第3章

面向对象

早期 JavaScript(ES5 前)使用构造函数和原型(prototype)链来实现面向对象编程,没有类的概念,从 ES6 开始引入了类,面向对象编程开始变得方便和直观。而 TypeScript 本身就是一个面向对象的编程语言,完全支持类、接口、继承等语法。通过本章的相关概念阐述和代码示例学习,读者应能理解并掌握 TypeScript 面向对象编程相关概念和技术。

面向对象编程(Object Oriented Programming,OOP),是通过对象的方式把现实世界映射到计算机模型的一种编程方法。相比于面向过程编程(Procedure Oriented Programming,POP),面向对象编程更适合分析和解决复杂项目问题。

简单而言,对象就是现实中的实体,类就是现实中的分类。比如,现在要实现一个员工管理系统。公司里有张珊、李思等员工。员工是一种分类,在面向对象编程中就是类,而张珊和李思这些具体实体在面向对象编程中就是对象。


视频讲解

3.1 类

类是面向对象编程的核心。

类是对象的抽象,是用于创建对象的模板。没有通过类创建的对象,就无法借助映射、模拟问题域中的实体来解决项目问题。

面向对象编程时,通常在项目的问题域中分析现实中的实体,将同类实体的特征、属性、功能、行为等抽象出来,形成类结构。

3.1.1 类结构

在 TypeScript 中,类可以看成由名字、属性、函数组成的一个封装结构体。

注意,在类中的函数习惯上被称作方法(method)。但本质上方法还是函数,只是函数在类中的别称而已。为便于陈述,本书不再区分方法和函数,统一称之为“函数”。

定义类结构的语法如下:

```
class 类名 {
    修饰符 属性名 : 类型
    constructor(参数名 : 类型, ...) {
        构造体
    }
    修饰符 函数名(参数名 : 类型, ...) {
        函数体
    }
}
```

类用关键字 class 声明；建议类名首字母大写；类可以没有属性和函数，也可以有多个；构造函数是用于创建对象的特殊函数，构造函数名用 constructor 表示。

【例 3-1】 定义员工类

```
1.    class Employee {
2.        name : string
3.        salary : number
4.        constructor(name : string, salary : number){
5.            this.name = name
6.            this.salary = salary
7.        }
8.        addSalary(increase : number){
9.            this.salary += increase
10.       }
11.   }
12.   let zhangShan : Employee = new Employee('张珊',5000)
13.   console.log(zhangShan);
14.   zhangShan.addSalary(1000)
15.   console.log(zhangShan.salary);
```

第 1～11 行，用 class 关键字定义了员工类 Employee，类的结构体则封装在一对花括号{}中。

第 2～3 行，定义了类的两个属性：string 类型的 name 和 number 类型的 salary。

第 4～7 行，constructor()为构造函数，是用于创建对象的特殊函数。如 12 行 new Employee('张珊',5000)就是用该构造函数创建员工类 Employee 的对象 zhangShan(变量名首字母小写)。注意，在第 5、6 行中，this.name 和 this.salary 与定义构造函数时传入的参数 name 和 salary 不同，“this.变量名”代表当前对象的属性。第 5、6 行的作用是通过传入的 name 和 salary 参数值初始化对象的两个属性值。关于构造函数的更多细节，可参考 3.1.5 节内容。

第 8～10 行，定义 addSalary()函数，其功能是通过传入参数 increase 的值来改变属性 salary 的值。

第 12 行，用关键字 new 加类名，调用构造函数来创建类 Employee 的对象 zhangShan，同时设置对象 zhangShan 的两个属性 name、salary 的值分别为'张珊'和 5000。

第 13 行，用 console.log()函数输出对象 zhangShan 的值。

第 14 行，调用 addSalary()函数，用于改变对象 zhangShan 的 salary 属性值。

第 15 行，用 console.log()函数输出 zhangShan 的 salary 属性值。

执行结果为：

```
Employee { name: '张珊', salary: 5000 }
6000
```

3.1.2　属性

属性在编程中常常被称为字段(field)或成员变量(member variable)。

属性值一般用于表示实体的具体状态或特征。如定义一个员工类，员工有编号、姓名、性别、基本工资等特征值，也具备是否为应届生、是否在职等状态值，这些都可以在员工类中用属性来表示。

1. 属性定义需要初始化

属性若不是可选类型，则定义时应该初始化(定义时直接赋值)，或在构造函数中进行初始化赋值，否则会出现语法错误。

【例 3-2】 未初始化属性值会出现语法错误

```
1.    class Pet {
2.          name : string
3.          constructor(){
4.          }
5.    }
```

第 2 行，未对属性 name 赋值，也没有在第 3～4 行的构造函数中赋值，会造成编译时报错：

```
Property 'name' has no initializer and is not definitely assigned in the constructor.
```

