The Differences between Type Aliases and Interfaces

This lesson highlights the differences between type aliases and interfaces.

WE'LL COVER THE FOLLOWING

As TypeScript evolves, the differences between type aliases and interfaces have diminished. Nevertheless, few differences are worth pointing out.

Difference with classes

It is only ever possible to implement an interface on a class, but never a type. At any time an interface is implementable, but a type cannot have a union.

This makes the `type` a little more tricky to reuse in some patterns. The following example works because there is not a union **at line 1**. Thus, **line 6** transpiles. **Line 10** remains legit because an interface always can be implemented by a class.

```
type Person = { name: string };
interface Person2 {
  name: string;
}

class Mother implements Person {
  name: string = "";
}

class Father implements Person2 {
  name: string = "";
}
```

The following code does not compile because the type is a union at **line 1**. A union means a type *or* another type making the implementation not deterministic. Hence, it makes sense that it does not work.

> 📜**Note:** the below code throws an error ❌

```
type Person = { name: string } | string;

class Mother implements Person {
  name: string = "";
}
```

Difference with interfaces

An interface can extend any interface. An interface can extend type but not all of them. Again, a type with a union is not extendable. It makes the interface easier to work with the interface because it does not need validation that the underlying type has a union or not.

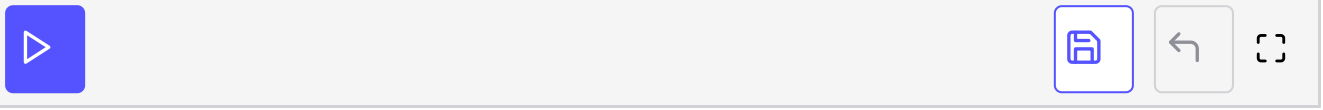
```
interface Person { name: string };
interface Father extends Person { numberOfChildren: number };
let f1: Father = { name: "MisterF1", numberOfChildren: 2 };
```

But the following code transpiles because it does not have union.

```
type Person = { name: string };
interface Father extends Person { numberOfChildren: number };
let f1: Father = { name: "MisterF1", numberOfChildren: 2 };
```

But not the following one.

```
type Person = { name: string } | string;
interface Father extends Person { numberOfChildren: number };
let f1: Father = { name: "MisterF1", numberOfChildren: 2 };
```



Accumulative difference

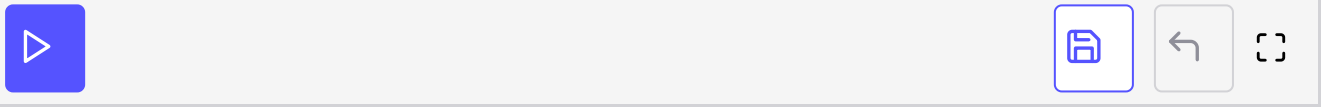
The interface allows accumulative properties among all declaration of the same-named interfaces. The type does not allow it and will throw a transpilation exception. The accumulative properties allow creating in the same or many different files the same interface much time to have a final merged type. This characteristic is not available with type. To have a merged type we new type will be needed. The main advantage of the interface is that it allows a third-party to expand existing code by third-party without changing the original code.

The following code defines the `Person` interface twice: **line 1** and **line 5**. In this lesson, they are next to each other, but they could have been in two different packages. At **line 9**, to compile, the code must contain both fields since they are both not marked as optional.

```
interface Person { // This interface could be in a third party package
  firstName: string;
}

interface Person { // This interface could be in your first party
  lastName: string;
}

let c: Person = { firstName: "First", lastName: "Last" };
```



The following code does not compile because it is two `type` and it does not allow two of the same name.

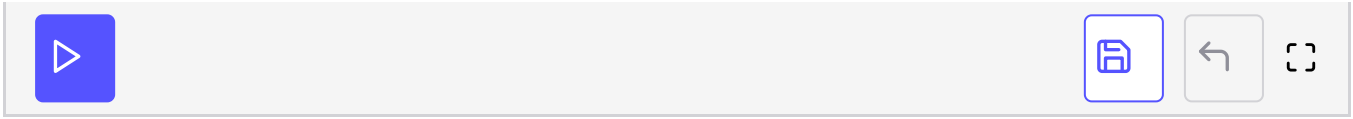
> 📜Note: the below code throws an error ✕

```
```

```
type PersonType = {
  firstName: string;
}


type PersonType {
  lastName: string;
}

let typeP: PersonType = { firstName: "First", lastName: "Last" };
```



Interface gives more flexibility and thus should be used most of time.