RxJS and TypeScript: `pipe`

This lesson looks into typing the `pipe` function in RxJS as an example of typing complex variadic functions.

WE'LL COVER THE FOLLOWING

Overview

If you're familiar with the RxJS library, you should be accustomed with the concept of **pipeable operators**. It's a mechanism of composing operators (applying an operator on top of other operators) that replaced chained method calls (`stream$.map(...).filter(...)`) to make the library tree-shakable. It's based on the idea of function composition from functional programming. Looking into how `pipe` is typed is a good exercise in advanced typing.

What is piping?

First, let's look at some examples of using `pipe`.

```
const number$ = of(1, 2, 3, 4, 5).pipe(
    filter(x => x % 2 === 0),
    map(x => x * x)
);

const response$ = fromEvent(button, 'click').pipe(
    debounceTime(500),
    tap(() => { console.log('fetching data...'); }),
    switchMap(()=> fetch(`http://example.com/api/status`)),
    switchMap(response => response.json()),
```

```
    catchError(() => empty())
);
```

As you can see, `pipe` can be called with a different number of arguments. What's more, these arguments can be very diverse as any RxJS operator, with any number of parameters, can be used. Finally, the type of each argument depends on the type of the previous argument. For example, the type of the function provided to the second `switchMap` is `(value: Response) => Promise<any>` because the function provided to the first `switchMap` returns a `Response`.

Typing `pipe`

Let's try to sketch the type of `pipe`. First of all, how do we deal with the fact that any number of arguments is provided? Although TypeScript allows us to type simple functions accepting any number of parameters (e.g. `(...numbers: number[]) => number`), this mechanism won't work in this case. The problem is that, as already mentioned, the type of each parameter is different and may depend on the type of the previous parameter. We'll deal with this problem by providing a number of function overloads, each with a different number of parameters.

Next, let's think about the types of parameters. Each parameter represents an operator. An RxJS operator is a function that takes a stream and returns a stream. The type of values in the returned stream might differ from the type of elements in the source stream. What's more, the type of elements in the source stream must be the same as the type of elements in the stream returned by the previous operator.

Given all these points, the types for `pipe` with up to three parameters might look like this:

```
function pipe<S, R>(
    operator1: (s1: Observable<S>) => Observable<R>,
): Observable<R>;
function pipe<S, T1, R>(
    operator1: (s1: Observable<S>) => Observable<T1>,
    operator2: (s2: Observable<T1>) => Observable<R>,
): Observable<R>;
function pipe<S, T1, T2, R>(
    operator1: (s1: Observable<S>) => Observable<T1>,
    operator2: (s2: Observable<T1>) => Observable<T2>,
```

```
    operator3: (s3: Observable<T2>) => Observable<R>,
): Observable<R>;

function pipe(...operators: ((s1: Observable<unknown>) => Observable<unkno
wn>)[]): Observable<unknown> { /* ... */ }
```

As you can see, function overloads in TypeScript are created by providing several function signatures and one implementation. When `pipe` is called, TypeScript will look for a matching overload and use its types. If it cannot find a matching overload, it will fall back to the type of implementation.

Each overload is a generic function with type arguments representing the type of elements in the source observable (`S`), element types of interim observables (`T1`, `T2`, ...), and the element type of the resulting observable (`T4`). Each argument is a function that takes an observable of one type and returns an observable of the "next" type.

`pipe` in RxJS

Let's now take a look at how `pipe` is implemented in RxJS. We'll look at the `pipe` *method* as opposed to the `pipe` *function*.

```
export declare class Observable<T> implements Subscribable<T> {
    pipe<A>(op1: OperatorFunction<T, A>): Observable<A>;
    pipe<A, B>(
        op1: OperatorFunction<T, A>,
        op2: OperatorFunction<A, B>,
    ): Observable<B>;
    pipe<A, B, C>(
        op1: OperatorFunction<T, A>,
        op2: OperatorFunction<A, B>,
        op3: OperatorFunction<B, C>,
    ): Observable<C>;
    pipe<A, B, C, D>(
        op1: OperatorFunction<T, A>,
        op2: OperatorFunction<A, B>,
        op3: OperatorFunction<B, C>,
        op4: OperatorFunction<C, D>,
    ): Observable<D>;
    /* ... */
}
```

As you can see, these definitions follow exactly the same pattern. The only difference is that instead of explicitly mentioning the function type, it uses an

alias `OperatorFunction`, which makes the whole thing slightly more readable.