upGrad

Introduction to TypeScript

Course: Introduction to TypeScript

Lecture On: Classes and Interfaces in TypeScript

Instructor: Aquib Ajani

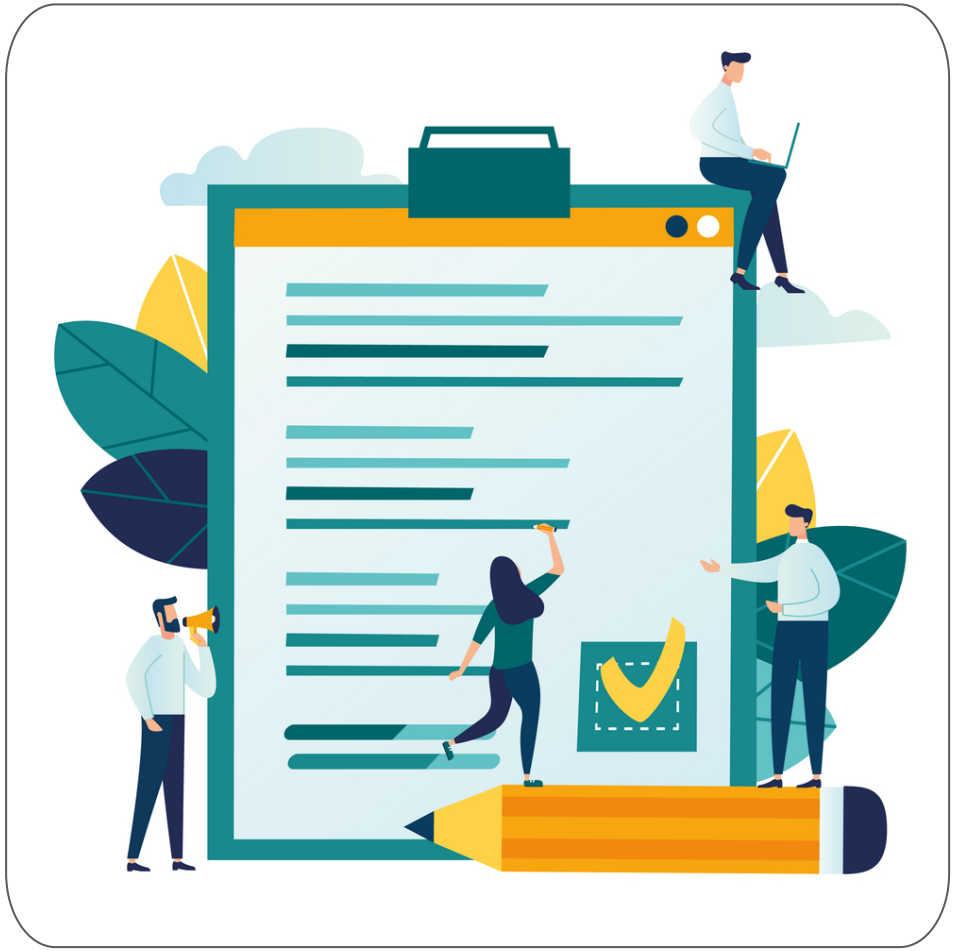
upGrad

COURSE ROAD MAP

- Introduction to TypeScript (TS)
- Operators, conditions and loops in TS
- Functions in TS
- Hands-on coding session - I
- **Classes and interfaces in TS**
- Type manipulation in TS
- Modules in TS
- Hands-on coding session - II

TODAY'S AGENDA

- Introduction to Classes in TS
- Introduction to Interfaces in TS
- Difference between Classes and Interfaces in TS



Classes and Interfaces in TS

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

- Classes are basic buildings blocks used to create objects, or in simple words, a particular data structure that holds its own data, providing initial values for member variables and implementation of member functions.
- A class is denoted by ‘class’ keyword.
- Classes encapsulates the data for the object.
- Objects are instances of a class.
- Objects can correspond to real-world objects or an abstract entity.

