TypeScript Minified

Amin Meyghani

Contents

1 Introduction

This is a book from the *Minifed* series on TypeScript. It goes through the essentials very fast so that you can get up to speed with TypeScript. The theme of this book is TypeScript and Angular2.

2 Object Orientation

Interfaces and classes are heavily used in Object Oriented Programming. In this chapter we will focus on these topics.

2.1 Interface

- An Interface is defined using the `interface` keyword
- Interfaces are used only during compilation time to check types
- By convention, interface definitions start with an `I`, e.g. : `IPoint`
- Interfaces are used in classical object oriented programming as a design tool
- Interfaces don't contain implementations
- They provide definitions only
- When an object implements an interface, it must adhere to the contract defined by the interface
- An interface defines what properties and methods an object must implement
- If an object implements an interface, it must adhere to the contract. If it doesn't the compiler will let us know.
- Interfaces also define custom types

2.1.1 Basic Interface

Below is an example of an Interface that defines two properties and three methods that implementers should provide implementations for:

```
interface IMyInterface {
  // some properties
  id: number;
  name: string;

  // some methods
  method(): void;
  methodWithReturnVal():number;
  sum(nums: number[]):number;
}
```

Using the interface above we can create an object that adheres to the interface:

```
let myObj: IMyInterface = {
  id: 2,
  name: 'some name',

  method() { console.log('hello'); },
  methodWithReturnVal () { return 2; },
  sum(numbers) {
    return numbers.reduce( (a,b) => { return a + b } );
  }
};
```

Notice that we had to provide values to **all** the properties defined by the Interface, and the implementations for **all** the methods defined by the Interface.

And then of course you can use your object methods to perform operations:

```
let sum = myObj.sum([1,2,3,4,5]); // -> 15
```

2.2 Classes

- Classes are heavily used in classical object oriented programming
- It defines what an object is and what it can do
- A class is defined using the `class` keyword followed by a name
- By convention, the name of the class start with an uppercase letter
- A class can be used to create multiple objects (instances) of the same class
- An object is created from a class using the `new` keyword

- A class can have a `constructor` which is called when an object is made from the class
- Properties of a class are called instance variables and its functions are called the class methods
- Access modifiers can be used to make them public or private
- The instance variables are attached to the instance itself but not the prototype
- Methods however are attached to the prototype object as opposed to the instance itself
- Classes can inherit functionality from other classes, but you should favor composition over inheritance or make sure you know when to use it
- Classes can implement interfaces

Let's make a class definition for a car and incrementally add more things to it.

2.2.1 Adding an Instance Variable

The `Car` class definition can be very simple and can define only a single instance variable that all cars can have:

```
class Car {
  distance: number;
}
```

- `Car` is the name of the class, which also defines the custom type `Car`
- `distance` is a property that tracks the distance that car has traveled
- Distance is of type `number` and only accepts `number` type.

Now that we have the definition for a car, we can create a car from the definition:

```
let myCar:Car = new Car();
myCar.distance = 0;
```

- `myCar:Car` means that `myCar` is of type `Car`
- `new Car()` creates an instance from the `Car` definition.
- `myCar.distance = 0` sets the initial value of the `distance` to 0 for the newly created `car`

2.2.2 Adding a Method

So far our car doesn't have any definitions for any actions. Let's define a `move` method that all the cars can have:

```
class Car {
  distance: number;
  move():void {
    this.distance += 1;
  };
}
```

- `move():void` means that `move` is a method that does not return any value, hence `void`.
- The body of the method is defined in `{ }`
- `this` refers to the instance, therefore `this.distance` points to the `distance` property defined on the car instance.
- Now you can call the `move` method on the car instance to increment the `distance` value by 1:

```
myCar.move();
console.log(myCar.distance) // -> 1
```

2.2.3 Adding a constructor

A `constructor` is a special method that gets called when an instance is created from a class. Let's add a constructor to the `Car` class that initializes the `distance` value to 0. This means that all the cars that are crated from this class, will have their `distance` set to 0 automatically:

```
class Car {
  distance: number;
  constructor () {
    this.distance = 0;
  };
  move():void {
    this.distance += 1;
  };
}
```

- `constructor()` is called automatically when a new car is created
- The body of the constructor is defined in the `{ }`

So now when we create a car, the `distance` property is automatically set to 0.

2.2.4 Using Access Modifiers

If you wanted to tell the compiler that the `distance` variable is private and can only be used by the object itself, you can use the `private` modifier before the name of the property:

```
class Car {
  private distance: number;
  constructor () {
    ...
  };
  ...
}
```

Access modifiers can be used in different places. Check out the access modifiers chapter for more details.

2.2.5 Implementing an Interface

Classes can implement one or multiple interfaces. We can make the `Car` class implement two interfaces:

interfaces

```
interface ICarProps {
  distance: number;
}
```

```
interface ICarMethods {
  move():void;
}
```

Making the `Car` class implement the interfaces:

```
class Car implements ICarProps, ICarMethods {
  distance: number;
  constructor () {
    this.distance = 5;
  };
  move():void {
    this.distance += 1;
  };
}
```

The above example is silly, but it shows the point that a class can implement one or more interfaces. Now if the class does not provide implementations for any of the interfaces, the compiler will complain. For example, if we leave out the `distance` instance variable, the compiler will print out the following error:

> error TS2420: Class 'Car' incorrectly implements interface 'ICarProps'. Property 'distance' is missing in type 'Car'.

3 Decorators

There are different types of decorators:

- class
- property
- attribute
- method

Below we are going to talk about each decorator type.

3.1 Class Decorator

Class decorators are used to decorate classes.

3.2 Property Decorators

Property decorators are used to decorate properties.

4 Angular and TypeScript

This chapter is about using Angular and TypeScript. It goes through some of Angular's source code and points out the way TypeScript is used. It is useful to know where things are and how to make sense of the source code in case you want to dig deeper in the srouce.

4.1 Annotations

- Annotations and decorators are practically the same.
- Annotations are Angular specific and are implemented by TypeScript decorators
- There are different type of decorators. Check out the decorators chapter for more details.
- The most commonly used decorators are the class decorators.
- Angular uses class decorators to define component annotation.
- Below is a simple component annotation using a class decorator:

```
@component({ ... });
class MyComponent {}
```

4.2 Interfaces

- Interfaces are used all over the place
- The most notable ones are the *LifeCycle Hooks*
- Angular defines the interfaces and you have to provide the implementation

- Angular defines the `lifeCyle` interfaces in the `link` folder:

```
export interface OnChanges {
  ngOnChanges(changes: {
    [key: string]: SimpleChange
  });
}

export interface OnInit {
  ngOnInit();
}

export interface OnDestroy {
  ngOnDestroy();
}
```

Below is an example of using the `onInit` hook:

```
import {bootstrap} from 'angular';
import {onClick} from 'angular';
@component({
  selector: 'app'
});
class App implements onClick {}
```

The corresponding html

```
<!DOCTYPE html>
<html>
<head>
  <title>example</title>
</head>
<body>
  <app [prop]='data'></app>
</body>
</html>
```

The corresponding css:

```
.app {
  display: block;
}
```