Mapped Types Introduction

This lesson introduces the concept of mapped types.

WE'LL COVER THE FOLLOWING

- Overview
- Mapped type example - `Readonly` type
- Deep immutability

Overview

Mapped types are another way to transform types. In simple words, they let you take an object type (an interface) and return a new type with each property transformed in some way.

Mapped type example - `Readonly` type

The best example for understanding mapped types is the built-in `Readonly` type. As you can see, `Readonly` is a generic type. It takes an object type `T`, enumerates its properties (`P in keyof T`), and adds the `readonly` modifier to each property. You can think of mapped types as an equivalent of the `for..of` loop in the world of types.

```
interface Person {
    name: string;
    age: number;
}

type Readonly<T> = {
    readonly [P in keyof T]: T[P];
};

type ReadonlyPerson = Readonly<Person>;
```

Hover over `ReadonlyPerson` to see the inferred type.

The syntax reminds the one for defining a regular interface. However, instead of specifying several properties, you define all of them at once by using the `[P in keyof T]` syntax (*index type query operator*). `P` becomes the literal type representing each property, and the value of each property is simply `T[P]` (*indexed access operator*).

Deep immutability

`Readonly` type is very helpful when dealing with immutable data structures (e.g., when working with Redux). Bear in mind that it only provides shallow immutability, meaning nested properties will not become `readonly`. You can achieve better results with the following `DeepReadonly` type.

```
type DeepReadonly<T> = {
  readonly [P in keyof T]: DeepReadonly<T[P]>;
};

type Employment = DeepReadonly<{
    person: Person;
    company: string;
}>;

interface Person { name: string; }

declare const employment: Employment;
employment.person.name = 'Milosz'; // ⚪ Error!
```

Run the code to see the error.

As you can see, mapped types can be recursive. For each property of `T` we map it to a read-only property and we declare its type as `DeepReadonly`. If the original property type is an object type, its properties will also be marked with `readonly`. Otherwise, the type will stay unmodified.

Unfortunately, this type does not provide immutability for nested arrays, Maps, Sets, or some other types. It's actually quite complex to properly implement such types, so it's best to use a library instead (like this one).

The next lesson walks you through several examples of built-in mapped types.