Aliases with the Structural Behavior of TypeScript

This lesson demonstrates the foundamental aspect of TypeScript to rely on objects' structures.

WE'LL COVER THE FOLLOWING

- ^
- The structural nature of TypeScript
- Adding an additional field
- Improving our code

The structural nature of TypeScript

TypeScript is a **structural language** and *not* a nominal language. This means that TypeScript compares the structure and not the name of an object to determine if the objects are similar.

The following code has a function that takes an anonymous type that has a property name. The example has three ways to call the function:

- 1. interface,
- 2. type, and
- 3. anonymous object.

It works in all three scenarios because they have the exact shape needed.

In the following code, there is MyTypeA that is defining a schema for an object with a single member name at line 2. At line 4, the type is the same which is also equivalent to the anonymous parameter at line 6.

Lines 10-11 define the two types and at **lines 12-13** invoke the function even if not the anonymous parameter. It works because they have the same structure. TypeScript is not nominal, but a structural language. Hence, at **line 14** the function can also be invoked.

```
interface MyTypeA {
   name: string; // Same property name-type than line 4 and 6
}

type MyTypeB = { name: string }; // Notice the name property is the same as the anonymous at

function ouputName(obj: { name: string }): void { // Anonymous type for first parameter
   console.log(obj.name);
}

let typeA: MyTypeA = { name: "A" }; // TypeA

let typeB: MyTypeB = { name: "B" }; // TypeB

ouputName(typeA);
ouputName(typeB);
ouputName({ name: "C" }); // Also anonymous
```

Adding an additional field

What is important to understand is that it does not need to match perfectly the desired type. The type must be at least structurally fully compatible but can have additional fields. The following example has InterfaceB having the additional id at line 7. Even if the field is not required at line 13 the invocation compiles because of the structural behavior of TypeScript.

```
interface InterfaceA {
    name: string;
}

interface InterfaceB {
    name: string;
    id: number;
}

let intA: InterfaceA = { name: "A" };
let intB: InterfaceB = { name: "B", id: 2 };

function ouputName(obj: { name: string }): void {
    console.log(obj.name);
}

ouputName(intA);
ouputName(intB);
```

An alternative of the code above is:

```
name: string;
}
interface InterfaceB {
  name: string;
  id: number;
}
let intA: InterfaceA = { name: "A" };
let intB: InterfaceB = { name: "B", id: 2 };
function outputInterfaceA(obj: InterfaceA): void { // Not anonymous console.log(obj.name);
}
outputInterfaceA(intA);
outputInterfaceA(intB); // Interface B can substitute for Interface A
```

Even though the InterfaceA is required at line 13 and that InterfaceB does not implement or extends InterfaceA, it transpiles because it has all the structural fields (name: string).

Improving our code

That being said, a better code would remove the redundancy by extending InterfaceA.

```
interface InterfaceA {
   name: string;
}

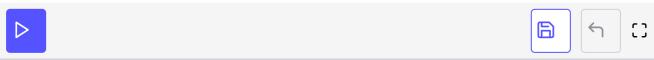
interface InterfaceB extends InterfaceA {
   id: number;
}

let intA: InterfaceA = { name: "A" };

let intB: InterfaceB = { name: "B", id: 2 };

function outputInterfaceA(obj:InterfaceA): void {
   console.log(obj.name);
}

outputInterfaceA(intA);
outputInterfaceA(intB);
```



In this lesson, we saw that what matter to TypeScript is to fulfill a structural

contract, not a nominal one. This means we can create an alias or a shortcut, to different structures as we wish. In the upcoming lesson, we will see different kinds of aliases.