Typescript essentials. Interfaces

Interfaces in Typescript determine the contract of interaction between different objects. When we use interface, it means that one object must implement this interface, create methods and properties, which were declared in there and another object will use this interface. The interface in Typescript can be compared for example with 3.5 audio interface. One object, which is for example headphones or maybe music speakers, implements this interface. And another, which can be mobile phone or notebook can accept an object with this interface. And mobile phone and headphones can interact with one another because they both follow the same standard (interface). In Typescript, interfaces work the same way – one classes implement interfaces and others use them.

To create an interface we use the keyword "interface" with the name of created interface. Then inside the interface we describe properties and methods that we want to be implemented in other classes. When interface in defined, we can make some class to follow this interface (fig. 1). For this, we use the keyword "implements".

```
1  interface Person {
2      name: string;
3      wolk: () => void;
4      speak: (string) => string;
5  }
6
7  class Man implements Person {
8      constructor(public name: string) {}
9
10     wolk() {
11         console.log(`Man ${this.name} wolks`);
12     }
13
14     speak(phrase: string) {
15         return `Man ${this.name} says ${phrase}`;
16     }
17 }
```

```
18
19  let man = new Man('John');
20  man.wolk();
21  console.log(man.speak('Hello'));
```

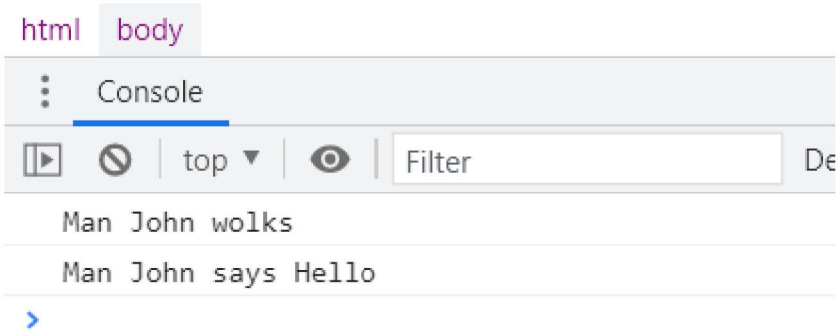

Figure 1 – Defining and implementing the interface

Interfaces in Typescript can be used not only for classes, but also for objects and function parameters. Let's start exactly from this area. In the following example we define a function with a parameter. But we want to be sure that this parameter has property "name". To ensure availability of such a property we can use object literal while defining a function signature. So it will be the same thing as an interface concept that has been given previously.

```
1   function hello(person: {name: string}) {
2       console.log("Hello " + person.name);
3   }
4
5   let person1 = {
6       name: 'Piter'
7   }
8
9   let person2 = {
10      name: 'John',
11      lastname: 'Smith',
12      age: 23
13  }
14
15  hello(person1);
16  hello(person2);
```

Figure 2 – Function with parameter given in form of literal

As we can see, object can contain other methods and properties, besides "name". But if we pass the object to our "hello" function, it must have "name" property (fig. 3) and it has to be "string" data type (fig. 4).

```
}
hello(p   Argument of type '{ type: string; gender: string; }' is not assignable to
          parameter of type '{ name: string; }'.
hello(p     Object literal may only specify known properties, and 'type' does not exist in
          type '{ name: string; }'. ts(2345)
          View Problem (Alt+F8)   No quick fixes available
hello({type: 'human', gender:'male'});
```

Figure 3 – Object with no "name" property

```
hello(p   Type 'number' is not assignable to type 'string'. ts(2322)
hello(p   parameters.ts(1, 25): The expected type comes from property 'name' which is
          declared here on type '{ name: string; }'
hello({   (property) name: string
          View Problem (Alt+F8)   No quick fixes available
hello({name: 123});
```

Figure 4 – Object with number "name" property

This code (fig. 2) works and can be used without any problems. But parameter restrictions can be the same in several different places. And it is not convenient to define the same literal in different places. Instead, it is better to define new datatype in form of interface. Let's realize previous example with function and "name" property but using interface (fig. 5).

```
21  interface PersonWithName {
22      name: string
23  }
24
25  function helloPerson(person: PersonWithName) {
26      console.log(`Hello ${person.name}`);
27  }
28
29  let person = {name: 'John', lastName: 'Smith'};
30  helloPerson(person);
```

Figure 5 – Interface "Person"

Interfaces can have optional properties. If property is optional, it doesn't need to be presented in object obligatory. Also we can use "readonly" keyword while

defining interface property. It makes possible to set value to such a property only when we create an object and restricts future changing of it.

```
1  interface Student {
2      readonly name: string;
3      groupNumber: string;
4      age?: number;
5  }
6
7  let student1: Student = {name: 'Helga', groupNumber: '108', age: 18};
8  let student2: Student = {name: 'Piter', groupNumber: '108'};
9
10  let student3: Student = {name: 'Frakn', age: 19};
```

```
Property 'groupNumber' is missing in type '{ name: string; age: number; }'
required in type 'Student'. ts(2741)

parameters2.ts(3, 5): 'groupNumber' is declared here.

let student3: Student
```

```
11
12  student1.groupNumber = '208';
13  student1.name = 'Olga';
```

```
Cannot assign to 'name' because it is a read-only property.

(property) Student.name: any
```

Figure 6 – Optional and readonly properties

As was shown, “Student” instance can be initialized with or without “age” property, because it is optional. But at the same time we cannot create an instance without obligatory “groupNumber” property (line 10). Also “groupNumber” property cannot be changed after the object was initialized (line 13) because this property is read only.