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

- Defining a class in TS:
  - You define class using the ‘class’ keyword itself in TS and the naming rules are same as that of variables.
- Class definition includes three main components:
- **Fields** - Represents the properties of the objects of class and can be accessed in the class only
- **Constructors** - Helps allocate memory for the object of the class and initialise the values for the fields. A constructor is a function that can be parameterised
- **Functions** - Also referred to as methods, as do certain operations
- To create an instance of the class, you use **new** keyword.
- For example:

```
Var newInstance = new ClassName();
```

- To access fields:

```
newInstance.fieldName;
```

- To call functions:

```
newInstance.functionCall();
```

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

strictPropertyInitialization

- **As the name suggests, if it is set to True, each property needs to be initialised either by itself or in the constructor.**

Example:

```
class UserAccount {
  name: string;
  accountType = "user";

  email: string;
//Property 'email' has no initializer and is not definitely assigned in the constructor.

  address: string | undefined;

  constructor(name: string) {
    this.name = name;
    // Note that this.email is not set
  }
}
```

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

Readonly modifier:

- If the property is marked as read only, it can be only initialised at the declaration or in the constructor but not outside the class. The property can be accessed outside the class but its value cannot be changed.

Example:

```
class Employee {
    readonly empCode: number;
    empName: string;

    constructor(code: number, name: string)     {
        this.empCode = code;
        this.empName = name;
    }
}
let emp = new Employee(10, "John");
emp.empCode = 20; //Compiler Error
emp.empName = 'Bill';
```

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

Super Keyword and its call

- To use ‘this’ in the derived class, you need to call super() just like that in JavaScript. In TS, it will throw an error for not calling super().

Example:

```
class Base {
  k = 4;
}

class Derived extends Base {
  constructor() {
    // Prints a wrong value in ES5; throws exception in ES6
    console.log(this.k);
'super' must be called before accessing 'this' in the constructor of a derived class.
    super();
  }
}
```

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

Methods in TS:

- Methods have nothing new is TS except standard type annotations. Function property in a class is known as methods.

Example:

```
class Demo {
  x = 10;
  y = 10;

  scaleBy(n: number): void {
    this.x *= n;
    this.y *= n;
  }
}
```

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

Getter/Setter in TS:

- These are known as accessors.

Example:

```
class C {
  _length = 0;
  get length() {
    return this._length;
  }
  set length(value) {
    this._length = value;
  }
}
```

Some important inference rules for accessors in TS are as follows:

- If get exists but no set, the property is automatically readonly
- If the type of the setter parameter is not specified, it is inferred from the return type of the getter
- Getters and setters must have the same Member Visibility

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

Type-only field declarations in TS:

- When useDefineForClassFields is set true, after the parent class constructor completes class fields are initialised, overwriting any value set by the parent class. When you only want to re-declare a more accurate type for an inherited field, this can create a problem. To handle these cases, you can write declare to indicate to TS that there should be no runtime effect for this field declaration.

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

Example:

```
interface Animal {
  dateOfBirth: any;
}
interface Dog extends Animal {
  breed: any;
}
class AnimalHouse {
  resident: Animal;
  constructor(animal: Animal) {
    this.resident = animal;
  }
}
class DogHouse extends AnimalHouse {
  // Does not emit JavaScript code,
  // only ensures the types are correct
  declare resident: Dog;
  constructor(dog: Dog) {
    super(dog);
  }
}
```

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

Class visibility:

- Using different access modifiers, one can decide whether the function or the properties are visible outside the class or not.

Public:

- Can be accessed anywhere

```
class Demo {
  public greet() {
    console.log("hi!");
  }
}
const g = new Demo();
g.greet();
```

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

Protected:

- Can be accessed only till the subclass of the class wherein they have been declared

```
class Demo {
  public greet() {
    console.log("Hello, " + this.getName());
  }
  protected getName() {
    return "hi";
  }
}
class SpecialDemo extends Demo {
  public howdy() {
    // OK to access protected member here
    console.log("Howdy, " + this.getName());
  }
}
const g = new SpecialDemo();
g.greet(); // OK
g.getName();
```

- Property 'getName' is protected and only accessible within the class 'Demo' and its subclasses.

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

Private:

- Can be accessed only in the class

```
class Base {
  private x = 0;
}
const b = new Base();
// Can't access from outside the class
console.log(b.x);
```

- Property 'x' is private and only accessible within the class 'Base'.

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

Static members:

- Static members are not associated with any instance of the class but can be accessed directly through the object of the constructor itself and can use same access modifiers, like public, private and protected, and can be inherited too.

```
class MyClass {
  static x = 0;
  static printX() {
    console.log(MyClass.x);
  }
}
console.log(MyClass.x);
MyClass.printX();
```

- There is no concept of static class in TS. Instead, you have a static block.

