Typescript para programadores Javascript

Já existe um pequeno conjunto de tipos primitivos disponíveis no javascript: boolean, bigint, null, number, string, symbol e undefined que é possível usar em uma interface. O typescript aumentar essa lista com mais alguns, como any (permitir qualquer valor), unknown (garantir que alguém que usa este tipo declare qual é o tipo), never (não é possível que este tipo possa existir), e void (uma função que retorna undefined ou não tem valor de retorno).

Existem duas sintaxes para a construção de tipos: interfaces e types. É preferível usar interface, use type quando precisar de recursos específicos.

Compondo Tipos

Com uma união, é possível declarar que um tipo pode ser um dentre muitos tipos. Por exemplo, é possível descrever um tipo boolean como sendo true ou false.

```
type MyBool = true | false
```

Se passar o mouse por cima do MyBool, verá que ele é classificado como boolean. Essa é uma propriedade do sistema de tipos estruturais

Um caso de uso popular para tipos de união é descrever o valor que um conjunto de literais de string ou number pode ter.

```
type WindowStates = "open" | "closed" | "minimized"
type LockStates = "locked" | "unlocked"
type PositivaeOddNumbersUnderTen = 1 | 3 | 5 | 7 | 9
```

As uniões oferecem uma forma de gerenciar tipos diferentes. Por exemplo, uma função que recebe array ou string:

```
function getLength(obj: string | string[]) {
  return obj.length;
}
```

É possível fazer uma função retornar valores diferentes dependendo se ela recebeu uma string ou array:

```
function wrapInArray(obj: string | string[]) {
  if (typeof obj === "string") {
    return [obj];

(parameter) obj: string
  }
  return obj;
}
```

Genéricos

Fornecem variáveis para tipos. Um exemplo é um array, ele sem genericos pode conter qualquer coisa. Ja ele com genéricos pode descrever os valores que aquele array contém.

```
type StringArray = Array<string>
type NumberArray = Array<number>
type ObjectWithNameArray = Array<{ name: string }>
```

É possível declarar tipos próprios que usam genéricos:

```
interface Backpack<Type> {
  add: (obj: Type) => void
  get: () => Type
}
```

Esse é um atalho para dizer ao Typescript que há uma constante chamada mochila, e não se preocupar de onde ela veio.

```
declare const backpack: Backpack<string>
```

Objeto é uma string, porque nós o declaramos acima como a parte variável de Mochila.

```
const object = backpack.get()
```

Já que a variável mochila é uma string, você não pode passar um número para a função adicionar.

```
backpack.add(23)
```

```
Argument of type 'number' is not assignable to parameter of type 'string'.
```

Sistema de Tipos Estruturais

Um dos princípios centrais do TypeScript é que a checagem de tipo é focada no formato que os valores têm. Isso é chamado as vezes de “tipagem do pato” ou “tipagem estrutural”.

Em um sistema de tipagem estruturado, se dois objetos tem o mesmo formato, eles são considerados do mesmo tipo.

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

function logPoint(p: Point) {
  console.log(`${p.x}, ${p.y}`)
}

// logs "12, 26"
const point = { x: 12, y: 26 }
logPoint(point)
```

A variável ponto nunca é declarada como sendo do tipo Ponto. Entretanto, o TypeScript compara o formato de ponto ao formato de Ponto na checagem de tipo. Eles têm o mesmo formato, então o código passa.

A correspondência de tipo só requere que um subconjunto de campos do objeto sejam correspondentes:

```
const point3 = { x: 12, y: 26, z: 89 }
logPoint(point3) // logs "12, 26"

const rect = { x: 33, y: 3, width: 30, height: 80 }
logPoint(rect) // logs "33, 3"

const color = { hex: "#187ABF" }
logPoint(color)
```

```
Argument of type '{ hex: string; }' is not assignable to parameter of type 'Point'.
Type '{ hex: string; }' is missing the following properties from type 'Point': x, y
```

Não há diferença entre como classes e objetos se adaptam às formas:

```
class VirtualPoint {
  x: number
  y: number

  constructor(x: number, y: number) {
    this.x = x
    this.y = y
  }
}

const newVPoint = new VirtualPoint(13, 56)
logPoint(newVPoint) // logs "13, 56"
```

Se o objeto ou classe tem todas as propriedades requeridas, TypeScript dirá que eles são correspondentes, independente dos detalhes de implementação.