Class in TypeScript

TypeScript is a superset of JavaScript, so whatever possible to do in JavaScript is also possible in TypeScript. Class is a new feature added from ES6 onward, so earlier in JavaScript the class type functionality was tried using a function with prototype functionality to reuse code. Using class, you can have our code almost close to languages like java, c#, python, etc., where the code can be reused. With the feature of class in TypeScript/JavaScript, it makes the language very powerful.

Defining a Class in TypeScript

Here is a basic class syntax in TypeScript:

```
class nameofclass {
    //define your properties here

   constructor() {
    // initialize your properties here
   }

  //define methods for class
}
```

Example: A working example on Class

```
class Students {
   age : number;
   name : string;
   roll_no : number;

   constructor(age: number, name:string, roll_no: number) {
      this.age = age;
      this.name = name;
      this.roll_no = roll_no;
   }

   getRollNo(): number {
      return this.roll_no;
   }

   getName() : string {
      return this.name;
   }

   getAge() : number {
      return this.age;
   }
}
```

In the above example, you have a class called Students. It has properties age, name, and roll_no.

Constructor in a TypeScript Class

The class Students example we have defined above, it has a constructor as shown below :

```
constructor(age: number, name:string, roll_no: number) {
        this.age = age;
        this.name = name;
        this.roll_no = roll_no;
    }
```

The constructor method has params age, name, and roll_no. The constructor will take care of initializing the properties when the class is called. The properties are accessed using **this** keyword. Example this.age to access age property, this.roll_no to access roll_no, etc. You can also have a default constructor, as shown below:

```
constructor () {}
```

Methods inside a TypeScript Class

The class Students example there are methods defined for example getRollNo(), getName(), getAge() which are used to give details of properties roll_no, name and age.

```
getRollNo(): number {
        return this.roll_no;
}

getName() : string {
                return this.name;
}

getAge() : number {
                return this.age;
}
```

Creating Instance of Class in TypeScript

Example:

In TypeScript to create an instance of a class you need to use the new operator. When we create an instance of a class using new operator we get the object which can access the properties and methods of the class as shown below:

```
            let student_details = new Students(15, "Harry John", 33);
            student_details.getAge(); // 15
            student_details.getName(); // Harry John
```

Compiling TypeScript Class to JavaScript

You can use tsc command as shown below to compile to Javascript.

```
                Command: tsc  Students.ts
```

The output of Javascript code on compilation is as shown below:

```
var Students = /** @class */ (function () {
    function Students(age, name, roll_no) {
        this.age = age;
```

```
        this.name = name;
        this.roll_no = roll_no;
    }
    Students.prototype.getRollNo = function () {
        return this.roll_no;
    };
    Students.prototype.getName = function () {
        return this.name;
    };
    Students.prototype.getAge = function () {
        return this.age;
    };
    return Students;
}());
```

In Javascript, the Class is converted into a self-invoked function.

Class Inheritance

Classes can be inherited using the **extend** keyword in TypeScript.

Class Inheritance Syntax:

```
class A {
     //define your properties here

    constructor() {
     // initialize your properties here
    }
   //define methods for class
}

class B extends A {
 //define your properties here

    constructor() {
     // initialize your properties here
    }
    //define methods for class
}
```

class B will be able to share **class A** methods and properties.

Here is a working example of a class using Inheritance

```
class Person {
    name: string;
    age: number;

    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }

    getName(): string {
        return this.name;
    }
```

```
    getAge(): number {
        return this.age;
    }
}

class Student extends Person {
    tmarks: number;
    getMarks(): number {
        return this.tmarks;
    }

    setMarks(tmarks) {
        this.tmarks = tmarks;
    }
}

let _std1 = new Student('Sheena', 24);
_std1.getAge(); // output is 24
_std1.setMarks(500);
_std1.getMarks(); // output is 500
```

You have two classes, Person and Student. Student class extends Person, and the object created on Student is able to access its own methods and properties as well as the class it has extended.

Now let us add some more changes to the above class.