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

Static blocks:

- Static helps write statements with their own scope that are capable of accessing private fields within the containing class.

```
class Foo {
    static #count = 0;

    get count() {
        return Foo.#count;
    }

    static {
        try {
            const lastInstances = loadLastInstances();
            Foo.#count += lastInstances.length;
        }
        catch {}
    }
}
```

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

This at runtime in Classes:

- TS does not change the runtime behaviour of JavaScript, but handling ‘this’ in TS is little different.

Example:

```
class Demo {
  name = "Demo";
  getName() {
    return this.name;
  }
}
const c = new Demo();
const obj = {
  name: "obj",
  getName: c.getName,
};

// Prints "obj", not "Demo"
console.log(obj.getName());
```

- To conclude: ‘this’ inside the function totally depends on from where it is called. Here, in the above example, it is called through the obj reference, so the value of ‘this’ was obj and not class instance.
  To prevent this, you have arrow functions for the same to mitigate this.

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

This parameter:

- In method, the initial parameter names have a totally different meaning in TS

Example:

```
class MyClass {
  name = "MyClass";
  getName(this: MyClass) {
    return this.name;
  }
}
const c = new MyClass();
// OK
c.getName();
```

This method makes the opposite trade-offs of the arrow function approach:

- JavaScript callers might still use the class method incorrectly without realising it.
- Only one function per class definition gets allocated, rather than one per class instance.
- Base method definitions can still be called via super.

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

'This' types:

- 'this' type is a special type that dynamically refers to the type of the current class.

```
class Box {
  contents: string = "";
  set(value: string) {


(method) Box.set(value: string): this
    this.contents = value;
    return this;
  }
}
```

- Hence, the inferred return type of set function is 'this' rather than Box.

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

Parameter properties:

- You can convert a constructor parameter to a class property with same name and value. These are called parameter properties and are created by appending the access modifiers to the parameter.

Example:

```
class Demo {
  constructor(
    public readonly x: number,
    protected y: number,
    private z: number
  ) {
    // No body necessary
  }
}
const a = new Demo(1, 2, 3);
console.log(a.x);
(roperty) Demo.x: number
console.log(a.z);
```

- Property 'z' is private and only accessible within class 'Params'.

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

Class expressions:

- These are similar to function expressions and are similar to class declarations, with the difference that they do not have a name and are bound to whatever identifier they are ended up to.

Example:

```
const someDemoClass = class<Type> {
  content: Type;
  constructor(value: Type) {
    this.content = value;
  }
};

const m = new someDemoClass("Hello, world");
```

CLASSES AND INTERFACES IN TS

Introduction to Classes in TS

Abstract classes and members:

- Abstract method or abstract fields have no direct implementation and exist inside the abstract class only, which cannot be directly instantiated. They work as a base class for subclasses and this base class and all its members need to be implemented.

Example:

```
abstract class Base {
  abstract getName(): string;

  printName() {
    console.log("Hello, " + this.getName());
  }
}
class Derived extends Base {
  getName() {
    return "world";
  }
}

const d = new Derived();
d.printName();
```

CLASSES AND INTERFACES IN TS

Introduction to Interfaces in TS

Interfaces:

- Interfaces are of abstract type and help compiler understand which property and its data type can a given object have using type inferring capability of TS. Interfaces give specification of an entity.

Example (declaring interface):

```
interface TataModelS { // interface keyword
    length: number; // properties with its type
    width: number;
    readonly wheelbase: number; //readonly property
    seatingCapacity: number;
    getTyrePressure?: () => number; // ? Here signifies the optional
property
    getRemCharging: () => number; // function property & return type
}
```

CLASSES AND INTERFACES IN TS

Introduction to Interfaces in TS

How to use interfaces:

```
function buildTataModelS (tataObj: TataModelS) {
}
buildTataModelS({
    length: 196,
    width: 86,
    wheelbase: 116,
    seatingCapacity: 4,
    getTyrePressure : function () {
        let tyrePressure = 20 // Evaluated after doing a few complex computations!
        return tyrePressure
    },
    getRemCharging: function () {
        let remCharging = 20 // Evaluated after doing a few complex computations!
        return remCharging
    }
})
```

CLASSES AND INTERFACES IN TS

Introduction to Interfaces in TS

Union type and interfaces:

- TS provides a very important feature that lets you combine multiple interfaces in a single union type. A function can be called based on the union of the type of interfaces.

