Discriminated Union Types

This lesson introduces the concept of discriminated union types.

WE'LL COVER THE FOLLOWING

- Overview
- Representing choice with optional properties
- Discriminated unions to the rescue
- Note on formatting

Overview

Now that you have a full understanding of union types, let's look at one of their special categories.

Creating **discriminated unions** is an extremely powerful way of composing types. Thanks to this mechanism you can write code that is type-safe beyond your imagination. Discriminated unions let you enforce some business logic rules **at compile-time**.

Representing choice with optional properties

Let's start by looking at an example. Imagine implementing an e-commerce application. When the customer registers on the app, they need to provide some contact details. One of the acceptance criteria says:

> Customer needs to provide either email or phone number.

The most straightforward way of defining the `Customer` type that satisfies this requirement is by using two optional properties.

```
interface Customer {
```

```
  name: string;
  email?: string;
  phone?: number;
}
```

This solution has one major drawback. It's possible to create a `Customer` with neither `email` nor `phone` provided. In other words, it's possible to create a `Customer` object that violates the business requirement.

You might argue that the type will be accompanied by a validation function that will throw an exception if the object doesn't have any of these two properties defined. This is slightly better, but still far from ideal because of the following:

- You will learn that the object is invalid at runtime
- When using `strictNullChecks`, you'll need to write `if` statements every time you want to access `email` or `phone`
- You might forget to call the validation function

Discriminated unions to the rescue

Instead of having two optional properties in `Customer` type, let's create a new `Contact` type that is a union of two interfaces. One union member has a non-optional `email` property and the other has a non-optional `phone` property.

These two union members follow a special convention. They both have a string literal property called `kind`. The type of that property is different in both members. This property is called a **discriminator**. It encodes type information into the object so that it is available at runtime. Thanks to this, TypeScript can figure out which union member you're dealing with by looking at the `kind` property.

```
interface EmailContact {
    kind: 'email';
    email: string;
}

interface PhoneContact {
    kind: 'phone';
    phone: number;
}

type Contact = EmailContact | PhoneContact;
```

```
interface Customer {
    name: string;
    contact: Contact;
}

function printCustomerContact({ contact }: Customer) {
    if (contact.kind === 'email') {
        // Type of `contact` is `EmailContact`!
        console.log(contact.email);
    } else {
        // Type of `contact` is `PhoneContact`!
        console.log(contact.phone);
    }
}
```

Hover over `contact` inside both branches of the `if` statement to see how the type is narrowed.

Here comes the power of discriminated unions. TypeScript analyses the code and sees an `if` statement that checks whether `contact.kind` is equal to `"email"`. If that's true, it can be certain that the type of `contact` is `EmailContact`, so it narrows the type inside the first `if` branch. The only other option for `contact` is to be a `PhoneContact`, so the type is narrowed to `PhoneContact` in the second `if` branch.

You might be worried about the usage of a *magic string* in the `if` statement. In fact, the string is not really *magic*. The type of `contact.kind` (outside of the `if`) is `'email' | 'phone'`. Try it yourself and see how the code editor suggests that the only possible values for `contact.kind` are `'email'` or `'phone'`.

We've made an illegal state unrepresentable. In the previous solution, it was possible to create a customer without phone or email (i.e., it was possible to create an illegal piece of application state). Now it's forbidden at compile time!

Note on formatting

`Contact` type can be written in a much terser way as it's not necessary to create standalone `PhoneContact` and `EmailContact` types - they can be inlined.

```
type Contact =
    | { kind: 'email', email: string }
    | { kind: 'phone', phone: number };
```

The first `|` has been added only for formatting purposes. It's allowed by TypeScript syntax and is a common convention to use when defining multi-

line unions.

The next lesson dives deeper into using discriminated unions.