10

Union Types e Type Guards

Aprendemos que o uso de herança e interfaces nos permite escrever códigos genéricos que se beneficiam da tipagem estática. Contudo, há outra forma de escrevermos um código genérico sem o uso de interfaces e sem apelarmos diretamente para o uso do tipo `any`.

Union Types

Temos a seguinte função que espera receber um token de autenticação para então tratá-lo (você pode criar um arquivo **union.ts** para testar gradativamente o código que veremos. Não é necessário executá-lo, apenas verificar os erros de compilação no VSCode).

```
function processaToken(token: string) {
    // muda o dígito 2 por X!
    return token.replace(/2/g,'X');
}
const tokenProcessado = processaToken('1234');
```

Excelente, mas no sistema legado que estamos utilizando TypeScript, em algums momentos `processaToken` não recebe uma `string`, mas um `number`.

```
// erro de compilação
const tokenProcessado = processaToken(1234);
```

Para resolvermos este problema, poderíamos ter usado o tipo `any`, mas nesse caso perderíamos o autocomplete e a checagem mais rígida do TypeScript que garante a saúde do nosso código.

Podemos fazer com que a função aceite tanto `string` quanto `number` através de **union types**. Alterando nosso código:

```
// ATENÇÃO, NOSSO CÓDIGO AINDA NÃO COMPILARÁ

// agora aceita os tipos string e number!
function processaToken(token: string | number) {

    // muda o dígito 2 por X!
    // erro de compilação aqui
    return token.replace(/2/g,'X');
}

// compila
const tokenProcessado1 = processaToken('1234');
// compila
const tokenProcessado2 = processaToken(1234);
```

Podemos agora passar os tipos `string` e `number` para `processaToken` que o compilador TypeScript aceitará, contudo, haverá um erro de compilação dentro da função `processaToken`, pois o método `replace` só existe em string e não em number. E agora? Para solucionar esse problema entra em ação o **Type Guards**.

Type Guards

Podemos fazer com que nosso código compile checando o tipo dentro da função:

```
function processaToken(token: string | number) {

    if(typeof(token) === 'string') {

        // typescript entende que é o tipo string e faz autocomplete para este tipo. A função re
        return token.replace(/2/g,'X');
    } else {
        // toFixed só existe em number!
        return token.toFixed().replace(/2/g,'X');
    }
}

const tokenProcessado1 = processaToken('1234');
const tokenProcessado2 = processaToken(1234);
```

Com Type Guards, quando realizamos o teste verificando o tipo do parâmetro, dentro da condição o compilador saber inferir o tipo correto e a checagem e autocomplete para aquele tipo é ativado. Dentro do `if` o tipo é considerado `string` e no else `number`.

Considerações sobre o uso de Type Guards

Apesar de ser um recurso da linguagem, essa estratégia remete à programação procedural pois envolve uma sucessão de if's para detectar o tipo dos elementos. É por este motivo que não foi utilizado em nosso projeto e demos preferência ao polimorfismo.