TypeScript Interface

Summary: in this tutorial, you'll learn about TypeScript interfaces and how to use them to enforce type-checking.

Introduction to TypeScript interfaces

TypeScript interfaces define the contracts within your code. They also provide explicit names for type checking.

Let's start with a simple example:

```
function getFullName(person: {
    firstName: string;
    lastName: string
}) {
    return `${person.firstName} ${person.lastName}`;
}

let person = {
    firstName: 'John',
    lastName: 'Doe'
};

console.log(getFullName(person));
```

Output:

```
John Doe
```

In this example, the TypeScript compiler checks the argument you pass into the `getFullName()` function.

If the argument has two properties `firstName` and `lastName` and their types are strings, then the TypeScript compiler passes the check. Otherwise, it'll issue an error.

The type annotation of the function argument makes the code difficult to read. To address this issue, TypeScript introduces the concept of interfaces.

The following uses an interface `Person` that has two string properties:

```
interface Person {
    firstName: string;
    lastName: string;
}
```

By convention, the interface names are in the PascalCase. They use a single capitalized letter to separate words in their names. For example, `Person` , `UserProfile` , and `FullName` .

After defining the `Person` interface, you can use it as a type. For example, you can annotate the function parameter with the interface name:

```
function getFullName(person: Person) {
    return `${person.firstName} ${person.lastName}`;
}

let john = {
    firstName: 'John',
    lastName: 'Doe'
};

console.log(getFullName(john));
```

The code now is easier to read than before.

To make the code more concise, you can use the object destructuring feature of JavaScript:

```
function getFullName({ firstName, lastName }: Person) {
  return `${firstName} ${lastName}`;
}
```

In the argument, we destructure the properties of the `person` object:

```
{ firstName, lastName }: Person
```

The `getFullName()` function will accept any object that has at least two string properties with the name `firstName` and `lastName`.

For example, the following code declares an object that has four properties:

```
let jane = {
  firstName: 'Jane',
  middleName: 'K.',
  lastName: 'Doe',
  age: 22,
};
```

Since the `jane` object has two string properties `firstName` and `lastName`, you can pass it on to the `getFullName()` function as follows:

```
let fullName = getFullName(jane);
console.log(fullName); // Jane Doe
```

Optional properties

An interface may have optional properties. To declare an optional property, you use the question mark (`?)` at the end of the property name in the declaration, like this:

```
interface Person {
    firstName: string;
    middleName?: string;
    lastName: string;
}
```

In this example, the `Person` interface has two required properties and one optional property.

And the following shows how to use the `Person` interface in the `getFullName()` function:

```
function getFullName(person: Person) {
    if (person.middleName) {
        return `${person.firstName} ${person.middleName} ${person.lastName}`;
    }
    return `${person.firstName} ${person.lastName}`;
}
```

Readonly properties

If properties should be modifiable only when the object is first created, you can use the `readonly` keyword before the name of the property:

```
interface Person {
  readonly ssn: string;
  firstName: string;
  lastName: string;
}

let person: Person;
person = {
  ssn: '171-28-0926',
  firstName: 'John',
  lastName: 'Doe',
};
```

In this example, the `ssn` property cannot be changed:

```
person.ssn = '171-28-0000';
```

Error:

```
error TS2540: Cannot assign to 'ssn' because it is a read-only property.
```

Function types

In addition to describing an object with properties, interfaces allow you to represent function types.

To describe a function type, you assign the interface to the function signature that contains the parameter list with types and returned types. For example:

```
interface StringFormat {
    (str: string, isUpper: boolean): string
}
```

Now, you can use this function-type interface.

The following illustrates how to declare a variable of a function type and assign it a function value of the same type:

```
let format: StringFormat;

format = function (str: string, isUpper: boolean) {
    return isUpper ? str.toLocaleUpperCase() : str.toLocaleLowerCase();
};

console.log(format('hi', true));
```

Output:

```
HI
```

Note that the parameter names don't need to match the function signature. The following example is equivalent to the above example:

```
let format: StringFormat;

format = function (src: string, upper: boolean) {
    return upper ? src.toLocaleUpperCase() : src.toLocaleLowerCase();
};

console.log(format('hi', true));
```

The `StringFormat` interface ensures that all the callers of the function that implements it pass in the required arguments: a `string` and a `boolean`.

The following code also works perfectly fine even though the `lowerCase` is assigned to a function that doesn't have the second argument:

```
let lowerCase: StringFormat;
lowerCase = function (str: string) {
    return str.toLowerCase();
}

console.log(lowerCase('Hi', false));
```

Notice that the second argument is passed when the `lowerCase()` function is called.

Class Types

If you have worked with Java or C#, you can find that the main use of the interface is to define a contract between classes.

For example, the following `Json` interface can be implemented by any class:

```
interface Json {
  toJson(): string;
}
```

The following declares a class that implements the `Json` interface:

```
class Person implements Json {
  constructor(private firstName: string, private lastName: string) {}
  toJson(): string {
    return JSON.stringify(this);
  }
}
```

In the `Person` class, we implemented the `toJson()` method of the `Json` interface.

The following example shows how to use the `Person` class:

```
let person = new Person('John', 'Doe');
console.log(person.toJson());
```

Output:

```
{"firstName":"John","lastName":"Doe"}
```

Summary

- TypeScript interfaces define contracts in your code and provide explicit names for type-checking.
- Interfaces may have optional properties or read-only properties.
- Interfaces can be used as function types.
- Interfaces are typically used as class types that make a contract between unrelated classes.