CLASSES AND INTERFACES IN TS

Introduction to Interfaces in TS

Example:

```
interface Dog {
      type: 'dog';
      name: string;
      barkVolumeDb: number;
}
interface Cat {
            type: 'cat';
            name: string;
            coatType: string;
}
type Pet = Dog | Cat;
const logPetDetails = (pet: Pet): void => {
            switch (pet.type) {
                        case 'dog': {
                                    console.log(`${pet.name}'s bark volume is ${pet.barkVolumeDb} db.`);
                                    break;
                        }
                        case 'cat': {
                                    console.log(`${pet.name}'s coat type is ${pet.coatType}.`);
                                    break;
                        }
            }
}
```

CLASSES AND INTERFACES IN TS

Introduction to Interfaces in TS

Arrays and interfaces:

- Using arrays in interfaces is very useful when dealing with complex data.

Example:

```
interface iPlanet {
  [index: string]: iPlaPo;
}
interface iPlaPo {
  name: string;
  position: number;
}
let planets = new Array<iPlanet>();

planets["earth"] = { name: "earth", position: 3 };
planets["mars"] = { name: "mars", position: 4 };

console.log(planets);
```

/* output

[

earth: { name: 'earth', position: 3 },

mars: { name: 'mars', position: 4 }

CLASSES AND INTERFACES IN TS

Introduction to Interfaces in TS

Interface inheritance:

- You have learnt about class inheritance. Interface inheritance is similar to that. It basically states that one interface can be extended by other to utilise the productivity of multiple interfaces at once.

Example:

```
interface Pet {
  name: string;
}
interface Dog extends Pet {
  barkVolumeDb: number;
}
interface Cat extends Pet {
  coatType: string;
}
const stewie = {
  type: 'dog',
  name: 'Stewie',
  barkVolumeDb: 77,
} as Dog;
const reginald = {
  type: 'cat',
  name: 'Reginald',
  coatType: 'tabby',
} as Cat;
```

CLASSES AND INTERFACES IN TS

Introduction to Interfaces in TS

Optional property:

- In some cases, it would not be necessary to have all the properties as defined in the interface. These are called optional properties and are represented in the interface by '?'.

Readonly property:

- Readonly properties once initialised cannot be changed as the word suggests. They will become readonly afterwards.

Example:

```
interface TataModelS {
    readonly length: number;
    readonly width: number;
    readonly wheelbase: number;
    readonly seatingCapacity: number;
    getTyrePressure?: () => number;
    getRemCharging: () => number;
}
```

CLASSES AND INTERFACES IN TS

Difference Between Classes and Interfaces in TS

Introduction	Classes are the fundamental entities used to used to create reusable components. It is a group of objects which have common properties. It can contain properties like fields, methods, construction, etc.	An interface defines a structure which acts as a contract in our application. It contains only the declaration of the methods and fields, but not the implementation.
Usage	It is used for object creation, encapsulation for fields, methods.	It is used to create a structure for an entity.
Keyword	We can create a class by using the class keyword.	We can create an interface by using the interface keyword.
Completion	A class cannot disappear during the complication of code.	Interface completely disappeared during the complication of code.

CLASSES AND INTERFACES IN TS

Difference Between Classes and Interfaces in TS

Real-Time Usage	A class can be instantiated to create an object.	Implements cannot be instantiated.
Instantiation	A class can be instantiated to create an object.	An interface cannot be instantiated.
Methods	The methods of a class are used to perform a specific action.	The methods in an interface are purely abstract (the only declaration, not have a body).
Access Specifier	The member of a class can be public, protected, or private.	The members of an interface are always public.
Constructor	A class can have a constructor.	An interface cannot have a constructor.
Implement/Extend	A class can extend only one class and can implement any number of the interface.	An interface can extend more than one interfaces but cannot implement any interface.

CODING QUESTION

- Create a developer class with name and position attributes and develop function.
- Also create a developer interface to be implemented in the class.
- Link to GitHub solution

CODING QUESTION

- Create a program using interface inheritance.
- Link to GitHub solution

KEY TAKEAWAYS

- Introduction to Classes in TS
- Introduction to Interfaces in TS
- Difference between Classes and Interfaces in TS

TASKS TO COMPLETE AFTER THE SESSION

MCQs
Coding Questions

upGrad

Thank You!