Discriminated Unions in Practice

This lesson contains some facts about discriminated unions that may come in handy when using them in the wild.

WE'LL COVER THE FOLLOWING

Non-string discriminators

In the example from the previous lesson, we used a string literal property called `type` as a discriminator. First, the discriminator doesn't have to be called `"type"`. Any property name will be fine.

Second, the discriminator doesn't have to be a string. You can also use number literals, Boolean literals, or even enums!

The built-in `IteratorResult` type in TypeScript (starting from version 3.6) uses the Boolean property `done` as a discriminator.

```
type IteratorResult<T, TReturn = any> = IteratorYieldResult<T> | IteratorR
eturnResult<TReturn>;

interface IteratorYieldResult<TYield> {
    done?: false;
    value: TYield;
}

interface IteratorReturnResult<TReturn> {
    done: true;
    value: TReturn;
}
```

Using enums can help you avoid *magic strings* if that's an issue for you. One advantage of this approach is that renaming discriminating values becomes much easier.

```
const enum ContactType { Phone, Email };

type Contact =
    | { type: ContactType.Email, email: string }
    | { type: ContactType.Phone, phone: number };
```

Multiple discriminating properties

Interestingly, a discriminator does not have to be a single property. A group of literal properties can also act as a discriminator! In such a case, every combination of values marks a different member of the union type.

```
type Foo =
  | { kind: 'A', type: 'X', abc: string }
  | { kind: 'A', type: 'Y', xyz: string }
  | { kind: 'B', type: 'X', rty: string }

declare const foo: Foo;

if (foo.kind === 'A' && foo.type === 'X') {
  console.log(foo.abc);
}
```

Hover over `foo` inside the `if` statement to see how its type is narrowed.

Algebraic data types

Discriminated unions are not unique to TypeScript. Statically-typed functional languages often make heavy use of *algebraic data types*. You can find an example from Haskell compared with a TypeScript equivalent below.

```
data Tree = Empty
          | Leaf Int
          | Node Tree Tree
```

```
type Tree =
      | { type: 'empty' }
      | { type: 'leaf', value: number }
```

```
    | { type: 'node', left: Tree, right: Tree };
```

One might argue that discriminated unions do not feel natural because of the necessity of adding the extra property. Traditional statically typed languages (like Haskell) encode type information in runtime values. However, since TypeScript compiles to JavaScript, all the type information is lost at runtime. You need to add the information yourself, hence the extra property. TypeScript just happens to understand this convention and utilizes it when analyzing your code.

Excercise: Precise domain modeling

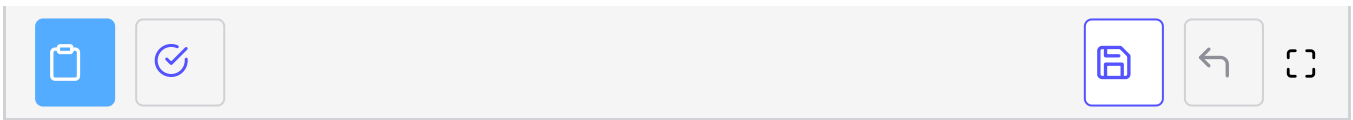
So far, we have seen an example of how discriminated unions help encode business rules in the type system. In this exercise, you're going to write types that capture a slightly more complex domain.

Create types for the following domain:

> Users place orders for products. Users have contact information, email or postal addresses, and at least one is required. Orders should include price, product name, quantity, payment date, paid amount, sending date, and delivery date.

*Note that the system can only verify that your code compiles without errors. To check your solution, click on **Show solution** and compare the code.*

```
type Order = {
  productName: string;
  price: number;
  quantity: number;
};
```



The next lesson talks about how discriminated unions can be leveraged to build state machines in TypeScript.