100 XP

Overview of interfaces in TypeScript

5 minutes

One of TypeScript's core principles is that type checking focuses on the shape that values have. This is sometimes called "duck typing" or "structural subtyping". In TypeScript, interfaces fill the role of naming these types, and are a powerful way of defining a "code contract" within your code as well as contracts with code outside of your project.

Let's start with an overview of interfaces, including what they are and how you can use them in your TypeScript code.

What is an interface

You can use interfaces to describe an object, naming and parameterizing the object's types, and to compose existing named object types into new ones.

This simple interface defines the two properties and a method of an `Employee` object.

TypeScript

```
interface Employee {
    firstName: string;
    lastName: string;
    fullName(): string;
}
```

Notice that the interface doesn't initialize or implement the properties declared within it. That's because the only job of an interface is to describe a type. It defines what the code contract requires, while a variable, function, or class that implements the interface satisfies the contract by providing the required implementation details.

After defining an interface, you can use it as a type and get all the benefits of type checking and Intellisense.

This example implements the interface by declaring a variable of the type `Employee`. It fulfills the contract by passing in values for the `firstName` and `lastName` properties and specifying

that the `fullName` method should combine the `firstName` and `lastName` properties and return the result.

TypeScript

```
let employee: Employee = {
    firstName : "Emil",
    lastName: "Andersson",
    fullName(): string {
        return this.firstName + " " + this.lastName;
    }
}

employee.firstName = 10;  //* Error - Type 'number' is not assignable to type
'string'
```

Type checking ensures that the number `10` is not assignable to `employee.firstName` because it is expecting a `string`.

Because TypeScript has a structural type system, an interface type with a particular set of members is considered identical to, and can be substituted for, another interface type or object type literal with an identical set of members. If an interface and a class implement the same structure, they can be used interchangeably. See Structural Typing in the Playground for an example.

Interfaces have no run-time representation; they are purely a compile-time construct. Interfaces are particularly useful for documenting and validating the required shape of properties, objects passed as parameters, and objects returned from functions. This enables you to catch errors and make sure you're passing the right parameters at compile time, rather than waiting to find out about them at runtime.

Reasons for using an interface in TypeScript

Interfaces are often the key point of contact between any two pieces of TypeScript code, especially when working with existing JavaScript code or built-in JavaScript objects.

You can use an interface to:

- Create shorthand names for commonly used types. With even a simple interface like the one declared in the earlier example, you still get the benefit of Intellisense and type checking.

- Drive consistency across a set of objects because every object that implements the interface operates under the same type definitions. This can be useful when you're working with a team of developers and you want to ensure that proper values are being passed into properties, constructors, or functions. For example, objects that implement an interface must implement all the **required** members of the interface. So, if you don't pass all the required parameters of the correct type, the TypeScript compiler will throw an error.
- Describe existing JavaScript APIs and clarify function parameters and return types. This is especially useful when you're working with JavaScript libraries like jQuery. An interface can provide you with a clear understanding of what a function is expecting and what it will return without repeat visits to the documentation.

How is an interface different from a type alias?

The `Employee` interface above can also be expressed as a type alias using the `type` keyword:

TypeScript

```
type Employee = {
    firstName: string;
    lastName: string;
    fullName(): string;
}
```

A type alias is a definition of a type of data, for example, a union, primitive, intersection, tuple, or any other type. Interfaces, on the other hand, are a way to describe data shapes, for example, an object. Type aliases can act like interfaces; however, there are some subtle differences. The key distinction is that a type alias cannot be reopened to add new properties whereas an interface is always extendable. Also, you can only describe a union or tuple using a type alias.

Next unit: Exercise - Declare and instantiate an interface in TypeScript

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