Example:

```
class Person {
    name: string;
    age: number;

    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }

    getName(): string {
        return this.name;
    }

    getAge(): number {
        return this.age;
    }
}

class Student extends Person {
    tmarks: number;
    constructor(name: string, age: number, tmarks: number) {
        super(name, age);
    }
    getMarks(): number {
        return this.tmarks;
    }

    setMarks(tmarks) {
        this.tmarks = tmarks;
```

```
    }
}

let _std1 = new Student('Sheena', 24, 500);
_std1.getAge(); // output is 24
_std1.getMarks(); // output is 500
```

The changes that you have added in comparison to the previous example is there is a constructor defined in class Student. The constructor has to take the same params as the base class and add any additional params of its own if any.
In TypeScript you need to call super will all the params as the bases params in it. This has to be the first thing to be done inside the constructor. The super will execute the constructor of the extended class.

Method overloading in TypeScript

```
class Greeter {
    greet(name: string): string;
    greet(names: string[]): string;
    greet(nameOrNames: string | string[]): string {
        if (typeof nameOrNames === "string") {
            return `Hello, ${nameOrNames}!`;
        } else {
            return `Hello, ${nameOrNames.join(" and ")}!`;
        }
    }
}

const greeter = new Greeter();
console.log(greeter.greet("Alice")); // Output: Hello, Alice!
console.log(greeter.greet(["Bob", "Charlie"])); // Output: Hello, Bob and Charlie!
```

Key Points about TypeScript Overloading:

- **Compile-Time Only:** The overloading in TypeScript is primarily a compile-time feature. It helps the TypeScript compiler provide better type checking and IntelliSense (autocompletion) for consumers of your code. At runtime, when TypeScript is compiled to JavaScript, there's only one function implementation.
- **Implementation Signature Compatibility:** The crucial part is that the single implementation's signature must be "broad enough" to cover all the overload signatures.
- **Improved API:** Method overloading makes your API more flexible and user-friendly by allowing a single method name to handle different input scenarios, without requiring users to remember separate function names for each variation.

While it's not "true" polymorphism in the sense of having entirely separate implementations for each overload, it's a powerful feature for enhancing type safety and developer experience in TypeScript.

Method Override

When a class extends another class, it can replace the members of the parent class with the same name.

Newer versions of TypeScript allow explicitly marking this with the `override` keyword.

Example

```
interface Shape {
  getArea: () => number;
}

class Rectangle implements Shape {
  // using protected for these members allows access from classes that extend from this class, such as Square


  public constructor(protected readonly width: number, protected readonly height: number) {}

  public getArea(): number {
    return this.width * this.height;
  }

  public toString(): string {
    return `Rectangle[width=${this.width}, height=${this.height}]`;
  }
}

class Square extends Rectangle {


  public constructor(width: number) {
    super(width, width);
  }

  // this toString replaces the toString from Rectangle
  public override toString(): string {
    return `Square[width=${this.width}]`;
  }
}

const obj1 = new Square(15);
console.log("Area of square: "+obj1.getArea());
console.log(obj1.toString());
```

By default the `override` keyword is optional when overriding a method, and only helps to prevent accidentally overriding a method that does not exist. Use the setting `noImplicitOverride` to force it to be used when overriding.

Access Modifiers in TypeScript

TypeScript supports public, private, and protected access modifiers to your methods and properties. By default, if access modifiers are not given the method or property is considered as public, and they will be easily accessible from the object of the class.

In case of private access modifiers, they are not available to be accessed from the object of the class and meant to be used inside the class only. They are not available for the inherited class.

In case of protected access modifiers, they are meant to be used inside the class and the inherited class and will not be accessible from the object of the class.

