

# 13.抽象类和接口

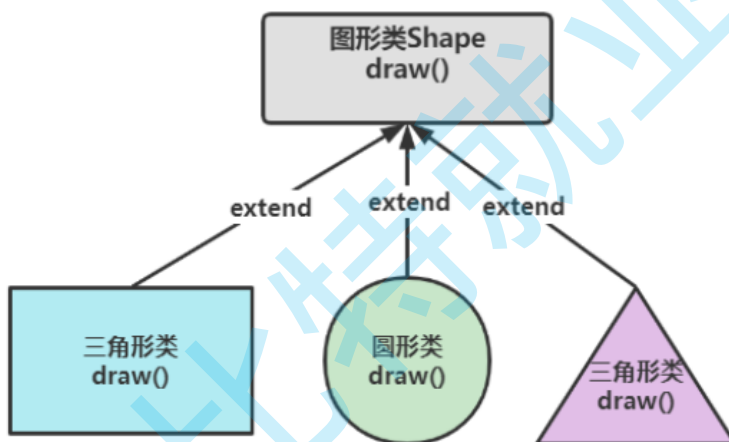
## 【本节目标】

1. 抽象类
2. 接口
3. Object类

## 1. 抽象类

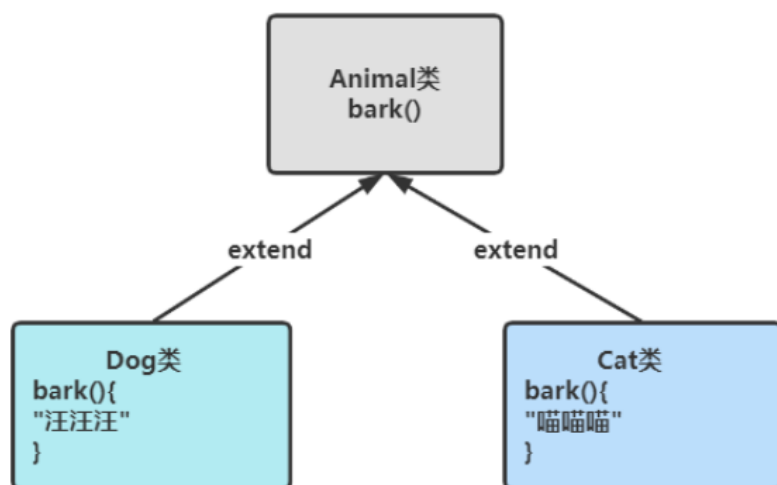
### 1.1 抽象类概念

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。比如：



说明：

1. 矩形、三角形、圆形都是图形，因此和Shape类的惯性应该是继承关系
2. 虽然图形图Shape中也存在draw的方法，但由于Shape类并不是具体的图形，因此其内部的draw方法实际是没有办法实现的
3. 由于Shape类没有办法描述一个具体的图形，导致其draw()方法无法具体实现，因此可以将Shape类设计为“抽象类”



说明:

1. Animal是动物类，每个动物都有叫的方法，但由于Animal不是一个具体的动物，因此其内部bark()方法无法具体实现
2. Dog是狗类，首先狗是动物，因此与Animal是继承关系，其次狗是一种具体的动物，狗叫：汪汪汪，其bark()可以实现
3. Cat是猫类，首先猫是动物，因此与Animal是继承关系，其次猫是一种具体的动物，猫叫：喵喵喵，其bark()可以实现
4. 因此：Animal可以设计为“抽象类”

在打印图形例子中，我们发现，父类 Shape 中的 draw 方法好像并没有什么实际工作，主要的绘制图形都是由 Shape 的各种子类的 draw 方法来完成的。像这种没有实际工作的方法，我们可以把它设计成一个**抽象方法(abstract method)**，包含抽象方法的类我们称为**抽象类(abstract class)**。

## 1.2 抽象类语法

在Java中，一个类如果被 `abstract` 修饰称为抽象类，抽象类中被 `abstract` 修饰的方法称为抽象方法，抽象方法不用给出具体的实现体。

```
1  // 抽象类：被abstract修饰的类
2  public abstract class Shape {
3      // 抽象方法：被abstract修饰的方法，没有方法体
4      abstract public void draw();
5      abstract void calcArea();
6
7      // 抽象类也是类，也可以增加普通方法和属性
8      public double getArea(){
9          return area;
10     }
11
12     protected double area;    // 面积
13 }
```