解决以上语法问题有 3 种方式：定义属性时直接赋值、在构造函数中对属性赋值、设置属性为可选类型。

【例 3-3】 未初始化属性值语法错误，解决方式一：定义属性时直接赋值

```
1.    class Pet{
2.          name : string = 'unknown'
3.          constructor(){}
4.    }
```

第 2 行，定义属性 name 时直接为其赋值'unknown'，不再出现语法错误。

【例 3-4】 未初始化属性值语法错误，解决方式二：在构造函数中对属性赋值

```
1.    class Pet{
2.          name : string
3.          constructor(name : string){
```

```
4.              this.name = name
5.          }
6.      }
```

第 4 行，在构造函数中为属性 name 赋予参数值。

【例 3-5】 未初始化属性值语法错误，解决方式三：设置属性为可选类型

```
1.      class Pet{
2.          name ?: string
3.          constructor(){}
4.      }
5.      let pet = new Pet()
6.      console.log(pet)
```

第 2 行，用问号？设置属性为可选类型，这样属性就不需要初始化了。当然，没有初始化相当于该属性不存在。因此，执行第 6 行语句不会输出属性 name，执行结果为：

```
Pet {}
```

2. 静态属性

除了用于描述对象状态和特征的属性外，还有能描述类的静态特征的属性，即静态属性。

普通属性可被称为实例属性或对象属性，必须先实例化对象，方能使用。而静态属性是属于“类”本身的属性，用 static 关键字声明，无须实例化对象，通过类名就可调用。

【例 3-6】 用 static 关键字声明静态属性并通过类名调用

```
1.      class MathTool{
2.          static pi:number = 3.14
3.      }
4.      MathTool.pi = 3.1415
5.      console.log(MathTool.pi)
```

第 2 行，用关键字 static 声明静态属性 pi。

第 4～5 行，用“类名.静态属性”方式调用 MathTool 类的静态属性 pi。

执行结果为：

```
3.1415
```

3.1.3 函数

函数是复用代码的最基本单位。

对 TypeScript 而言，函数可单独定义，也可以定义在类中。与属性类似，定义在类中的函数也分为实例和静态两种。实例函数属于对象，必须先创建对象才能使用；静态函数属于类，用 static 声明，通过类名直接调用。

【例 3-7】　在类中定义实例函数

```
1.    class MathTool {
2.         els : number[] = []
3.         max(...els : number[]) : number {
4.              els.sort()
5.              return els[els.length - 1]
6.         }
7.    }
8.    let mt = new MathTool()
9.    mt.els = [1,4,2,3,2]
10.   let max = mt.max(1,4,2,3,2)
11.   console.log(max)
```

第 3～7 行，定义 MathTool 类的实例函数 max()。

第 8 行，创建 MathTool 类的对象 mt。

第 10 行，用“对象名.实例函数”方式调用对象 mt 的实例函数 max()。

执行结果为：

```
4
```

【例 3-8】　在类中定义静态函数

```
1.    class MathTool{
2.         static max(...els:number[]):number{
3.              els.sort()
4.              return els[els.length - 1]
5.         }
6.    }
7.    let max = MathTool.max(1,4,2,3,2)
8.    console.log(max)
```

第 2 行，用关键字 static 声明静态函数 max()。

第 7 行，用“类名.静态函数”方式调用类 MathTool 的静态函数 max()。

执行结果为：

```
4
```

3.1.4　存储器与访问器

存储器(setter)和访问器(getter)，就是用来获取和设置属性值的特殊函数。

如果外界可以随意访问属性，就会引发安全问题，这就是引入存储器和访问器来间接访问属性的原因。另外，存储器和访问器实际上为函数结构，因此还可在内部实现一些额外的逻辑功能。

访问器用关键字 get 定义，存储器用关键字 set 定义。

【例 3-9】 定义存储器和访问器

```
1.    class Emp {
2.         constructor(){}
3.         private _name : string
4.         get name() : string{                    //访问器
5.             return this._name
6.         }
7.         set name(name : string){                //存储器,不允许有返回类型
8.             this._name = name
9.         }
10.   }
11.   let zs = new Emp()
12.   zs.name = "张珊"                             //实际调用 setter 存储器: set name("张珊")
13.   console.log(zs.name)                         //实际调用 getter 访问器: get name()
```

第 4～6 行,用 get 定义访问器,访问器对属性_name 做了简单返回。

第 7～9 行,用 set 定义存储器,将输入参数赋值给属性_name。注意,存储器不允许标注返回值类型,即便是无返回值也不允许使用关键字 void。

第 12～13 行,分别调用存储器 set name("张珊")和访问器 get name()。

执行结果为:

```
张珊
```

注意,如果编译时有如下报错:

```
TS1056: Accessors are only available when targeting ECMAScript 5 and higher.
```