Example:

```
class Person {
    protected name: string;
    protected age: number;
    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }
    private getName(): string {
        return this.name;
    }
    getDetails(): string {
        return "Name is "+ this.getName();
    }
}

class Student extends Person {
    tmarks: number;
    constructor(name: string, age: number, tmarks: number) {
        super(name, age);
        this.tmarks = tmarks;
    }
    getMarks(): number {
        return this.tmarks;
    }

    getFullName(): string {
        return this.name;
    }

    setMarks(tmarks) {
        this.tmarks = tmarks;
    }
}

let _std1 = new Student('Sheena', 24, 500);
_std1.getMarks(); // output is 500
_std1.getFullName(); // output is Sheena
_std1.getDetails(); // output is Name is Sheena
```

- **Private:** properties or methods cannot be accessed by the object of the class and also the derived class, they are meant to be used internally inside the class.
- **Protected:** properties and methods also cannot be accessed by the object created. They are accessible from inside the class and available to the class extending it.
- **Public:** properties and methods are declared without any keyword. They are easily accessed using the object of the class from outside.

Abstract Classes

Classes can be written in a way that allows them to be used as a base class for other classes without having to implement all the members. This is done by using the `abstract` keyword. Members that are left unimplemented also use the `abstract` keyword.

Example

```
abstract class Polygon {
  public abstract getArea(): number;

  public toString(): string {
    return `Polygon[area=${this.getArea()}]`;
  }
}

class Rectangle extends Polygon {
  public constructor(protected readonly width:
number, protected readonly height: number) {
    super();
  }

  public getArea(): number {
    return this.width * this.height;
  }
}
```

Abstract classes cannot be directly instantiated, as they do not have all their members implemented.

Interface in TypeScript

One of the core features of TypeScript is interfaces. The interface is a set of a rule defined which needs to be implemented by the entity using it. The entity can be a class, function, or variable. An interface can be made up of properties and methods. You can define properties as optional using “?” syntax for that property or method. The interface adds a strong type check for any function, variable, or class implementing the interface.

Syntax of an Interface in TypeScript

```
interface Dimension {
    width: string;
    height: string;
}
```

You have defined an interface named as Dimension which has properties width and height, and both have type as a string.
Now this interface can be implemented by a variable, a function, or a class. Here is the example of variable implementing the interface Dimension.

Example:

```
interface Dimension {
    width: string;
    height: string;
}

let _imagedim: Dimension = {
    width: "100px",
    height: "200px"
};
```

The signature of the interface Dimension has width and height, and both are mandatory. In case while implementing the interface, any of the property is missed, or the type is changed, it will give a compile time error while compiling the code to javascript.

The above code, when compiled to javascript, looks as follows:

```
var _imagedim = {
    width: "100px",
    height: "200px"
};
```

Let us now see how to use an interface with a function.

Using Interface on a function as a return type

Example:

```
interface Dimension {
    width: string;
    height: string;
}

function getDimension() : Dimension {
    let width = "300px";
    let height = "250px";
    return {
        width: width,
        height: height
    }
}
```

In the above example, the interface Dimension is implemented on the function getDimension() as the return type. The return type of getDimension() has to match with the properties and type mentioned for Interface Dimension.

The compiled code to Javascript will be as follows:

```
function getDimension() {
    var width = "300px";
    var height = "250px";
    return {
        width: width,
        height: height
    };
}
```

During compilation, if the return type does not match with the interface, it will throw an error.

Interface as function parameter

```
interface Dimension {
    width: string;
    height: string;
}

function getDimension(dim: Dimension) : string {
    let finaldim  = dim.width +"-"+ dim.height;
    return finaldim;
}

getDimension({width:"300px", height:"250px"}); // will get "300px-250px"
```

So above example, you have used Interface Dimension as a parameter to the function getDimension(). When you call the function, you need to make sure the parameter passed to it matches to the Interface rule defined.
The compiled code to Javascript will be as follows:

```
function getDimension(dim) {
    var finaldim = dim.width + "-" + dim.height;
    return finaldim;
}
getDimension({ width: "300px", height: "250px" });
```

Class implementing Interface

To make use of interface with a class, you need to use the keyword **implements**.
Syntax for Class implementing an interface:

```
class NameofClass implements InterfaceName {
}
```