**注意：**抽象类也是类，内部可以包含普通方法和属性，甚至构造方法

## 1.3 抽象类特性

1. 抽象类不能直接实例化对象

```
1 Shape shape = new Shape();
2
3 // 编译出错
4 Error:(30, 23) java: Shape是抽象的; 无法实例化
```

## 2. 抽象方法不能是 private 的

```
1 abstract class Shape {
2     abstract private void draw();
3 }
4
5 // 编译出错
6 Error:(4, 27) java: 非法的修饰符组合: abstract和private
```

## 3. 抽象方法不能被final和static修饰, 因为抽象方法要被子类重写

```
1 public abstract class Shape {
2     abstract final void methodA();
3     abstract public static void methodB();
4 }
5
6 // 编译报错:
7 // Error:(20, 25) java: 非法的修饰符组合: abstract和final
8 // Error:(21, 33) java: 非法的修饰符组合: abstract和static
```

## 4. 抽象类必须被继承, 并且继承后子类要重写父类中的抽象方法, 否则子类也是抽象类, 必须要使用 `abstract` 修饰

```
1 // 矩形类
2 public class Rect extends Shape {
3     private double length;
4     private double width;
5
6     Rect(double length, double width){
7         this.length = length;
8         this.width = width;
9     }
10
11     public void draw(){
12         System.out.println("矩形: length= "+length+" width= " + width);
13     }
```

```

14
15     public void calcArea(){
16         area = length * width;
17     }
18 }
19
20 // 圆类:
21 public class Circle extends Shape{
22     private double r;
23     final private static double PI = 3.14;
24     public Circle(double r){
25         this.r = r;
26     }
27
28     public void draw(){
29         System.out.println("圆: r = "+r);
30     }
31
32     public void calcArea(){
33         area = PI * r * r;
34     }
35 }
36
37 // 三角形类:
38 public abstract class Triangle extends Shape {
39     private double a;
40     private double b;
41     private double c;
42
43     @Override
44     public void draw() {
45         System.out.println("三角形: a = "+a + " b = "+b+" c = "+c);
46     }
47
48     // 三角形: 直角三角形、等腰三角形等, 还可以继续细化
49     // @Override
50     // double calcArea();    // 编译失败: 要么实现该抽象方法, 要么将三角形设计为抽象类
51 }

```

5. 抽象类中不一定包含抽象方法, 但是有抽象方法的类一定是抽象类

6. 抽象类中可以有构造方法, 供子类创建对象时, 初始化父类的成员变量

## 1.4 抽象类的作用

抽象类本身不能被实例化, 要想使用, 只能创建该抽象类的子类. 然后让子类重写抽象类中的抽象方法.

有些同学可能会说了, 普通的类也可以被继承呀, 普通的方法也可以被重写呀, 为啥非得用抽象类和抽象方法呢?

确实如此. 但是使用抽象类相当于多了一重编译器的校验.

使用抽象类的场景就如上面的代码, 实际工作不应该由父类完成, 而应由子类完成. 那么此时如果不小心误用成父类了, 使用普通类编译器是不会报错的. 但是父类是抽象类就会在实例化的时候提示错误, 让我们尽早发现问题.

很多语法存在的意义都是为了 "预防出错", 例如我们曾经用过的 `final` 也是类似. 创建的变量用户不去修改, 不就相当于常量嘛? 但是加上 `final` 能够在不小心误修改的时候, 让编译器及时提醒我们.

充分利用编译器的校验, 在实际开发中是非常有意义的.

## 2. 接口

### 2.1 接口的概念

在现实生活中, 接口的例子比比皆是, 比如: 笔记本上的USB口, 电源插座等



电脑的USB口上, 可以插: U盘、鼠标、键盘...所有符合USB协议的设备

电源插座插孔上, 可以插: 电脑、电视机、电饭煲...所有符合规范的设备

通过上述例子可以看出: **接口就是公共的行为规范标准, 大家在实现时, 只要符合规范标准, 就可以通用。**

**在Java中, 接口可以看成是: 多个类的公共规范, 是一种引用数据类型。**

### 2.2 语法规则

接口的定义格式与定义类的格式基本相同, 将`class`关键字换成 `interface` 关键字, 就定义了一个接口。

```
1 public interface 接口名称{  
2     // 抽象方法
```

```

3      public abstract void method1();    // public abstract 是固定搭配, 可以不写
4      public void method2();
5      abstract void method3();
6      void method4();
7
8      // 注意: 在接口中上述写法都是抽象方法, 跟推荐方式4, 代码更简洁
9  }

```

提示:

1. 创建接口时, 接口的命名一般以大写字母 **I** 开头.
2. 接口的命名一般使用 "形容词" 词性的单词.
3. 阿里编码规范中约定, 接口中的方法和属性不要加任何修饰符号, 保持代码的简洁性.