是因为编译的 ES 版本不支持访问器语法,此时可指定支持该语法的 ES 版本进行编译,如:

```
tsc -t es5 notes.ts
```

访问器和函数一样,也可以是静态的。

【例 3-10】 定义静态的存储器和访问器

```
1.    class Emp {
2.         private static _count : number = 0
3.         constructor(){ Emp._count++}
4.         static get count() : number{     //定义静态访问器
5.              return Emp._count
6.         }
7.         static set count(num : number){ //定义静态存储器
8.              Emp._count = num
9.         }
10.   }
```

```
11.    console.log(Emp.count)       //0, 实际调用静态访问器: get count()
12.    new Emp()
13.    console.log(Emp.count)       //1, 实际调用静态访问器: get count()
14.    Emp.count = 9                //1, 实际调用静态存储器: set count(num : number)
15.    console.log(Emp.count)       //9, 实际调用静态访问器: get count()
```

第 4～6 行,用关键字 static 定义静态访问器 get count(),用于返回类 Emp 的静态属性_count 的值。

第 7～19 行,用关键字 static 定义静态存储器 set count(num : number),用于设置 Emp 的静态属性_count 的值。

第 11 行、第 13 行、第 15 行,代码 Emp.count 实际调用了静态访问器 get count()。

第 14 行,代码 Emp.count=9 实际调用了静态存储器 set count(num : number)。

执行结果为:

```
0
1
9
```

3.1.5　构造函数

构造函数又称构造器(constructor),简称构造,是用于创建对象的特殊函数。

对象又称实例。用构造函数创建对象的过程,即为类的实例化过程。

TypeScript 没有指定构造函数时,系统会生成一个默认的无参构造函数。TypeScript 构造函数用 constructor 命名,使用 new 关键字加类名会调用构造函数进行对象的创建。

通常,在实例化对象的过程中会同时初始化其属性值。在构造函数中,可通过关键字 this 来访问当前对象的属性和函数。

【例 3-11】 用构造函数对属性进行初始化

```
1.    class Dog{
2.         name : string
3.         constructor( name : string ){
4.         this.name = name
5.         }
6.    }
7.    let doggie = new Dog("doggie")
8.    console.log(doggie)
```

第 3～5 行,用关键字 constructor 定义了 Dog 类的构造函数。this.name 使用关键字 this 引用了 name 属性,因此 this.name=name 是用初始化属性 name 的值为输入参数 name 赋值。

第 7 行,用关键字 new 调用构造函数,创建 Dog 类对象 doggie。注意,在调用构造函数时,使用了类名 Dog 而非 constructor;传入的实际参数值"doggie"将初始化对象的 name 属性值。

第 8 行，用 console.log()函数输出对象 doggie 的信息。

执行结果为：

```
Dog { name: 'doggie' }
```

注意，若构造函数只对属性进行初始化，则可使用“初始化属性速记写法”，令代码更简洁易读。

【例 3-12】 对属性进行初始化，构造函数可使用“初始化属性速记写法”

```
1.    class Cat{
2.         constructor(public name : string ){}
3.    }
4.    let kitty = new Cat('kitty')
5.    console.log(kitty)
```

第 2 行，构造函数中有形式参数 name，但在花括号{}中没有明确对属性 name 赋值，实际上，Cat 类甚至没有声明过 name 属性。

第 4 行，通过构造函数创建对象 kitty。

第 5 行，用 console.log()函数输出对象 kitty。

执行结果为：

```
Cat { name: 'kitty' }
```

Cat 对象内多了属性 name，且值被初始化了。这说明第 2 行构造函数的写法是种初始化属性的缩略结构。它允许在不声明属性的情况下，构造函数写入的参数将成为实例属性，并进行初始化操作。

注意，第 2 行中构造函数的参数上有访问修饰符 public(也可为 protected 或 private)，若不写访问修饰符，是不会生成相应属性和对属性进行初始化赋值的。


视频讲解

3.2 对象

在 TypeScript 中，对象可以被视为包含一组键值对的实例，因此，对象可以用字面方式(即使用花括号{})来创建。对象作为类的实例，当然可以通过类的构造函数方式来创建；另外，所有对象都是 Object 的子类对象，因此对象也可以使用 new Object()方式创建。

3.2.1 对象概述

对象是现实问题域中的实体在计算机程序中的映射。

现实中的实体都有自己的特征(或状态)和行为(或功能)，如通讯录应用中的李思同学。李思有特征：编号“002”、姓名“李思”、性别“男”、电话号码“138××××0686”、联系地址“北京市双清路 30 号”；李思也有行为：修改电话号码、修改联系地址。

