Discriminated Unions vs Subtyping

This lesson discusses the differences between the two approaches of expressing choice in the type system.

WE'LL COVER THE FOLLOWING

- Overview
- Defining types
- Adding an operation
- Comparison

Overview

Subtyping is a form of type polymorphism that is very popular in Object-Oriented Programming. If you're familiar with OOP languages such as Java or C#, then you very likely know it already.

Subtyping offers an alternative to Algebraic Data Types (discriminated unions). In this lesson, I'd like to compare these two approaches.

Defining types

Let's go back to the example of a discriminated union representing a binary tree. It could, for example, be a tree of product categories in an e-commerce web application.

```
type Tree =
        | { type: 'empty' }
        | { type: 'leaf', value: number }
        | { type: 'node', left: Tree, right: Tree };
```

A tree can either be:

- a node with references to left and right subtrees
- a leaf containing a numerical value

- a leaf containing a numerical value
- empty

We can represent the same data structure using the OOP approach. We define an abstract class `Tree` and three concrete classes extending from `Tree`.

```
abstract class Tree {}

class Empty extends Tree { }

class Leaf extends Tree {
  constructor(private value: number) {
    super();
  }
}

class Node extends Tree {
  constructor(
    private left: Tree,
    private right: Tree,
  ) {
    super();
  }
}
```

Adding an operation

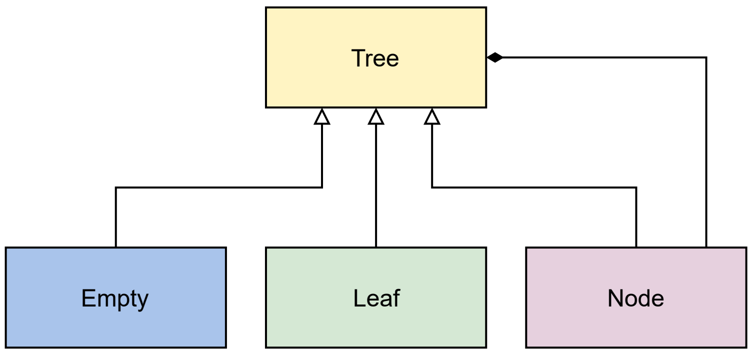
Now, let's say that we want to create a textual representation of the tree. Let's see how the implementation would look in both approaches.

```
type Tree =
      | { type: 'empty' }
      | { type: 'leaf', value: number }
      | { type: 'node', left: Tree, right: Tree };

function toString(tree: Tree): string {
    switch (tree.type) {
        case 'empty':
            return 'Empty';
        case 'leaf':
            return `Leaf(${tree.value})`;
        case 'node':
            return `Node(${toString(tree.left)}, ${toString(tree.right)})`;
    }
}
```

Hover over `tree` inside `toString` to see how its type is narrowed.

This `toString` function looks very similar to what you've seen in this chapter. It switches over the discriminator of `Tree` type and returns a different string for each case. Note that the type of `tree` is narrowed in each `case` of the `switch` statement.

```
abstract class Tree {
    abstract toString(): string;
}

class Empty extends Tree {
    toString() {
        return `Empty`;
    }
}

class Leaf extends Tree {
  constructor(private value: number) {
    super();
  }

  toString() {
      return `Leaf(${this.value})`;
  }
}

class Node extends Tree {
  constructor(
    private left: Tree,
    private right: Tree,
  ) {
    super();
  }

  toString() {
      return `Node(${this.left.toString()}, ${this.right.toString()})`;
  }
}
```



UML diagram representation of the above class hierarchy.

As you can see, the OOP approach is very different. Instead of defining a standalone function, we've added an abstract method to the `Tree` abstract class. By doing that, we declare that all implementations of `Tree` have to implement this method. What's more, when calling `toString` on a `Tree` object, the correct implementation will be chosen at runtime based on its concrete type. There is no `switch` statement. Instead, we're taking advantage of polymorphism in JavaScript.

Each concrete class provides its own implementation of `toString`. The implementations are very similar to those used in specific `case` statements in the previous approach, except they refer to `this` instead of the function argument.

Comparison

As you can see, both approaches can be used to achieve the same result. However, there are a few major differences. First, the OOP approach is much more verbose. As you can see, much more code had to be written to achieve the same result.

The most important difference is how these two approaches cope with change. Imagine that we want to support a new operation on the `Tree` (e.g., `sum` that returns the sum of all nodes in the tree). In OOP approach, we need to modify every concrete class extending from `Tree`. This can be difficult, especially if we don't own all of these classes. In the ADT approach, all we need to do is write another standalone function with a `switch` statement.

On the other hand, imagine that we want to add a new type of tree node (e.g., `NodeWithValue`). As a consequence, we need to modify all `Tree` operations. In the OOP approach, all we need to do is create a new subclass (`NodeWithValue`) and implement all abstract methods. However, in the ADT approach, we have to add a new `case` to every `switch` statement that switches over the discriminator property. It might be tricky, especially if we don't own all of the operations.

As you can see, both approaches react to change differently. I personally prefer the ADT approach, since it's terser and more in line with the functional programming paradigm.