## 2.3 接口使用

接口不能直接使用, 必须要有一个"实现类"来"实现"该接口, 实现接口中的所有抽象方法。

```

1  public class 类名称 implements 接口名称{
2      // ...
3  }

```

注意: 子类 and 父类之间是 extends 继承关系, 类与接口之间是 implements 实现关系。

- 1 实现笔记本电脑使用USB鼠标、USB键盘的例子
- 2 1. USB接口: 包含打开设备、关闭设备功能
- 3 2. 笔记本类: 包含开机功能、关机功能、使用USB设备功能
- 4 3. 鼠标类: 实现USB接口, 并具备点击功能
- 5 4. 键盘类: 实现USB接口, 并具备输入功能

```

1  // USB接口
2  public interface USB {
3      void openDevice();
4      void closeDevice();
5  }
6
7  // 鼠标类, 实现USB接口
8  public class Mouse implements USB {
9      @Override
10     public void openDevice() {
11         System.out.println("打开鼠标");

```

```
12     }
13
14     @Override
15     public void closeDevice() {
16         System.out.println("关闭鼠标");
17     }
18
19     public void click(){
20         System.out.println("鼠标点击");
21     }
22 }
23
24 // 键盘类, 实现USB接口
25 public class KeyBoard implements USB {
26     @Override
27     public void openDevice() {
28         System.out.println("打开键盘");
29     }
30
31     @Override
32     public void closeDevice() {
33         System.out.println("关闭键盘");
34     }
35
36     public void inPut(){
37         System.out.println("键盘输入");
38     }
39 }
40
41 // 笔记本类: 使用USB设备
42 public class Computer {
43     public void powerOn(){
44         System.out.println("打开笔记本电脑");
45     }
46
47     public void powerOff(){
48         System.out.println("关闭笔记本电脑");
49     }
50
51     public void useDevice(USB usb){
52         usb.openDevice();
53         if(usb instanceof Mouse){
54             Mouse mouse = (Mouse)usb;
55             mouse.click();
56         }else if(usb instanceof KeyBoard){
57             KeyBoard keyBoard = (KeyBoard)usb;
58             keyBoard.inPut();
```

```

59         }
60         usb.closeDevice();
61     }
62 }
63
64 // 测试类:
65 public class TestUSB {
66     public static void main(String[] args) {
67         Computer computer = new Computer();
68         computer.powerOn();
69
70         // 使用鼠标设备
71         computer.useDevice(new Mouse());
72
73         // 使用键盘设备
74         computer.useDevice(new KeyBoard());
75
76         computer.powerOff();
77     }
78 }

```

## 2.4 接口特性

1. 接口类型是一种引用类型，但是不能直接new接口的对象

```

1  public class TestUSB {
2      public static void main(String[] args) {
3          USB usb = new USB();
4      }
5  }
6
7  // Error:(10, 19) java: day20210915.USB是抽象的; 无法实例化

```

2. 接口中每一个方法都是public的抽象方法, 即接口中的方法会被隐式的指定为 **public abstract** (只能是 public abstract, 其他修饰符都会报错)

```

1  public interface USB {
2      // Error:(4, 18) java: 此处不允许使用修饰符private
3      private void openDevice();
4      void closeDevice();
5  }

```



### 3. 接口中的方法是不能在接口中实现的，只能由实现接口的类来实现

```
1 public interface USB {
2     void openDevice();
3
4     // 编译失败：因为接口中的方式默认为抽象方法
4     // Error:(5, 23) java: 接口抽象方法不能带有主体
6     void closeDevice(){
7         System.out.println("关闭USB设备");
8     }
9 }
```

### 4. 重写接口中方法时，不能使用默认访问权限

```
1 public interface USB {
2     void openDevice(); // 默认是public的
3     void closeDevice(); // 默认是public的
4 }
5
6 public class Mouse implements USB {
7     @Override
8     void openDevice() {
9         System.out.println("打开鼠标");
10    }
11
12    // ...
13 }
14
15 // 编译报错，重写USB中openDevice方法时，不能使用默认修饰符
16 // 正在尝试分配更低的访问权限；以前为public
```