Following example shows working of interface with class.

```
interface Dimension {
    width : string,
    height: string,
    getWidth(): string;
}

class Shapes implements Dimension {
```

```
    width: string;
    height: string;
    constructor (width:string, height:string) {
        this.width = width;
        this.height = height;
    }

    getWidth() {
        return this.width;
    }
}
```

In the above example, you have defined interface Dimension with properties width and height both of type string and a method called getWidth() which has return value as a string. The compiled code to Javascript will be as follows:

```
var Shapes = /** @class */ (function () {
    function Shapes(width, height) {
        this.width = width;
        this.height = height;
    }
    Shapes.prototype.getWidth = function () {
        return this.width;
    };
    return Shapes;
}());
```

Functions in TypeScript

Functions are set of instructions performed to carry out a task. In Javascript, most of the code is written in the form of functions and plays a major role. In TypeScript, you have class, interfaces, modules, namespaces available, but still, functions play an important role. The difference between the function in JavaScript and TypeScript function is the return type available with TypeScript function.

JavaScript function:

```
function add (a1, b1) {
  return a1+b1;
}
```

TypeScript function:

```
function  add(a1 : number, b1: number) : number {
    return a1 + b1;
}
```

In the above functions, the name of the function is added, the params are **a1**, and **b1** both have a type as a number, and the return type is also a number. If you happen to pass a string to the function, it will throw a compile-time error while compiling it to JavaScript.

Making call to the function: add

```
let  x = add(5, 10) ;  // will return 15
let  b = add(5); // will throw an error : error TS2554: Expected 2 arguments, but got 1.
let c = add(3,4,5); // will throw an error : error TS2554: Expected 2 arguments, but got 3.
let t = add("Harry", "John");// will throw an error :  error TS2345: Argument of type '"Harry"' is not assignable to parameter of type 'number'.
```

The params **a1** and **b1** are mandatory parameters and will throw an error if not received in that manner. Also, the param type and return type is very important and cannot change once defined.

Optional parameters to a function

In javascript, all the parameters to the functions are optional and considered as undefined if not passed. But same is not the case with TypeScript, once you define the params you need to send them too, but in case you want to keep any param optional, you can do so by using? against the param name as shown below:

```
function getName(firstname: string, lastname?: string): string {
   return firstname + lastname;
}

let a = getName("John"); // will return Johnundefined.
let b = getName("John", "Harry"); // will return JohnHarry
let c = getName("John", "H", "Harry"); // error TS2554: Expected 1-2 arguments, but got 3.
```

Please note that the optional params are to be defined in a function at the last only, you cannot have the first param as optional and second param as mandatory.When you call the function with one param compiler will throw an error. So it is necessary to keep the optional params at the end.

Assign Default Values to Params

You can assign default values to params as shown below:

```
function getName(firstname: string, lastname = "Harry"): string {
   return firstname + lastname;
}

let a = getName("John"); // will return JohnHarry
let b = getName("John", "H"); // will return JohnH
```

Similar to optional params, here too default initialized params has to be kept at the end in a function.

Rest Parameters

You have seen how TypeScript handles mandatory params, optional params, and the default value initialized params. Now, will take a look at rest params. Rest params are a group of optional params defined together, and they are defined using three **dots (...)** followed by the name of the param, which is an array.

Syntax for Rest params:

```
function testFunc(a: string, ...arr: string[]) :string {
   return a + arr.join("");
}
```

As shown above, the rest params are defined using (...param-name); the rest param is an array prefixed by three dots. The array will have all the params passed to it. You can call the function, as shown in the example below:

Example:

```
let a = testFunc("Monday", "Tuesday", "Wednesday", "Thursday"); // will get output as
MondayTuesdayWednesdayThursday
```

Arrow Functions

An arrow function is one of the important features released in ES6, and it is available in TypeScript too. Arrow function syntax has a fat arrow in it due to which the function is called an arrow function.

Arrow function Syntax:

```
var nameoffunction = (params) => {
 // code here
}
```

What is the use of Arrow Function?