在 TypeScript 中，对象可被看成包含一组键值对的实例。其值可以是原始类型，也可

以是数组、函数、对象等。

TypeScript 映射实体时，一般将实体的特征、状态转换为属性，将行为、功能转换为函数。映射实体李思同学为对象 lisi，可直观地表示为例 3-13 所示代码。

【例 3-13】 映射实体李思同学为对象 lisi

```
let lisi = {
      sno : '002',
      name : '李思',
      sex : '男',
      tel : '138×××0686',
      addr : '北京市双清路 30 号',
      changeTel : function(tel : string) : void {
          this.tel = tel
      },
       changeAddr : function(addr : string) : void {
         this.addr = addr
       }
}
```

实体也可以是相对抽象的事物。以实体“圆”为例：其半径值为该实体“圆”的特征、计算半径和计算面积就是该实体“圆”的 2 个功能。

【例 3-14】 实体可以是抽象事物，比如“圆”

```
let circle = {
     radius : 10,
     getCircumference : function() : number{
         return 2 * 3.14 * this.radius
     },
     getArea : function() : number{
         return 3.14 * this.radius * this.radius
     }
}
```

3.2.2　创建对象

实际上，在 TypeScript 中创建对象有多种方式。除了直接用字面方式创建对象外，还可以用 new Object()方式创建，或用构造函数创建。

1. 字面方式创建对象，在定义结构的同时创建对象

【例 3-15】 用字面方式创建对象

```
enum Color{Red, Blonde, Brown, While, Black, Gray}
let doggie = {
     name : 'awang',
     color : Color.Blonde,
     birth : new Date(2020,3,6),
```

```
6.          skills : [],
7.          eat : function(){},
8.          bark : () => {}
9.      }
10.     console.log(doggie.birth)
```

第 2～9 行，用字面方式创建对象，内部包含一组键值对。

第 3～6 行，定义 4 个不同类型的属性，分别是 string、枚举、日期和数组类型。

第 7～8 行，定义 2 个函数，第 7 行为普通函数、第 8 行为箭头函数。

第 10 行，用 console.log()函数输出对象 doggie 的信息。

执行结果为：

```
{
  name: 'awang',
  color: 1,
  birth: 2020-04-05T16:00:00.000Z,
  skills: [],
  eat: [Function: eat],
  bark: [Function: bark]
}
```

从运行结果看，字面方式成功创建了 doggie 对象。

2. 用 new Object()方式创建空对象，再按需追加对象属性和函数

在 TypeScript 中，Object 类是所有类的终极父类，new Object()就是用 Object 类的构造函数创建一个空对象。

【例 3-16】 用 new Object()方式创建对象

```
1.      let dogB : any = new Object()                    //any类型
2.      dogB.name = "awang"
3.      dogB.eat = function(){}
4.      dogB.eat()
```

第 1 行，用 new Object()创建对象 dogB。注意，应声明 dogB 的类型为 any，否则后面为对象动态添加属性或函数时会出现如下报错：

```
Property 'name' does not exist on type 'Object'
```

第 2～3 行，分别动态添加属性和函数。

第 4 行，调用动态增加的 eat()函数。

以上 new Object()写法还可以用 Object.create(null)方式等价替代。

【例 3-17】 用 Object.create(null)方式创建对象

```
1.      const dogC = Object.create(null)
2.      dogC.name = "awang"
3.      dogC.eat = function(){}
4.      dogC.eat()
```

3. 由类的构造函数创建对象

3.1.5 节已经介绍过，可使用类的构造函数创建对象。

【例 3-18】 用类的构造函数创建对象

```
1.    class Dog{
2.        constructor(public name:string ){}
3.    }
4.    let doggie = new Dog('aWang')
5.    console.dir(doggie )
```

第 1～3 行，定义 Dog 类，内部构造函数使用“初始化属性速记写法”对属性 name 进行初始化。

第 4 行，用关键字 new 调用构造函数，创建了 Dog 类对象 doggie。

执行结果为：

```
Dog { name: 'aWang' }
```


视频讲解

3.3　继承

继承是面向对象编程中实现“类扩展”的机制，是类层次上的代码复用。

继承就相当于将父类的属性和函数直接定义到了子类中，子类可直接使用这些继承的属性和函数。

3.3.1　继承语法

在 TypeScript 中，用关键字 extends 指明继承关系。

继承的基础语法结构如下所示：

```
class 子类 extends 父类
{
    类结构体(属性、构造、函数)
}
```

父类(parent class)又被称为基类(base class)、超类(super class)，子类(sub class)又被称为派生类(derived class)。

【例 3-19】 子类继承父类

```
1.    class Animal{
2.         constructor(public name : string){}
3.         move(){
4.              console.log(this.name + ' moved')
5.         }
6.    }
7.    class Dog extends Animal{}
```

```
8.    const doggie = new Dog('Kipper')
9.    console.log(doggie)
10.   doggie.move()
```