### 5. 接口中可以含有变量，但是接口中的变量会被隐式的指定为 **public static final** 变量

```
1 public interface USB {
2     double brand = 3.0; // 默认被: final public static修饰
3     void openDevice();
4     void closeDevice();
5 }
6
7 public class TestUSB {
8     public static void main(String[] args) {
9         System.out.println(USB.brand); // 可以直接通过接口名访问，说明是静态的
```

```

10
11          // 编译报错: Error:(12, 12) java: 无法为最终变量brand分配值
12      USB.brand = 2.0;          // 说明brand具有final属性
13  }
14  }

```

## 6. 接口中不能有静态代码块和构造方法

```

1  public interface USB {
2      // 编译失败
3      public USB(){
4
5      }
6
7      {}    // 编译失败
8
9      void openDevice();
10     void closeDevice();
11 }

```

7. 接口虽然不是类，但是接口编译完成后字节码文件的后缀格式也是.class

8. 如果类没有实现接口中的所有抽象方法，则类必须设置为抽象类

9. jdk8中：接口中还可以包含default方法。

## 2.5 实现多个接口

在Java中，类和类之间是单继承的，一个类只能有一个父类，即**Java中不支持多继承**，但是一个类可以实现多个接口。下面通过类来表示一组动物。

```

1  class Animal {
2      protected String name;
3
4      public Animal(String name) {
5          this.name = name;
6      }
7  }

```

另外我们再提供一组接口，分别表示 "会飞的", "会跑的", "会游泳的".

```

1  interface IFlying {

```

```
2     void fly();
3 }
4
5 interface IRunning {
6     void run();
7 }
8
9 interface ISwimming {
10     void swim();
11 }
```

接下来我们创建几个具体的动物

猫,是会跑的.

```
1 class Cat extends Animal implements IRunning {
2     public Cat(String name) {
3         super(name);
4     }
5
6     @Override
7     public void run() {
8         System.out.println(this.name + "正在用四条腿跑");
9     }
10 }
```

鱼,是会游的.

```
1 class Fish extends Animal implements ISwimming {
2     public Fish(String name) {
3         super(name);
4     }
5
6     @Override
7     public void swim() {
8         System.out.println(this.name + "正在用尾巴游泳");
9     }
10 }
```

青蛙,既能跑,又能游(两栖动物)

```
1 class Frog extends Animal implements IRunning, ISwimming {
```

```

2     public Frog(String name) {
3         super(name);
4     }
5
6     @Override
7     public void run() {
8         System.out.println(this.name + "正在往前跳");
9     }
10
11    @Override
12    public void swim() {
13        System.out.println(this.name + "正在蹬腿游泳");
14    }
15 }

```

**注意：一个类实现多个接口时，每个接口中的抽象方法都要实现，否则类必须设置为抽象类。**

提示, IDEA 中使用 ctrl + i 快速实现接口

还有一种神奇的动物, 水陆空三栖, 叫做 "鸭子"

```

1  class Duck extends Animal implements IRunning, ISwimming, IFlying {
2      public Duck(String name) {
3          super(name);
4      }
5
6      @Override
7      public void fly() {
8          System.out.println(this.name + "正在用翅膀飞");
9      }
10
11
12     @Override
13     public void run() {
14         System.out.println(this.name + "正在用两条腿跑");
15     }
16
17     @Override
18     public void swim() {
19         System.out.println(this.name + "正在漂在水上");
20     }
21 }

```

上面的代码展示了 Java 面向对象编程中最常见的用法: 一个类继承一个父类, 同时实现多种接口。

继承表达的含义是 `is - a` 语义, 而接口表达的含义是 具有 `xxx` 特性 .

猫是一种动物, 具有会跑的特性.

青蛙也是一种动物, 既能跑, 也能游泳

鸭子也是一种动物, 既能跑, 也能游, 还能飞

这样设计有什么好处呢? 时刻牢记多态的好处, 让程序猿**忘记类型**. 有了接口之后, 类的使用者就不必关注具体类型, 而只关注某个类是否具备某种能力.

