Generic Interfaces - FP Perspective

This lesson talks about generic interfaces from the perspective of Functional Programming.

WE'LL COVER THE FOLLOWING

Overview

Unlike OOP, objects don't have behavior in Functional Programming. Interfaces are used not to enumerate methods of an object, but to describe the shape of the data contained by the object.

In this context, generic interfaces are used to describe a data shape when you don't know or care about the exact type of some properties of the interface. It often makes sense when the data types *contain* some value.

You can find an example of such an interface below. It represents a form field that could be part of some dynamic form implementation. We want to store certain information about a form field while we don't care about the type of data stored in that field. That's why we introduced the generic type parameter `T`.

```
interface FormField<T> {
    value?: T;
    defaultValue: T;
    isValid: boolean;
}
```

Working with generic data shapes

It's not possible to refer to a generic interface without providing its type argument. For example, if you want to write a function that accepts an

instance of a generic interface, you have two options:

- Write a specialized function that accepts a very specific instance of the interface (e.g. `FormField<string>`)
- Write a *generic* function

```
interface FormField<T> {
    value?: T;
    defaultValue: T;
    isValid: boolean;
}

// Very specialized. Only works with `FormField<string>`.
function getStringFieldValue(field: FormField<string>): string {
    if (!field.isValid || field.value === undefined) {
        // Thanks to the specialization, the compiler knows the exact type of `field.defaultV
        return field.defaultValue.toLowerCase();
    }
    return field.value;
}

// Generic. Can be called with any `FormField`.
function getFieldValue<T>(field: FormField<T>): T {
    if (!field.isValid || field.value === undefined) {
        // On the other hand, we don't know anything about the type of `field.defaultValue`.
        return field.defaultValue;
    }
    return field.value;
}
```

Hover over different variables to see how their types differ in both versions.

It's important to understand that in the second example (`getFieldValue`) `T` is the type argument **of the function**, not of the `FormField` interface. It gets passed only to `FormField` like any regular type does.

The next lesson talks about type argument constraints in generic types.