TypeScript
TUTORIAL

TypeScript Type Guards

Summary: in this tutorial, you will learn about the Type Guard in TypeScript to narrow down the type of a variable.

Type Guards allow you to narrow down the type of a variable within a conditional block.

typeof

Let's take a look at the following example:

```
type alphanumeric = string | number;

function add(a: alphanumeric, b: alphanumeric) {
    if (typeof a === 'number' && typeof b === 'number') {
        return a + b;
    }

    if (typeof a === 'string' && typeof b === 'string') {
        return a.concat(b);
    }

    throw new Error('Invalid arguments. Both arguments must be either numbers or strings.'
}
```

How it works:

- First, define the `alphanumeric` type that can hold either a string or a number.

- Next, declare a function that adds two variables `a` and `b` with the type of `alphanumeric`.
- Then, check if both types of arguments are numbers using the `typeof` operator. If yes, then calculate the sum of arguments using the `+` operator.
- After that, check if both types of arguments are strings using the `typeof` operator. If yes, then concatenate two arguments.
- Finally, throw an error if arguments are neither numbers nor strings.

In this example, TypeScript knows the usage of the `typeof` operator in the conditional blocks. Inside the following `if` block, TypeScript realizes that `a` and `b` are numbers.

```
if (typeof a === 'number' && typeof b === 'number') {
    return a + b;
}
```

Similarly, in the following `if` block, TypeScript treats `a` and `b` as strings, therefore, you can concatenate them into one:

```
if (typeof a === 'string' && typeof b === 'string') {
    return a.concat(b);
}
```

instanceof

Similar to the `typeof` operator, TypeScript is also aware of the usage of the `instanceof` operator. For example:

```
class Customer {
    isCreditAllowed(): boolean {
        // ...
        return true;
    }
}

class Supplier {
    isInShortList(): boolean {
        // ...
        return true;
    }
}

type BusinessPartner = Customer | Supplier;

function signContract(partner: BusinessPartner) : string {
    let message: string;
    if (partner instanceof Customer) {
        message = partner.isCreditAllowed() ? 'Sign a new contract with the customer' : 'C
    }

    if (partner instanceof Supplier) {
        message = partner.isInShortList() ? 'Sign a new contract the supplier' : 'Need to
    }

    return message;
}
```

How it works:

- First, declare the `Customer` and `Supplier` classes.
  Second, create a type alias `BusinessPartner` which is a union type of `Customer` and `Supplier`.
- Third, declare a function `signContract()` that accepts a parameter with the type `BusinessPartner`.
- Finally, check if the partner is an instance of `Customer` or `Supplier`, and then provide the respective logic.

Inside the following `if` block, TypeScript knows that the partner is an instance of the `Customer` type due to the `instanceof` operator:

```
if (partner instanceof Customer) {
    message = partner.isCreditAllowed() ? 'Sign a new contract with the customer' : 'Credi
}
```

Likewise, TypeScript knows that the partner is an instance of `Supplier` inside the following `if` block:

```
if (partner instanceof Supplier) {
    message = partner.isInShortList() ? 'Sign a new contract with the supplier' : 'Need tc
}
```

When an if narrows out one type, TypeScript knows that within the `else` it is not that type but the other. For example:

```
function signContract(partner: BusinessPartner) : string {
    let message: string;
    if (partner instanceof Customer) {
        message = partner.isCreditAllowed() ? 'Sign a new contract with the customer' : 'C
    } else {
        // must be Supplier
        message = partner.isInShortList() ? 'Sign a new contract with the supplier' : 'Nee
    }
    return message;
}
```

in

The `in` operator carries a safe check for the existence of a property on an object. You can also use it as a type guard. For example:

```
function signContract(partner: BusinessPartner) : string {
    let message: string;
    if ('isCreditAllowed' in partner) {
        message = partner.isCreditAllowed() ? 'Sign a new contract with the customer' : 'C
    } else {
        // must be Supplier
        message = partner.isInShortList() ? 'Sign a new contract the supplier ' : 'Need tc
    }
    return message;
}
```

User-defined Type Guards

User-defined type guards allow you to define a type guard or help TypeScript infer a type when you use a function.

A user-defined type guard function is a function that simply returns `arg is aType`. For example:

```
function isCustomer(partner: any): partner is Customer {
    return partner instanceof Customer;
}
```

In this example, the `isCustomer()` is a user-defined type guard function. Now you can use it in as follows:

```
function signContract(partner: BusinessPartner): string {
    let message: string;
    if (isCustomer(partner)) {
        message = partner.isCreditAllowed() ? 'Sign a new contract with the customer' : 'C
    } else {
        message = partner.isInShortList() ? 'Sign a new contract with the supplier' : 'Nee
    }

    return message;
}
```

Summary

- Type guards narrow down the type of a variable within a conditional block.
- Use the `typeof` and `instanceof` operators to implement type guards in the conditional blocks