1. class介绍

类封装了属性和方法，是面向对象编程的基本结构

1. 属性的类型

类的属性可以在顶层声明，也可以在构造方法内部声明。

对于顶层声明的属性，可以在声明的同时给出类型。

如果声明时没有初始化，也不给出类型，ts会认为是any类型。

```
class Point {
  x:number;
  y:number;
}
```

如果声明时给了初始值，不写类型，ts会自动推断属性的类型

ts中有一个编译选项strictPropertyInitialization，只要打开(默认是打开的)，对于顶层声明的属性就会检测是否设置了初始值，如果没有就会报错。

如果开启了编译选项strictPropertyInitialization，但是没有初始化值，还不想报错，可以使用**非空断言，就是在属性名后添加感叹号，表示这两个属性肯定不会为空，见类型断言。**

```
class Point {
  x!: number;
  y!: number;
}
```

2. readonly修饰符

属性名前加上readonly修饰符，表示该属性是只读的，实例对象不能修改这个属性。

readonly修饰符要加在顶层声明属性名前面，不能写在构造方法内。

readonly属性的初始值，可以同时写在顶层属性和构造方法里面，如果同时写，会以构造方法中的初始值为准

实例对象a，想修改属性id，会报错。

```
class A {
  readonly id:string;

  constructor() {
    this.id = 'bar'; // 正确
  }
}
const a = new A();
console.log(a.id); // 'bar'
a.id = 'str'; // 报错

```

3.方法的类型

类的方法就是普通函数，类型声明方式和函数一样。

```
class Point {
  x:number;
  y:number;

  constructor(x:number, y:number) {
    this.x = x;
    this.y = y;
  }

  add(point:Point) {
    return new Point(
      this.x + point.x,
      this.y + point.y
    );
  }
}
```

类的方法跟普通函数一样，可以使用参数默认值，以及函数重载。

类的构造方法不能声明返回值类型，因为返回值是永远是实例对象。

4. 存取器方法

包含取值器getter和存值器setter两种方法

getter用于读取属性，setter用于存入属性

```
class C {
  _name = '';
  get name() {
    return this._name;
  }
  set name(value) {
    this._name = value;
  }
}
```

set函数规则

- 如果某个属性只有get方法，没有set方法，那么该属性默认成为只读属性。
- ts5.1之前，set方法的参数类型必须兼容get方法的返回值类型，否则会报错。ts5.1之后，可以不用兼容
- get方法与set方法的可访问性必须一致，要不都为公开方法，要么都是私有方法。

5. 属性的索引

类允许定义属性的索引。

[s\:string]表示所有属性名类型为字符串的属性，它们的属性值要不是布尔值，要么是返回布尔值的函数。

```
class MyClass {
  [s:string]: boolean |
    ((s:string) => boolean);

  get(s:string) {
    return this[s] as boolean;
  }
}
```

类的方法是一种特殊的属性（属性值是函数），所以如果一个对象同时定义了属性索引和方法，属性索引的类型定义也要包含方法，否则会报错。

```
class MyClass {
  [s:string]: boolean;
  f() { // 报错
    return true;
  }
}
```

```
class MyClass {
  [s:string]: boolean | (() => boolean);
  f() {
    return true;
  }
}
```

属性的get、set方法，虽然是一个函数的方法，但是它们被认为是一个属性，所以属性索引的类型定义时不用考虑set、get方法。

2. 类的interface接口

1. implement关键字

interface接口和type别名，可以用对象的形式为类指定一组检查条件，**类可以使用implement关键字，用来校验当前的类是否满足这些类型的限制，但是类中必须声明外部规定的属性以及属性类型，否则会报错**

```
interface A {
  s: string;
}

class B implements A {
//   s: string = '这是外部限定的属性名'; // 报错
    s: string = '这是外部限定的属性名'
}

```

类也可以定义接口没有声明的属性和方法

```ts
interface Point {
  x: number;
  y: number;
}

class MyPoint implements Point {
  x = 1;
  y = 1;
  z:number = 1;
}
```

`implements`关键字后面，不仅可以是接口，也可以是另一个类。这时，后面的类将被当作接口。**此时该类也要实现这个类的所有的属性和方法，跟interface、type一样。**

```ts
class Car {
  id:number = 1;
  move():void {};
}

class MyCar implements Car {
  id = 2; // 不可省略
  move():void {};   // 不可省略
}
```

注意点：**interface描述的是类的对外接口，也就是公开的属性和方法，不能定义为私有属性和方法。**

因为ts中，私有属性应该是在类的内部实现，接口作为模板，不涉及类的内部代码写法。

```ts
interface Foo {
  private member:{}; // 报错
}
```

