é uma propriedade somente leitura.
}
```

Tuplas tendem a ser criadas e deixadas não modificadas na maioria do código, então anotar tipos como tuplas `readonly` quando possível é um bom padrão. Isso também é importante considerando que literais de array com afirmações `const` serão inferidos como tipos de tupla `readonly`.

```
let point = [3, 4] as const;

function distanceFromOrigin([x, y]: [number, number]) {
  return Math.sqrt(x ** 2 + y ** 2);
}

distanceFromOrigin(point);
Argumento do tipo 'readonly [3, 4]' não é atribuível ao parâmetro do tipo '[number,
number]'.
  O tipo 'readonly [3, 4]' é 'readonly' e não pode ser atribuído ao tipo mutável
'[number, number]'.
```

Aqui, `distanceFromOrigin` nunca modifica seus elementos, mas espera uma tupla mutável. Como o tipo de `point` foi inferido como `readonly [3, 4]`, não será compatível com `[number, number]`, uma vez que esse tipo não pode garantir que os elementos de `point` não serão modificados.