TypeScript - Interfaces

- Interface is a structure that is defined as a blueprint for an application.
- It defines the syntax for classes to follow. Classes that are derived from an interface must follow the structure provided by their interface.
- The TypeScript compiler does not convert the interface to JavaScript. It uses an interface for type checking. This is also known as **"duck typing"** or **"structural subtyping".**

An interface is defined with the keyword `interface` and it can include properties and method declarations using a function or an arrow function.

JavaScript

```
interface IEmployee {
    empCode: number;
    empName: string;
    getSalary: (number) => number; // arrow function
    getManagerName(number): string;
}
```

- In the above example, the `IEmployee` interface includes two properties `empCode` and `empName`.
- It also includes a method declaration `getSalaray` using an arrow function which includes one number parameter and a number return type.
- The `getManagerName` method is declared using a normal function.
- This means that any object of type `IEmployee` must define the two properties and two methods.

Extending Interfaces

Interfaces can extends one or more interfaces. This makes writing interfaces flexible and reusable.

JavaScript

```
interface IPerson {
    name: string;
    gender: string;
}

interface IEmployee extends IPerson {
    empCode: number;
}

let empObj:IEmployee = {
    empCode:1,
    name:"Bill",
    gender:"Male"
}
```

In the above example, the `IEmployee` interface extends the `IPerson` interface. So, objects of `IEmployee` must include all the properties and methods of the `IPerson` interface otherwise, the compiler will show an error.

Implementing an Interface

Similar to languages like Java and C#, interfaces in TypeScript can be implemented with a Class. The Class implementing the interface needs to strictly conform to the structure of the interface.

JavaScript

```
interface IEmployee {
    empCode: number;
    name: string;
    getSalary:(empCode: number) => number;
}

class Employee implements IEmployee {
    empCode: number;
    name: string;
```

```
    constructor(code: number, name: string) {
        this.empCode = code;
        this.name = name;
    }

    getSalary(empCode:number):number {
        return 20000;
    }
}

let emp = new Employee(1, "Steve");
```

In the above example, the `IEmployee` interface is implemented in the Employee class using the the implement keyword. The implementing class should strictly define the properties and the function with the same name and data type. If the implementing class does not follow the structure, then the compiler will show an error.

Of course, the implementing class can define extra properties and methods, but at least it must define all the members of an interface.

Interface as Type

- Interface in TypeScript can be used to define a type and also to implement it in the class.
- The following interface `IEmployee` defines a type of a variable.

JavaScript

```
interface KeyPair {
    key: number;
    value: string;
}

let kv1: KeyPair = { key:1, value:"Steve" }; // OK
let kv2: KeyPair = { key:1, val:"Steve" }; // Compiler Error: 'val'
doesn't exist in type 'KeyPair'
let kv3: KeyPair = { key:1, value:100 }; // Compiler Error:
```

In the above example, an interface `KeyPair` includes two properties `key` and `value`.

- A variable `kv1` is declared as `KeyPair` type. So, it must follow the same structure as `KeyPair`. It means only an object with properties `key` of number type and `value` of string type can be assigned to a variable `kv1`. The TypeScript compiler will show an error if there is any change in the name of the properties or the data type is different than `KeyPair`.
- Another variable `kv2` is also declared as `KeyPair` type but the assigned value is `val` instead of `value`, so this will cause an error.
- In the same way, kv3 assigns a number to the `value` property, so the compiler will show an error. Thus, TypeScript uses an interface to ensure the proper structure of an object.