第 1～6 行，定义 Animal 类，该类包含一个构造函数和一个普通函数。其中第 2 行是构造函数，该构造函数使用了“初始化属性速记写法”，会为 Animal 类加上对属性 name。

第 7 行，用关键字 extends 实现 Dog 类对 Animal 类的继承。此处，子类 Dog 中没有定义属性和函数，但实际上通过继承，相当于 Dog 类有了来自父类的属性 name 和 move()函数。

第 8 行，用构造函数创建了 doggie 对象。

第 9～10 行，用 console.log()函数输出对象 doggie 的信息，并调用对象 doggie 的 move()函数。

执行结果为：

```
Dog { name: 'Kipper' }
Kipper moved
```

从结果看，子类 Dog 继承了父类 Animal 的属性 name 和 move()函数。

3.3.2 单继承

在 TypeScript 中，类是单继承的，即只能继承一个类，不支持继承多个类。

【例 3-20】 继承多个父类导致编译时报错

```
1.    class Father{}
2.    class Mother{}
3.    class Son extends Father, Mother{}
```

第 3 行，Son 类继承了两个父类，显然继承多个父类会导致语法出错。将鼠标移至下画波浪线处，可观察到相应的错误提示，如图 3-1 所示。

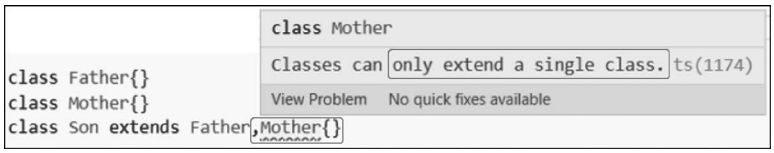

图 3-1 继承多个父类导致语法出错

虽然类不允许多继承，但是允许 A 类继承 B 类，B 类继承 C 类……这种链式的继承满足单继承语法要求。

【例 3-21】 类的单继承“链”

```
1.    class Animal{                          //动物
2.        constructor(public name:string){}
3.    }
```

```
4.    class Mammal extends Animal{              //哺乳动物继承动物
5.         static breastFeed = true
6.         constructor(name:string) {
7.             super(name)
8.         }
9.    }
10.    class Panda extends Mammal{              //熊猫继承哺乳动物
11.         eatBamboo():void{}
12.         constructor(name:string) {
13.             super(name)
14.         }
15.   }
16.   console.log(Panda.breastFeed)
17.   let panpan = new Panda('PanPan')
18.   console.log(panpan.name)
```

第 1～3 行，定义 Animal 类。

第 2 行，使用构造函数的“初始化属性速记写法”，为 Animal 类添加属性 name。

第 4～9 行，定义继承 Animal 类的 Mammal 类，并增加 1 个静态属性 breastFeed。

第 10～15 行，定义继承 Mammal 类的 Panda 类，并增加 1 个 eatBamboo()函数。

第 16 行，用 Panda 类直接调用来自其父类 Mammal 的静态属性 breastFeed。

第 17 行，用构造函数创建 Panda 对象，并通过参数初始化来自父类 Animal 的属性 name。

第 18 行，通过 panpan.name，访问 panpan 对象继承自 Animal 类的 name 属性。

执行结果为：

```
true
PanPan
```

这说明，TypeScript 是允许单链继承的，即子类可继承父类及其“祖先类”的属性和函数。

3.3.3　函数覆盖与多态

1. 函数覆盖

继承后，若子类中定义了与父类函数签名完全相同的函数（名称相同，参数个数和类型也相同），则被称为函数的覆盖（override）或重写。

【例 3-22】　函数覆盖

```
1.    class Animal{
2.         eat() : void {
3.             console.log('animal eated')
4.         }
5.    }
6.    class Dog extends Animal{
7.         eat() : void {
```

```
8.              console.log('dog eated')
9.          }
10.     }
11.     let doggie = new Dog()
12.     doggie.eat()
```

第 1～10 行，分别定义父类 Animal 和子类 Dog。父类和子类中都有 eat()函数，并且函数名和参数形式都相同(都没有参数)。此时子类中的 eat()就覆盖了父类中的 eat()函数。

第 11～12 行，创建子类 Dog 的对象 doggie，并调用 eat()函数。注意，此时执行的 eat()函数应该是子类中的函数。

执行结果为：

```
dog eated
```

从结果看，调用的确实是子类中的函数。若要调用从父类继承的 eat()函数，可使用关键字 super。

【例 3-23】 使用关键字 super 调用从父类继承的函数

```
1.      class Dog extends Animal{
2.          eat(): void {
3.              console.log('dog eated')
4.              super.eat()
5.          }
6.      }
```

