On How TypeScript Handles Variance

This lesson gives the theory on how TypeScript handles the variance.

WE'LL COVER THE FOLLOWING

Type system variances

TypeScript variances use the nomenclature of another language, which is confusing by nature. It's not something you must master since the compiler will stop you from doing anything crazy.

However, it's good to understand before having the compiler manage the complexity for us. Before getting started, let's create some interfaces to examine how TypeScript handles (or not) covariant, contravariant, bivariant and invariant types.

We will create three interfaces which will be all inherited in the chain. So, `InterfaceA` inherits `InterfaceB` which inherits `InterfaceC`.

```
interface InterfaceA {
  a1: number;
}

interface InterfaceB extends InterfaceA {
  b2: string;
}

interface InterfaceC extends InterfaceB {
```

```
  c3: boolean;
}
```

Type 1: Invariance

The concept of invariance is the simplest of the type systems. This means: the variable's type is the only type accepted.

Invariance *accepts* supertypes; it also does *not* accept subtypes.

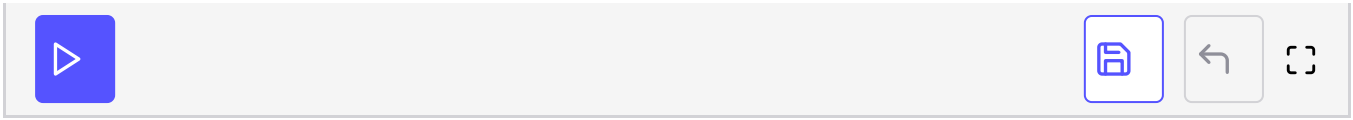
```
interface InterfaceA {
  a1: number;
}

interface InterfaceB extends InterfaceA {
  b2: string;
}

interface InterfaceC extends InterfaceB {
  c3: boolean;
}

let a: InterfaceA = { a1: 1 };
let b: InterfaceB = { a1: 1, b2: "2" };
let c: InterfaceC = { a1: 1, b2: "2", c3: true };

b = a; // Error, does not accept super type
b = c; // Accept subtype
```



With our three interfaces in the previous block of code, if a function had the type `InterfaceB`, that will be the only interface that is acceptable. To have an invariant language, you need a *sound* language, which TypeScript is not, since it is a *structural* language. This explains why **line 17** does not transpile.

Type 2: Covariance

TypeSCript is not *invariant* as we discussed in the previous paragraph, so what is it? TypeScript relies on *covariance*. Covariance accepts supertypes but not subtypes. In our example, a function accepting type `InterfaceB` would only accept `InterfaceB` and `InterfaceC`. However, it does not accept a supertype like `InterfaceA`.

Type 3: Contravariance

Contravariance is the opposite of covariance: it accepts the type and subtype.

Contravariance is the opposite of covariance, it accepts the type and subtype. `InterfaceA` and InterfaceB would be accepted to a variable with type

`InterfaceC`

Type 4: Bivariance

Finally, bivariance is the only option that allows everything from the chain.

The important notion of this lesson is that TypeScript uses covariance.