Interfaces also can be used to describe not only object type but also a function type too. For example, let’s define “Speakeble” interface that will be defined as function literal with one string parameter and returns string result (fig. 7).

```
1  interface speakable {
2      (string): string;
3  }
4
```

```
let speak: speakable = function(phrase) {
    return `Somebody says ${phrase}`;
}

speak('Hello !');
```

Figure 7 – “Speakable” interface

Then we define a variable with the “Speakable” type and initialize it with the value of the function with the same signature.

But mostly interfaces are used with classes. In this case we implements one or several interfaces in one or several classes. It gives us insurances that class has some certain properties or can do some actions using certain methods. In the next example we define “Speakable” and “Wolkable” interfaces and implements them in three different classes. Then we define arrays of “Speakable” and “Wolkable” and try to execute “speak” and “wolk” methods (fig. 8).

```
interface Speakable {
    speak(string): string;
}

interface Wolkable {
    wolk(): void;
}

class Cat implements Wolkable {
    constructor(public name: string) {}

    wolk() {
        console.log('Cat ' + this.name + ' wolks');
    }
}

class Boy implements Wolkable, Speakable {
    constructor(public name: string) {}

    wolk() {
        console.log('Boy ' + this.name + ' wolks');
    }

```

```
    speak(phrase: string) {
        return `Boy ${this.name} says ${phrase}`;
    }
}

class Speaker implements Speakable {
    speak(phrase: string) {
        return `Speaker says ${phrase}`;
    }
}

const cat = new Cat('Tom');
const boy = new Boy('Jack') ;
const speaker = new Speaker();

let speakebles: Speakable[] = [boy, speaker];
let wolkables: Wolkable[] = [cat, boy];

for(let oneSpeakeble of speakebles) {
    console.log(oneSpeakeble.speak('hello'));
}

for(let oneWolkable of wolkables) {
    oneWolkable.wolk();
}
```

Figure 8 – Implementing interfaces in different classes

And finally let's dive into another interesting type in Typescript, which is union type. This is used if some variable can accept several data types. For example, "groupNumber" variable can be number, but in some cases it also can be a string (fig. 9).

```
let groupNumber: string | number;

groupNumber = 108;
groupNumber = '608m'
```

Figure 9 – Union data type

To determine what exactly type variable has, "typeof" keyword can be used. In the next example we will create "add" function that sums two numbers or concatenate two strings, if one of the parameters is not a number (fig. 10).

```
6  function add(a: string | number, b: string | number): string | number {
7      if (typeof a == 'number' && typeof b == 'number') {
8          return a + b;
9      }
10     return a.toString() + b.toString();
11 }
12
13 console.log(add(1, 2));
14 console.log(add(1, '2'));
```

```
Console
top   Filter
3
12
```

Figure 10 – Add function with “union” parameters data type

Another Typescript particularity is difference in the type of variables and constants. When variable is initialized without directly defining the type, Typescript does it for us (fig. 11).

```
let numb1: number
let numb1 = 10;
```

Figure 11 – Implicitly defining variable data type

But when constant is initialized, it gets datatype of direct value (fig. 12).

```
const pi: 3.1415
const pi = 3.1415;
```

Figure 12 – Implicitly defining constant data type

We can use this particularity combined with union data type to define variable that can accept several certain values (fig. 13).

```
19
20 let myDigit: 1 | 10 | 100;
21 myDigit = 10;
19
20
21
22
23 myDigit = 5;
```

```
Type '5' is not assignable to type '1 | 10 | 100'. ts(2322)
let myDigit: 10 | 1 | 100
View Problem (Alt+F8)   No quick fixes available
```

Figure 13 – Variable that accepts only 1, 10 or 100

While working with union data type we can also use previously defined classes. For example, let's realize a function that will accept "Rat", "Dog" classes and string. This function will determine, what animal was passed to in as an argument (fig. 14).

```
28  function animalType(animal: Rat | Dog | String) {
29      if (animal instanceof Rat) {
30          return 'This is a Rat';
31      } else if(animal instanceof Dog) {
32          return 'This is a Dog';
33      } else {
34          return 'This is a ' + animal;
35      }
36  }
37
38  console.log(animalType(new Rat()));
39  console.log(animalType('Tiger'));
```

Figure 14 – Union data type another example

If some union data type is used in many places, we can define it as a separate data type and use it in different places. So in previous example we can define "animal" type that can be "Rat", "Dog" or some string value (fig. 15).

```
28  type Animal = Rat | Dog | String
29
30  function animalType(animal: Animal) {
31      if (animal instanceof Rat) {
32          return 'This is a Rat';
```

Figure 15 – "Animal" data type defining