例如, 现在实现一个方法, 叫 "散步"

```
1 public static void walk(IRunning running) {
2     System.out.println("我带着伙伴去散步");
3     running.run();
4 }
```

在这个 `walk` 方法内部, 我们并不关注到底是哪种动物, 只要参数是会跑的, 就行

```
1 Cat cat = new Cat("小猫");
2 walk(cat);
3
4 Frog frog = new Frog("小青蛙");
5 walk(frog);
6
7 // 执行结果
8 我带着伙伴去散步
9 小猫正在用四条腿跑
10 我带着伙伴去散步
11 小青蛙正在往前跳
```

甚至参数可以不是 "动物", 只要会跑!

```
1 class Robot implements IRunning {
2     private String name;
3     public Robot(String name) {
4         this.name = name;
5     }
6
7     @Override
8     public void run() {
9         System.out.println(this.name + "正在用轮子跑");
10    }
```

```
11 }
12
13 Robot robot = new Robot("机器人");
14 walk(robot);
15
16 // 执行结果
17 机器人正在用轮子跑
```

## 2.6 接口间的继承

在Java中，类和类之间是单继承的，一个类可以实现多个接口，接口与接口之间可以多继承。即：用接口可以达到多继承的目的。

接口可以继承一个接口, 达到复用的效果. 使用 extends 关键字.

```
1 interface IRunning {
2     void run();
3 }
4
5 interface ISwimming {
6     void swim();
7 }
8
9 // 两栖的动物，既能跑，也能游
10 interface IAmphibious extends IRunning, ISwimming {
11
12 }
13
14 class Frog implements IAmphibious {
15     ...
16 }
```

通过接口继承创建一个新的接口 IAmphibious 表示 "两栖的". 此时实现接口创建的 Frog 类, 就继续要实现 run 方法, 也需要实现 swim 方法.

接口间的继承相当于把多个接口合并在一起.

## 2.7 接口使用实例

对象之间进行大小关系比较

```
1 class Student {
2     public String name;
3     public int score;
4 }
```

```

5     public Student(String name, int score) {
6         this.name = name;
7         this.score = score;
8     }
9
10    @Override
11    public String toString() {
12        return "[" + this.name + ":" + this.score + "]";
13    }
14 }

```

```

1  public class Test {
2      public static void main(String[] args) {
3          Student s1 = new Student("zhangsan",10);
4          Student s2 = new Student("lisi",20);
5          System.out.println(s1 > s2 );
6      }
7  }

```

此时程序会编译报错，并没有指定根据分数还是什么进行大小比较。所以，应该指定以什么样的方式进行比较。

不太好的点是，只要按照该方式比较后，不灵活。无法按照其他方式进行比较

### 方式一：使用Comparable 接口

让我们的 Student 类实现 **Comparable** 接口, 并实现其中的 compareTo 方法

```

1  class Student implements Comparable {
2      private String name;
3      private int score;
4
5      public Student(String name, int score) {
6          this.name = name;
7          this.score = score;
8      }
9
10     @Override
11     public String toString() {
12         return "[" + this.name + ":" + this.score + "]";
13     }
14
15     @Override
16     public int compareTo(Object o) {

```

```

17         Student s = (Student)o;
18         if (this.score > s.score) {
19             return -1;
20         } else if (this.score < s.score) {
21             return 1;
22         } else {
23             return 0;
24         }
25     }
26 }

```

```

1  public class Test {
2      public static void main(String[] args) {
3          Student s1 = new Student("zhangsan",10);
4          Student s2 = new Student("lisi",20);
5          System.out.println(s1.compareTo(s2) );
6      }
7  }

```

如果s1大于s2那么返回大于0的数字，如果相同返回0，否则返回小于0的数字。

## 方式二：使用Comparator接口

```

1  class Student {
2      public String name;
3      public int score;
4
5      public Student(String name, int score) {
6          this.name = name;
7          this.score = score;
8      }
9
10     @Override
11     public String toString() {
12         return "[" + this.name + ":" + this.score + "]";
13     }
14 }
15
16 class ScoreComparator implements Comparator<Student> {
17
18     @Override
19     public int compare(Student o1, Student o2) {
20         return o1.score-o2.score;
21     }

```



```

22     }
23
24     class NameComparator implements Comparator<Student> {
25
26         @Override
27         public int compare(Student o1, Student o2) {
28             return o1.name.compareTo(o2.name);
29         }
30     }

```

```

1     public class Test {
2         public static void main(String[] args) {
3             Student s1 = new Student("zhangsan",10);
4             Student s2 = new Student("lisi",20);
5             //根据分数进行比较
6             ScoreComparator scoreComparator = new ScoreComparator();
7             System.out.println(scoreComparator.compare(s1, s2));
8
9             //根据姓名进行比较
10            NameComparator nameComparator = new NameComparator();
11            System.out.println(nameComparator.compare(s1, s2));
12
13        }
14    }

