Typescript essentials. Inheritance

Now let's have a look at class inheritance concept in Typescript. Inheritance is a mechanism that allows to create classes by extending already existing classes. In Typescript inheritance is realized using keyword "extends". After this, derived class gets properties and methods from the base class.

Next, we create base class "BaseClass" with three properties (with three different property access modifiers) and three methods with the same access modifiers.

```
1   class BaseClass {
2       private privateProp: string;
3       protected protectedProp: string;
4       public publicProp: string;
5
6       constructor() {
7           this.privateProp = 'Private property';
8           this.protectedProp = 'Protected property';
9           this.publicProp = 'Public property';
10          console.log('Message from base constructor');
11      }
13      private privateMethod() {
14          console.log('Private method');
15      }
16
17      protected protectedMethod() {
18          console.log('Protected method');
19      }
20
21      public publicMethod() {
22          console.log('Public method');
23      }
24  }
25
```

Figure 1 – Base class code

When we try to access private or protected methods or properties outside the class, we have an error (fig. 2).

```
26  console.log('Start with base class');
27  let base = new BaseClass();
28  base.publicProp = "new public value";
29  base.protectedProp = "new protected value";
30  base.privateProp = "new private value";
```

Figure 2 – Errors while try to access private or protected methods and properties

Next, we extend this class by creating derived “ChildClass”. Inside it, we will try to access properties and methods of the base class. In addition, one public method of the base class will be overwritten (fig. 3). As we can see in “testProtectedMethod” in derived class, we can access protected methods of the base class. However, “testPrivateMethod” illustrates that it is no access to private methods of the base class from the derived class. Also “getProperties” method shows that we can access public and protected properties of the base class but cannot access private property.

```
32  class ChildClass extends BaseClass {
33
34      constructor() {
35          console.log('Before base constructor');
36          super();
37          console.log('After base constructor');
38          this.publicProp = 'child public property';
39      }
40
41      public testPrivateMethod() {
42          this.privateMethod();
43      }
44
45      public testProtectedMethod() {
46          this.protectedMethod();
47      }
48
```

```
49      public publicMethod() {
50          console.log('Public overwritten method');
51      }
52
53      public publicMethod2() {
54          base.publicMethod();
55      }
56
57      public getProperties() {
58          console.log(this.publicProp);
59          console.log(this.protectedProp);
60          console.log(this.privateProp);
61      }
62  }
```

Figure 3 – Derived class "ChildClass"

As was mentioned, some base methods can be overwritten. It is shown in the "publicMethod" of the "ChildClass". Overwritten base methods also can be accessed from the child class. It illustrates the "publicMethod2" of the "ChildClass". If we use constructor in child class we must to call base class constructor too. It can be done by using "base()" method.

Then we create an instance of "ChildClass", try to set public property and call some of its methods (fig. 4). The result of running the code that was given in fig. 1-4, is given in fig. 5.

```
64  console.log('Start with child class');
65  let child = new ChildClass();
66  child.publicProp = "new child public value";
67  child.publicMethod();
68  child.publicMethod2();
```

Figure 4 – Creating an instance of "ChildClass"

```
Start with base class
Message from base constructor
Start with child class
Before base constructor
Message from base constructor
After base constructor
Public overwritten method
Public method
```

Figure 5 – The result of running previously given code

Further, let's consider another example of class inheritance. There will be the "Shape" class with the "shapeType" property, constructor and "showShape" method that will return string with information about the shape. Then two derived classes will be defined. They will inherit from "Shape" but also have their own behavior. So, let's begin with the "Shape" class (fig. 6).

```
1   class Shape {
2       shapeType: string;
3
4       constructor(type: string) {
5           this.shapeType = type;
6       }
7
8       showShape() {
9           return "This is the " + this.shapeType;
10      }
11  }
```