第 4 行，使用关键字 super 调用 eat()函数，该函数为从父类继承的。

注意，关键字 super 代表父类对象，因此可用 super 调用父类对象的属性和函数。

执行结果为：

```
dog eated
animal eated
```

2. 多态

在掌握了函数覆盖的基础上，可进一步深入理解多态特征。

多态(polymorphism)是面向对象编程的一个重要特征。在多态中，同一个函数名可以在不同的类中具有不同的实现。当调用实例的属性和函数时，会根据实例的实际类型进行动态调用，而不是根据声明类型进行调用。

多态行为使开发人员能够以统一的方式处理不同类的对象，而无须关心具体的对象类型。合理应用多态能够提高代码的重用性和可维护性。

【例 3-24】 多态特征：根据实例的“实际类型”调用相应的函数

```
1.      class Shape{                          //图形
2.          draw(){ console.log('draw a shape'); }
3.      }
```

```
4.    class Circle extends Shape{        //圆
5.         draw(){ console.log('draw a circle'); }
6.    }
7.    class Square extends Shape{        //正方形
8.         draw(){ console.log('draw a square'); }
9.    }
10.   class ShapeTool{                   //工具类
11.        static doDraw(shape:Shape){
12.             shape.draw()
13.         }
14.   }
15.   let s1 : Shape = new Circle()
16.   ShapeTool.doDraw(s1)
17.   ShapeTool.doDraw(new Square())
```

第 1～9 行，分别定义 3 个类，其中 Shape 是父类，Circle 和 Square 都是 Shape 的子类。3 个类中都有 draw()函数，子类中的 draw()函数覆盖了父类中的 draw()函数。

第 10～14 行，定义工具类 ShapeTool，内部有静态 doDraw()函数。注意，参数 shape 的类型为父类 Shape，在函数内调用的是 shape 的 draw()函数。

第 15～16 行，创建对象 s1。注意，s1 的声明类型是 Shape，但实际类型为 Circle。当执行 ShapeTool.doDraw(s1)时，传入 s1，因此执行 s1.draw()时，执行的是实际类型 Circle 的函数，而非声明类型 Shape 的函数。这正是多态的表现。

第 17 行，创建 Square 对象并传入 ShapeTool 的 doDraw()函数，同样执行的是实际类型 Square 的函数，而非声明类型 Shape 的函数。

执行结果为：

```
draw a circle
draw a square
```

3.3.4　this 与 super

前面几个章节中多次提到两个关键字：this 和 super。这里系统地总结一下两者的区别和相应的使用场合。

this 代表当前对象本身，而 super 代表父类对象。

1. 访问属性和函数

this 用于访问当前对象中的属性和函数，当属性和函数不存在时，会自动调用继承自父类对象的属性和函数。使用 super 则可直接调用父类对象中的属性和函数。

【例 3-25】 用 this 和 super 分别调用自身的函数和父类函数

```
1.    class Parent{
2.         doSth() : void {
3.             console.log('parent do sth');
4.         }
5.    }
```

```
6.    class Derived extends Parent{
7.         doSth() : void {
8.             console.log('derived do sth')
9.         }
10.        test(){
11.            this.doSth()
12.            super.doSth()
13.        }
14.   }
15.   let obj = new Derived()
16.   obj.test()
```

第 1～5 行，定义 Parent 类，该类内部有一个 doSth()函数。

第 6～14 行，定义 Parent 类的子类 Derived，子类的内部也定义了一个 doSth()函数，此外还有一个测试 this 和 super 关键字用法的 test()函数。注意，在 test()中，用 this.doSth()和 super.doSth()分别调用了本类对象的函数和父类对象的函数。

执行结果为：

```
derived do sth
parent do sth
```

2. 用 super 调用父类的构造函数

在子类的构造函数中，可使用 super 调用父类的构造函数。需要注意的是，在子类的构造函数中，用 super 调用构造函数的代码必须放在有效的第一行上。

【例 3-26】 用 super 调用父类构造

```
1.    class Tag {
2.         constructor(public name : string){}
3.    }
4.    class Img extends Tag{
5.         src : string
6.         constructor(name : string, src : string){
7.             super(name)
8.             this.src = src
9.         }
10.   }
11.   let img = new Img('logo', 'img/logo.png')
12.   console.log(img)
```

第 1～3 行，定义类 Tag。第 2 行定义构造函数，该构造函数使用“初始化属性速记写法”对属性 name 进行初始化。

第 4～10 行，定义类 Tag 的子类 Img，所以类 Img 会继承类 Tag 的属性 name。

第 5 行，在类 Img 中增加属性 src。

第 6～10 行，定义类 Img 的构造函数。其中第 7 行 super(name)就是调用父类的构造函数，因此会对属性 name 进行初始化；第 8 行对属性 src 进行初始化。

