TypeScript - Interfaces

Interface is a structure that defines the contract in your 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 interface to JavaScript. It uses 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](https://www.tutorialsteacher.com/typescript/arrow-function).

Example: Interface

 Copy

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.

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.

Example: Interface as Type

 Copy

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.

Interface as Function Type

TypeScript interface is also used to define a type of a function. This ensures the function signature.

Example: Function Type

 Copy

interface KeyValueProcessor

{

(key: number, value: string): void;

};

function addKeyValue(key:number, value:string):void {

console.log('addKeyValue: key = ' + key + ', value = ' + value)

}

function updateKeyValue(key: number, value:string):void {

console.log('updateKeyValue: key = '+ key + ', value = ' + value)

}

let kvp: KeyValueProcessor = addKeyValue;

kvp(1, 'Bill'); //Output: addKeyValue: key = 1, value = Bill

kvp = updateKeyValue;

kvp(2, 'Steve'); //Output: updateKeyValue: key = 2, value = Steve

In the above example, an interface KeyValueProcessor includes a method signature. This defines the function type. Now, we can define a variable of type KeyValueProcessor which can only point to functions with the same signature as defined in the KeyValueProcessor interface. So, addKeyValue or updateKeyValue function is assigned to kvp. So, kvp can be called like a function.

Trying to assign a function with a different signature will cause an error.

function delete(key:number):void {

console.log('Key deleted.')

}

let kvp: KeyValueProcessor = delete; //Compiler Error

Interface for Array Type

An interface can also define the type of an array where you can define the type of index as well as values.

Example: Type of Array

 Copy

interface NumList {

[index:number]:number

}

let numArr: NumList = [1, 2, 3];

numArr[0];

numArr[1];

interface IStringList {

[index:string]:string

}

let strArr : IStringList;

strArr["TS"] = "TypeScript";

strArr["JS"] = "JavaScript";

In the above example, interface NumList defines a type of array with index as number and value as number type. In the same way, IStringList defines a string array with index as string and value as string.

Optional Property

Sometimes, we may declare an interface with excess properties but may not expect all objects to define all the given interface properties. We can have optional properties, marked with a "?". In such cases, objects of the interface may or may not define these properties.

Example: Optional Property

 Copy

interface IEmployee {

empCode: number;

empName: string;

empDept?:string;

}

let empObj1:IEmployee = { // OK

empCode:1,

empName:"Steve"

}

let empObj2:IEmployee = { // OK

empCode:1,

empName:"Bill",

empDept:"IT"

}

In the above example, empDept is marked with ?, so objects of IEmployee may or may not include this property.

Read only Properties

TypeScript provides a way to mark a property as read only. This means that once a property is assigned a value, it cannot be changed!

Example: Readonly Property

 Copy

interface Citizen {

name: string;

readonly SSN: number;

}

let personObj: Citizen = { SSN: 110555444, name: 'James Bond' }

personObj.name = 'Steve Smith'; // OK

personObj.SSN = '333666888'; // Compiler Error

In the above example, the SSN property is read only. We define the personObj object of type Citizen and assign values to the two interface properties. Next, we try to change the values assigned to both the properties-name and SSN. The TypeScript compiler will show an error when we try to change the read only SSN property.

Extending Interfaces

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

Example: Extend Interface

 Copy

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.

Example: Interface Implementation

 Copy

interface IEmployee {

empCode: number;

name: string;

getSalary:(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.