Figure 6 "Shape" class

First of two derived classes will be the "Rectangle" class. This class beside "shapeType" property that it inherits from the parent "Shape" class will also have "width" and "height" properties. We define its own constructor as well, where these properties will be set. In the constructor the obligatory requirement is fulfilled – is called the constructor of the super class. And finally, we overwrite "showShape" method to display all rectangle information. While doing this, we also use the "showShape" method of the base class (fig. 7).

```
13  class Rectangle extends Shape {
14      height: number;
15      width: number;
16
17      constructor(h: number, w: number) {
18          super('rectangle');
19          this.height = h;
20          this.width = w;
21      }
22
23      showShape() {
24          return `${super.showShape()}. Width: ${this.width}, height: ${this.height}`;
25      }
26  }
```

Figure 7 – Child class "Rectangle"

Second derived class “circle” has one additional property “radius”. And we define it by adding an access modifier to the constructor parameter (as we already did previously). Also, the same way the “showShape” method is overwritten (fig. 8).

```
28  class Circle extends Shape {
29      constructor(public radius: number) {
30          super('circle');
31      }
32
33      showShape() {
34          return `${super.showShape()} with radius ${this.radius}`;
35      }
36  }
```

Figure 8 – Child class “Circle”

Using “Shape”, “Circle” and “Rectangle” classes we can also demonstrate one of the basis of object oriented programming – the polymorphism. The point is that if we have the basic object behavior, we don’t need to know its type exactly to run or access some basis method or property. Let’s demonstrate this by creating an array of shapes, where besides shape will be also circle and rectangle. Because they all inherit from “Shape” class, they all have “showShape” method, but this method will work differently depending on the final object type (fig. 9). In this example, we also demonstrate the Typescript feature that the object does not have to inherit the base type directly. We can simply define an object that will have all public properties and methods of the base type class and that will be enough.

```
38  let shapes: Shape[] = [];
39
40  const shape = new Shape('Simply shape');
41  shapes.push(shape);
42
43  shapes.push(new Circle(20));
44  shapes.push(new Rectangle(5, 10));
45
```

```
46  shapes.push({
47      shapeType: 'some object',
48      showShape: () => {
49          return "Show some object";
50      }
51  });
52
53  for (let oneShape of shapes) {
54      console.log(oneShape.showShape());
55  }
```

Figure 9 – Polymorphism example

The result of executing previously given code is shown in figure 10.

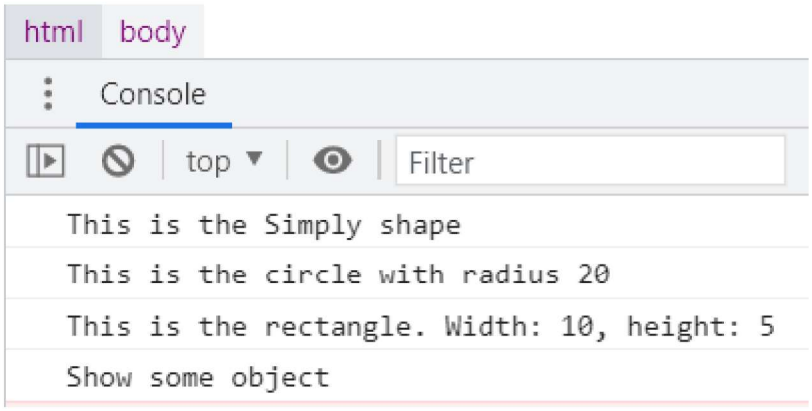

Figure 10 – Executing previously given code

As many other languages, Typescript has an option to mark some class as abstract. If it is done, nobody can create an instance of such a class. It can be useful if the base class doesn't have any sense by it's own. For example, if we don't want to make it possible to create some shape without clarifying what the shape it is, we need to make the "Shape" class abstract. After making "Shape" class abstract in the previous example we will have an error while trying to define an instance of the "Shape" class at the 40th row (fig. 11).

```
1   abstract class Shape {
2       shapeType: string;
3
39
40  const shape = new Shape('Simply shape');
41  shapes.push(shape);
42
```

Figure 11 – Error while trying to create an instance of abstract class

Another particularity of abstract classes is that they can contain abstract methods. It means that we are certain about the fact that future objects must have some functionality, but we don't know its correct behavior yet (at the stage of abstract class creation). So abstract method – is the method that has no implementation, but obligatory needs to be implemented in the derived non-abstract classes. Also abstract methods can be defined only in abstract classes and class with abstract method has to be abstract.