2. 实现多个接口

类可以实现多个接口（多重限制），每个接口之间使用逗号分隔。多重实现即一个接口同时实现多个接口，不同接口之间的同名属性的类型不能冲突。

这里类Car同时实现了三个接口，类Car必须要有这三个接口声明的所有属性和方法。

```ts
class Car implements MotorVehicle, Flyable, Swimmable {
  // ...
}
```

同时实现多个接口会容易使代码很难管理，解决方法

- 类的继承，就是先继承类，再实现其他接口

```ts
class Car implements MotorVehicle {
}
class SecretCar extends Car implements Flyable, Swimmable {
}
```

- 接口继承，就是将多个接口继承成一个接口，此时就直接实现一个新接口即可

3. 类与接口的合并

ts中不允许两个同名的类，如果一个接口和一个类同名，接口会被合并到类中，合并时如果有同名的属性，该属性的类型必须一致，否则会报错。

```
class A {
  x:number = 1;
}

interface A {
  y:number;
}

let a = new A();
a.y = 10;

a.x // 1
a.y // 10

```

3. Class类型

1. 实例类型

ts的类本身就是一种类型，代表该类的实例类型，而不是class的自身类型。

这里定义了类Color，类名就代表一种类型，实例对象green就属于该类型。

```
class Color {
  name:string;

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

const green:Color = new Color('green');
```

对于引用实例对象的变量来说，既可以声明类型为Class，也可以声明为interface，因为都代表实例对象的类型。**但是此时如果类中有自己定义的属性和方法，变量类型声明为interface的，则没有类中自己定义的属性和方法**

```
interface MotorVehicle {
    num: number;
}
class Car implements MotorVehicle {
    num: number = 1;
    name: string = 'hello world';
}

// 写法一
const c1:Car = new Car(); // c1 num
// 写法二
const c2:MotorVehicle = new Car(); // c2 num、name
```

类作为类型使用时，只能表示实例的类型，不能表示类自身的类型。

```
class Point {
  x:number;
  y:number;

  constructor(x:number, y:number) {
    this.x = x;
    this.y = y;
  }
}

// 错误
function createPoint(
  PointClass:Point,
  x: number,
  y: number
) {
  return new PointClass(x, y);
}
```

2. 类的自身类型

要获取一个类的自身类型，可以使用typeof运算符

```
function createPoint(
  PointClass:typeof Point,
  x:number,
  y:number
):Point {
  return new PointClass(x, y);
}
```

js 语言中，**类只是构造函数的一种语法糖，本质上是构造函数的另一种写法。所以，类的自身类型可以写成构造函数的形式。**

```ts
interface PointConstructor {
  new(x:number, y:number):Point;
}

function createPoint(
  PointClass: PointConstructor,
  x: number,
  y: number
):Point {
  return new PointClass(x, y);
```

3. 结构类型原则

Class也遵循**结构类型原则**，即一个对象只要满足Class的实例结构，就跟该Class属于同一个类型。

如果两个类的实例结构相同，那么这两个类就是兼容的，可以用在对方的使用场合。总之，只要 A 类具有 B 类的结构，哪怕还有额外的属性和方法，ts也会认为 A 兼容 B 的类型。

```ts
class Person {
  name: string;
}

class Customer {
  name: string;
}

// 正确
const cust:Customer = new Person();
```

如果某个对象跟某个class的实例结构相同，也很认为两者的类型相同。

```ts
class Person {
  name: string;
}

const obj = { name: 'John' };
const p:Person = obj; // 正确
```

空类不包含任何成员，任何其他类都可以看作与空类结构相同。因此，凡是类型为空类的地方，所有类（包括对象）都可以使用。

两个类的兼容，只检查实例成员，不考虑静态成员和构造方法。

如果类中存在私有成员（private）或保护成员（protected），那么确定兼容关系时，ts 要求私有成员和保护成员来自同一个类，这意味着两个类需要存在继承关系。

4. 类的继承

类(子类)可以使用extends关键字继承另一个类(基类)的所有属性和方法

```
class A {
  greet() {
    console.log('Hello, world!');
  }
}

class B extends A {
}

const b = new B();
b.greet() // "Hello, world!"
```

根据**结构类型原则**，子类也可以用在类型为基类的场合。

如果子类在继承时会覆盖基类的同名，两者的类型不能冲突。

`extends`关键字后面不一定是类名，可以是一个表达式，只要它的类型是构造函数就可以了。

当编译设置的target大于2022时，对于那些子类只设置了类型、没有初值的顶层属性在基类中被赋值后，会被重置为undefined。解决方法：使用declare命令，去声明顶层成员的类型，告诉ts这些成员的赋值是由基类实现的。

