✓ 100 XP

Other ways to use interfaces in Typescript

5 minutes

Now that you know the basics of declaring and implementing an interface, let's look at some other ways you can use them.

Create indexable types

You can use interfaces that describe array types that you can index into.

Indexable types have an **index signature** that describes the type you can use to index into the object, along with the corresponding return types when indexing.

For example, the `IceCreamArray` interface declares an index signature as a `number` and returns a `string` type. This index signature states that `IceCreamArray` is indexed with a number and it will return a string.

TypeScript

```typescript
interface IceCreamArray {
    [index: number]: string;
}

let myIceCream: IceCreamArray;
myIceCream = ['chocolate', 'vanilla', 'strawberry'];
let myStr: string = myIceCream[0];
console.log(myStr);
```

You can also use the built-in array type or create a type alias for a custom array, but by using an interface, you can define your own array type so that anyone who wants to work with that interface can apply it consistently.

Describe a JavaScript API using an interface

A common pain point for JavaScript and TypeScript developers alike is working with external JavaScript libraries. You can use an interface to describe existing JavaScript APIs and clarify function parameters and return types. The interface provides you with a clear understanding of what an API is expecting and what it will return.

The `fetch` API is a native JavaScript function that you can use to interact with web services. This example declares an interface called `Post` for the return types in a JSON file and then uses `fetch` with `async` and `await` to generate a strongly typed response.

TypeScript

```typescript
const fetchURL = 'https://jsonplaceholder.typicode.com/posts'
// Interface describing the shape of our json data
interface Post {
    userId: number;
    id: number;
    title: string;
    body: string;
}
async function fetchPosts(url: string) {
    let response = await fetch(url);
    let body = await response.json();
    return body as Post[];
}
async function showPost() {
    let posts = await fetchPosts(fetchURL);
    // Display the contents of the first item in the response
    let post = posts[0];
    console.log('Post #' + post.id)
    // If the userId is 1, then display a note that it's an administrator
    console.log('Author: ' + (post.userId === 1 ? "Administrator" :
post.userId.toString()))
    console.log('Title: ' + post.title)
    console.log('Body: ' + post.body)
}

showPost();
```

Using `async/await` are beyond the scope of this learning path. If you want to learn more about it, see these videos from the Beginner's Series to JavaScript course:

- Using async/await in JavaScript with long running operations
- async/await for managing promises
- Demo: async/await for managing promises

Tip

While earlier versions of ECMAScript, such as ES3, do not support `async` and `await`, the TypeScript compiler is able to generate compatible code to implement this feature. This enables you to take advantage of the newer feature while still being able to target older browsers! In the Playground, copy and paste the example above, set the target to ES3, and check out the helper code that TypeScript generates to make this possible.

Next unit: Lab - Use interfaces in TypeScript

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