For example, let's add an abstract method "area" that will calculate and return the area of the shape. We don't know already how to calculate it, because each shape will have its own calculation formula. But we want from each shape to be able to do this (fig. 12).

```
1  abstract class Shape {
2      shapeType: string;
3
4      constructor(type: string) {
5          this.shapeType = type;
6      }
7
8      showShape() {
9          return "This is the " + this.shapeType;
10     }
11
12     abstract area(): number;
13 }
```

Figure 12 – Abstract method "area" of the "Shape" class

But after this we will have an error in "Circle" and "Rectangle" classes, because they have not implemented this method yet (fig. 13).

```
10     }
11
12     ab
13 }
14
15 class Rectangle extends Shape {
16     height: number;
17     width: number;
```

Non-abstract class 'Rectangle' does not implement inherited abstract member 'area' from class 'Shape'. ts(2515)

class Rectangle

View Problem (Alt+F8) Quick Fix... (Ctrl+.)

```
25      sh
26          Non-abstract class 'Circle' does not implement inherited abstract member 'area'
27      }   from class 'Shape'. ts(2515)
28  }       class Circle
29          View Problem (Alt+F8)   Quick Fix... (Ctrl+.)
30  class Circle extends Shape {
31      constructor(public radius: number) {
32          super('circle');
33      }
```

Figure 13 – Errors in “Circle” and “Rectangle” classes

So, we need to implement them in these classes (fig. 14).

```
15  class Rectangle extends Shape {
16      height: number;
17      width: number;
18
29      area(): number {
30          return this.height * this.width;
31      }

34  class Circle extends Shape {

43      area(): number {
44          return 3.14 * Math.pow(this.radius, 2);
45      }
```

Figure 14 – Implementing abstract “Area” methods in derived classes

After doing this, everything will be all right and we can test the results by trying to call the “area” method for some “Shape” instances (fig. 15).

```
48  let shapes: Shape[] = [];
49
50  shapes.push(new Circle(20));
51  shapes.push(new Rectangle(5, 10));
52
53  shapes.push({
54      shapeType: 'some object',
55      showShape: () => {
56          return "Show some object";
57      },
58      area: () => 0
59  });
60
61  for (let oneShape of shapes) {
62      console.log(oneShape.showShape() + '. Its area: ' + oneShape.area());
63  }
```

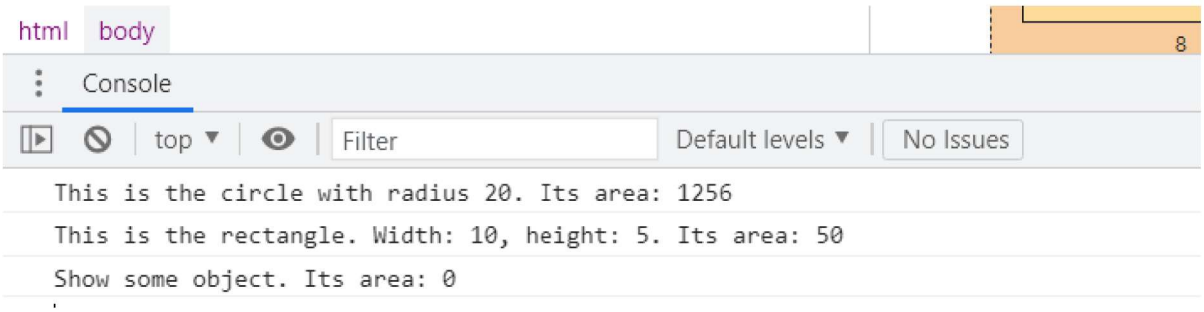

Figure 15 – Calculate areas of different shapes