5. 可访问性修饰符

类内部的成员是否让外部访问，可以使用public、private和protected三个修饰符决定。

1. public

表示公开成员，外部可以自由访问，默认不用写。

2. private

表示私有成员，只能在当前类的内部使用，类的实例和子类都不能使用。

子类不能定义父类私有成员的同名成员。

其实private定义的私有成员并不是真正意义的私有成员，因为当编译成js后，就没有该关键字了，在ES2022发布了私有成员的写法#propName，所以可以直接使用js的写法。

```
class A {
  #x = 1;
}

const a = new A();
a['x'] // 报错
```

3. protected

表示该成员是保护成员，只能在类的内部、子类内部可以使用该成员，实例无法使用。

子类还可以定义同名成员，如果子类定义成public，则外界也可以读取这个属性。

```
class A {
  protected x = 1;
}

class B extends A {
  x = 2;
}
```

4. 实例属性的简写形式

在开发中很多实例属性的值，是通过构造方法传入的，但是在类中要对同一个属性声明两次类型，一次是在类的头部，一次是在构造方法的参数里面，很麻烦。

```
class Point {
  x:number;
  y:number;

  constructor(x:number, y:number) {
    this.x = x;
    this.y = y;
  }
}
```

简写：此时的修饰符public不能简写，除了public还有private、protected、readonly，并且readonly还能与其他三个可访问修饰符一起使用。

```
class Point {
  constructor(
    public x:number,
    public y:number
  ) {}
}

const p = new Point(10, 10);
p.x // 10
p.y // 10
```

6. 静态成员

用static关键字，定义静态成员，静态成员只能通过类本身使用，不能通过实例对象使用。

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

MyClass.x // 0
MyClass.printX() // 0
```

`static`关键字前面可以使用 public、private、protected 修饰符。

静态私有属性也可以用 ES6 语法的`#`前缀表示

其中`public`和`protected`的静态成员可以被继承。

```
class A {
  public static x = 1;
  protected static y = 1;
}

class B extends A {
  static getY() {
    return B.y;
  }
}

B.x // 1
B.getY() // 1
```

7. 泛型类型

类也可以写成泛型，使用类型参数，见**泛型**

类Box的类型参数Type，属于泛型类。实例创建时，变量的类型声明需要带有类型参数的值。

```
class Box<Type> {
  contents: Type;

  constructor(value:Type) {
    this.contents = value;
  }
}

const b:Box<string> = new Box('hello!');
```

静态成员不能使用泛型的类型参数

```
class Box<Type> {
  static defaultContents: Type; // 报错
}
```

8. 抽象类、抽象成员

在类定义的前面加上关键字abstract，表示该类不能实例化，只能当做其他类的模板，这种类叫做抽象类。

```
abstract class A {
  id = 1;
}

const a = new A(); // 报错
```

抽象类只能当做基类，在它的基础上定义子类。

抽象类的子类也可以是抽象类，也就是抽象类可以继承其他抽象类

```
abstract class A {
  foo:number;
}

abstract class B extends A {
  bar:string;
}
```

抽象类的内部可以有已经定义好的属性和方法，如果有没有实现的属性和方法，需要在前面加上abstract关键字，这种没有实现的属性和方法叫做抽象成员，**表示该方法需要子类实现，要是子类不实现，就会报错。**

```
abstract class A {
  abstract foo:string;
  bar:string = '';
}

class B extends A {
  foo = 'b';
}
```

注意点：

- 抽象成员只能存在抽象类中，普通类中没有
- 抽象成员不能有具体实现的代码
- 抽象成员前也不能有private修饰符，否则无法在子类中实现该成员
- 一个子类最多只能继承一个抽象类

总之，抽象类的作用是，确保各种相关的子类都拥有跟基类相同的接口，可以看作是模块。其中抽象成员必须由子类实现，非抽象成员则表示基类已经实现的，直接由所有子类共享。

9. this问题

类的方法中的this，表示该方法当前所在的对象。

有些场合需要给出this类型，所以可以在参数列表的第一位，定义一个this类型，this参数的类型可以声明为各种对象。

```
// 编译前
function fn(
  this: SomeType,
  x: number
) {
  /* ... */
}

// 编译后
function fn(x) {
  /* ... */
}
```

ts提供了一个编译选项noImplicitThis，如果被打开，this的值被推断为any类型时就会报错。

在类的内部，this本身也可以当做类型使用，表示当前类的实例对象。

```
class Box {
  contents:string = '';

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

this类型不能用在静态成员中，否则会报错