Let's take a look at the example to understand the use case of Arrow function:

Example:

```
var ScoreCard = function () {
   this.score = 0;

   this.getScore = function () {
      setTimeout(function () {
         console.log(this.score);    // gives undefined.
      }, 1000);
   }
}

var a = new ScoreCard();
a.getScore();
```

You have created an anonymous function which has a property this. Score initialize to 0 and a method getScore which internally has a setTimeout, and in 1 second it consoles this.score. The consoled value gives undefined though you have this.score defined and initialized. The issue here is with th**is** keyword. The function inside setTimeout has its own this, and it tries to refer the score internally, and since it is not defined, it gives undefined.

The same can be taken care using Arrow function as shown below:

```
var ScoreCard = function () {
   this.score = 0;

   this.getScore = function () {
```

```
        setTimeout(()=>{
            console.log(this.score);   // you get  0
        }, 1000);
    }
}

var a = new ScoreCard();
a.getScore();
```

You have changed the function inside setTimeout to an arrow function as shown below:

```
setTimeout(()=>{
        console.log(this.score);   // you get  0
    }, 1000);
```

An arrow function does not have its own **this** defined and it shares its parent **this**, so variables declared outside are easily accessible using this inside an arrow function. They are useful because of the shorter syntax as well as for callbacks, event handlers, inside timing functions, etc.

TypeScript Enums

TypeScript Enum is an object which has a collection of related values stored together. Enums are defined using keyword enum.

Syntax:

```
enum NameofEnum {
   value1,
   value2,
    ..
}
```

Example:

```
enum Directions {
North,
South,
East,
West
}
```

In the above example, you have defined an enum called Directions. The value given is North, South, East, West. The values are numbered from 0 for the first value in the enum and subsequently increments by 1 for the next value.

Declare An Enum with a numeric value

By default, if an enum is not given any value, it considers it a number starting from 0. Following example shows an enum with a numeric value.

```
enum Directions {
North = 0,
South = 1,
East =2,
```

```
West =3
}
```

You may also assign a start value to the enum and the next enum values will get the incremented values. For example:

```
enum Directions {
North = 5,
South, // will be 6
East, // 7
West // 8
}
```

Now the enum value North starts with 5, so South will get value as 6, East = 7 and West = 8. You may also assign values of your choice instead of taking the default ones. For Example:

```
enum Directions {
North = 5,
South = 4,
East = 6,
West = 8
}
```

Accessing an Enum

Following example shows how to make use of Enum in your code:

```
enum Directions {
   North,
   South,
   East,
   West
}

console.log(Directions.North); // output is  0
console.log(Directions["North"]); // output is 0
console.log(Directions[0]); //output is North
```

Note: If the enum members (North, South, etc., in the above) are not assigned with any value, they automatically assume value starting with 0 for the first member, increasing sequentially thereon.

The compiled code to javascript is as follows:

```
var Directions;
(function (Directions) {
   Directions[Directions["North"] = 0] = "North";
   Directions[Directions["South"] = 1] = "South";
   Directions[Directions["East"] = 2] = "East";
   Directions[Directions["West"] = 3] = "West";
})(Directions || (Directions = {}));
console.log(Directions.North);
console.log(Directions["North"]);
console.log(Directions[0]);
```

Since JavaScript does not support enums, it converts the enum into a self-invoked functions as shown above.

Declare An Enum with a string value

You can assign string values of your choice, as shown in the example below:

Example:

```
enum Directions {
   North = "N",
   South = "S",
   East = "E",
   West = "W"
}

console.log(Directions.North); // output is N
console.log(Directions["North"]); // output is N
console.log(Directions[0]); // This will throw an error, as index 0 value is not available
//For the above, only available options are (Directions.North) and Directions["North"]
```

If you have the below enum,

```
enum Directions {
North = 5,
South = 4,
East = 6,
West = 8
}
```

You can access 'South' as below:

```
   console.log(Directions[4]);
```