注意，super()必须放在子类的构造函数的第一行，否则会报错。这是因为，创建本类对象前，需先创建父类对象，以便能继承父类对象的属性。

第 11～12 行，通过构造函数创建类 Img 的对象 img，并用 console.log()函数输出对象 img 的信息。

执行结果为：

```
Img { name: 'logo', src: 'img/logo.png' }
```


视频讲解

3.4　抽象类

抽象类(abstract class)是一种特殊的类。和普通类一样，抽象类可以有属性和函数，但不允许用构造函数直接创建对象。

抽象类通常作为其他类的父类存在，它的主要作用是为子类提供一个通用的模板，定义一些共同的属性和函数。子类需要实现(重写)在抽象类中声明的抽象函数才能创建对象。

在抽象类中，可以定义抽象函数和普通函数。抽象函数是没有具体实现的函数，只有函数签名，因此子类必须实现这些抽象函数。而普通函数则可以有具体的实现。

在 class 前加关键字 abstract 来定义抽象类。

【例 3-27】　定义抽象类

```
1.    abstract class Shape {
2.         public abstract Draw() : void
3.    }
4.    let s = new Shape()                    //抽象类不能直接创建，会报错
```

第 1 行，关键字 class 前加了关键字 abstract，说明该类是抽象类。

第 2 行，在函数前加 abstract，说明该函数是抽象函数。所谓抽象函数，就是只有函数签名，没有函数体的函数。

注意，若类内部有抽象函数，则相当于类结构部分抽象，那么类在整体上就是个抽象类，此时，必须在 class 前加 abstract 关键字，将该类定义为抽象类。反之，抽象类中的函数可以都是非抽象的，并不要求一定要存在抽象函数。

此外，抽象函数是不能加花括号{}的，因为花括号{}代表“实现”，这样的函数就不能被称作抽象函数了。

第 4 行，创建抽象类 Shape 的对象，会有如下报错：

```
Cannot create an instance of an abstract class.
```

这说明抽象类无法用构造函数直接创建对象，但可以通过子类进行间接创建。

【例 3-28】　通过子类间接创建抽象类对象

```
1.    abstract class Shape {
2.         constructor(public name : string){}
```

```
3.          public abstract draw(): void
4.      }
5.      class Circle extends Shape{
6.          public draw() : void {
7.              console.log('draw a circle')
8.          }
9.          constructor(){
10.             super('circle')
11.         }
12.     }
13.     let c = new Circle()
14.     console.log(c)
```

第 1～4 行，用关键字 abstract 定义抽象类 Shape。

第 5～12 行，定义了抽象类 Shape 的子类 Circle。

第 6～8 行，在子类 Circle 中，对抽象父类 Shape 中的抽象函数 draw()进行实现。

注意，子类中若不实现继承的抽象函数，则子类在整体上就是抽象类，必须用关键字 abstract 标注该子类为抽象类，否则就会出现语法错误。

第 9～11 行，子类 Circle 定义了自己的构造函数，并用关键字 super 调用父类构造函数来初始化继承的 name 属性。

第 13～14 行，创建子类 Circle 的对象 c。注意，在创建子类过程中，会先创建父类对象。虽然此时父类是抽象的，但这并不妨碍它的对象被创建出来。用 console.log()函数输出对象 c 的信息时，可观察到继承自抽象父类的 name 属性值，如下所示：

```
Circle { name: 'circle' }
```

3.5 接口

接口(interface)可被视为更为彻底的抽象类。

抽象类中允许有具体实现函数，甚至可以全部为具体实现函数。而接口本身并不包含任何具体实现函数，仅用于声明必须实现哪些函数，函数的具体功能由子类实现。为此，接口的子类又被称为接口的实现类。

3.5.1 定义接口

在接口中，可以定义属性和抽象函数。另外，与类的单一继承不同，接口可以继承多个必接口。

定义接口的语法如下所示：

```
interface 接口名 [ extends 接口 1, 接口 2 …] {
    [属性…]
    [抽象函数…]
}
```

关键字 interface 用于定义接口；关键字 extends 用于继承父接口；接口中可定义属性和抽象函数。

接口是定义了功能的契约，对子类起规范作用，即实现接口的具体子类必须实现接口中的所有抽象函数。

【例 3-29】 用关键字 interface 定义接口 IShape

```
1.    interface IShape{
2.        name:string
3.        // pi:number = 3.14
4.        //static pi:number
5.        draw():void
6.    }
```

第 1 行，用关键字 interface 定义接口 IShape。

第 2 行，声明属性 name。

注意，在接口中，属性不能初始化赋值，如第 3 行会出错。另外，在接口中不允许声明静态属性，如第 4 行会出错。

第 5 行，声明抽象函数 draw()。注意，接口函数默认就是抽象的，接口中的函数不用加也不能加 abstract 关键字。

3.5.2　接口实现类

接口是定义了功能的契约。当类通过关键字 implements 声明要实现接口时，必须具体化接口中所有的抽象函数和属性，否则说明该类违背了实现接口的契约，会出错。

【例 3-30】 接口的实现

```
1.    interface IShape {
2.        getCircumference() : number                    //声明抽象函数
3.        getArea() : number                             //声明抽象函数
4.    }
5.    class Circle implements IShape{
6.        constructor(public radius : number){}
7.        getCircumference() : number {                  //实现抽象函数
8.            return 2 * Math.PI * this.radius
9.        }
10.       getArea() : number {                           //实现抽象函数
11.           return Math.PI * this.radius * this.radius
12.       }
13.   }
14.   class Square implements IShape{
15.       constructor(public sideLen : number){}
16.       getCircumference() : number {                  //实现抽象函数
17.           return 4 * this.sideLen
18.        }
19.        //遗漏 getArea()实现
20.   }
```

第 1～4 行，定义接口 IShape，在接口中声明两个抽象函数。

第 5～13 行，定义接口 IShape 的实现类 Circle。关键字 implements 声明子类要实现接口。因为在类 Circle 中实现了接口 IShape 所有声明的函数，因此语法没有问题。

第 14～20 行，定义接口 IShape 的实现类 Square。因为遗漏了 getArea()函数的实现，因此语法会出错，如下所示：

```
Class 'Square' incorrectly implements interface 'IShape'.
Property 'getArea' is missing in type 'Square' but required in type 'IShape'.
```

3.5.3 接口多继承

在 TypeScript 中，类之间是单继承的，但接口允许多继承。一旦继承，则子接口将继承所有父接口中的属性和抽象函数。

【例 3-31】 接口的多继承

```
1.    interface Flyable { Fly() : void }
2.    interface Singable { Sing() : void }
3.    interface IfcBird extends Flyable, Singable {}
```

第 3 行，IfcBird 接口用关键字 extends 继承了两个接口，接口间用逗号分隔。

【例 3-32】 实现接口时，它继承的所有函数都得实现

```
1.    interface Flyable { fly() : void }
2.    interface Singable { sing() : void }
3.    interface IfcBird extends Flyable, Singable { jump() : void }
4.    class Parrot implements IfcBird {
5.         fly() : void {}
6.         sing() : void {}
7.         jump() :void{}
8.    }
```

第 3 行，通过多继承，接口 IfcBird 继承了 Flyable 和 Singable 两个接口，同时声明了抽象函数 jump()。

第 4～8 行，类 Parrot 实现了 IfcBird 接口，而接口 IfcBird 又继承了 Flyable 和 Singable 两个接口。因此 Parrot 类需实现 3 个接口中的全部 3 个函数。

3.6 实战闯关——面向对象

针对面向对象编程，需掌握的重点知识和技能为：类的设计、属性设置、类的继承、函数覆盖、接口设计等。

【实战 3-1】 类设计

定义一个商品类，要求如下：

商品属性包括商品编号、商品名称、商品所在分类编号、商品价格、商品图片 URL。

商品函数包括修改商品分类编号、修改商品价格、修改商品图片 URL。

【实战 3-2】 属性设置

定义一个员工类。类中有实例属性 name(姓名)和 salary(工资),还有静态属性 count(员工数)。当创建一个员工对象时,count 的值需加 1。

【实战 3-3】 父类与子类的继承和函数覆盖

定义父类 Shape(图形):内有 whoAmI()函数,输出“我是一个图形”。

定义类 Shape 的子类 Square(正方形):内有代表边长的 width 属性,以及用于计算面积的 getArea()函数。

在 Square 中也定义一个 whoAmI()函数,输出“我是一个边长为 width 的正方形”,其中的 width 用 width 属性值代替。

【实战 3-4】 接口与实现类

定义接口 ICustomer,它有 4 个属性:

name(姓名)、tel(联系电话)、addr(联系地址)和 level(客户级别),还有一个用于获取折扣率的抽象函数 getDiscountRate()。

定义接口 ICustomer 的两个实现类 Customer 和 VIP:

Customer 类,设置 Level 的默认值为 0,实现 getDiscountRate()函数返回 1。

VIP 类,设置 Level 的默认值为 1,实现 getDiscountRate()函数返回 0.95。

【实战 3-5】 接口的继承

定义接口 Point,它有 2 个代表坐标的属性:x 和 y,都为 number 类型。

定义接口 Point3D,在 Point3D 内有 3 个代表坐标的属性 x、y 和 z,都为 number 类型。注意,其中 x 和 y 属性需继承自接口 Point。

声明 Point3D 类型变量 p3,将 3 个属性值分别设置为 1、2 和 3。