```

## 2.8 Cloneable 接口和深拷贝

Java 中内置了一些很有用的接口, Cloneable 就是其中之一。

Object 类中存在一个 clone 方法, 调用这个方法可以创建一个对象的 "拷贝". 但是要想合法调用 clone 方法, 必须要先实现 Cloneable 接口, 否则就会抛出 CloneNotSupportedException 异常。

```

1     class Animal implements Cloneable {
2         private String name;
3
4         @Override
5         public Animal clone() {
6             Animal o = null;
7             try {
8                 o = (Animal)super.clone();
9             } catch (CloneNotSupportedException e) {
10                 e.printStackTrace();
11             }
12             return o;

```

```

13     }
14 }
15
16 public class Test {
17     public static void main(String[] args) {
18         Animal animal = new Animal();
19         Animal animal2 = animal.clone();
20         System.out.println(animal == animal2);
21     }
22 }
23
24 // 输出结果
25 // false

```

## 浅拷贝 VS 深拷贝

Cloneable 拷贝出的对象是一份 "浅拷贝"

观察以下代码:

```

1  class Money {
2      public double m = 99.99;
3  }
4
5  class Person implements Cloneable{
6      public Money money = new Money();
7
8      @Override
9      protected Object clone() throws CloneNotSupportedException {
10         return super.clone();
11     }
12 }
13
14 public class TestDemo3 {
15     public static void main(String[] args) throws
CloneNotSupportedException {
16         Person person1 = new Person();
17         Person person2 = (Person) person1.clone();
18         System.out.println("通过person2修改前的结果");
19         System.out.println(person1.money.m);
20         System.out.println(person2.money.m);
21         person2.money.m = 13.6;
22         System.out.println("通过person2修改后的结果");
23         System.out.println(person1.money.m);
24         System.out.println(person2.money.m);
25     }

```

```
26  }
27
28  // 执行结果
29  通过person2修改前的结果
30  99.99
31  99.99
32  通过person2修改后的结果
33  13.6
34  13.6
```

如上代码，我们可以看到，通过clone，我们只是拷贝了Person对象。但是Person对象中的Money对象，并没有拷贝。通过person2这个引用修改了m的值后，person1这个引用访问m的时候，值也发生了改变。这里就是发生了浅拷贝。那么同学们想一下如何实现深拷贝呢？

## 2.9 抽象类和接口的区别

抽象类和接口都是 Java 中多态的常见使用方式. 都需要重点掌握. 同时又要认清两者的区别(重要!!! 常见面试题).

**核心区别:** 抽象类中可以包含普通方法和普通字段, 这样的普通方法和字段可以被子类直接使用(不必重写), 而接口中不能包含普通方法, 子类必须重写所有的抽象方法.

如之前写的 Animal 例子. 此处的 Animal 中包含一个 name 这样的属性, 这个属性在任何子类中都是存在的. 因此此处的 Animal 只能作为一个抽象类, 而不应该成为一个接口.

```
1  class Animal {
2      protected String name;
3
4      public Animal(String name) {
5          this.name = name;
6      }
7  }
```

再次提醒:

抽象类存在的意义是为了让编译器更好的校验, 像 Animal 这样的类我们并不会直接使用, 而是使用它的子类. 万一不小心创建了 Animal 的实例, 编译器会及时提醒我们.

No	区别	抽象类(abstract)	接口(interface)
1	结构组成	普通类+抽象方法	抽象方法+全局常量
2	权限	各种权限	public
3	子类使用	使用extends关键字继承抽象类	使用implements关键字实现接口
4	关系	一个抽象类可以实现若干接口	接口不能继承抽象类，但是接口可以使用extends关键字继承多个父接口
5	子类限制	一个子类只能继承一个抽象类	一个子类可以实现多个接口

### 3. 内部类

当一个事物的内部，还有一个部分需要一个完整的结构进行描述，而这个内部的完整的结构又只为外部事物提供服务，那么这个内部的完整结构最好使用内部类。在 Java 中，**可以将一个类定义在另一个类或者一个方法的内部，前者称为内部类，后者称为外部类。**内部类也是封装的一种体现。

```
1 public class OutClass {
2     class InnerClass{
3
4     }
5 }
6
7 // OutClass是外部类
8 // InnerClass是内部类
```

【注意事项】

- 1. 定义在class 类名{}花括号外部的，即使是在一个文件里，都不能称为内部类

```
1 public class A{
2
3 }
4
5 class B{
6
7 }
8
9 // A 和 B是两个独立的类，彼此之前没有关系
```

## 2. 内部类和外部类共用同一个java源文件，但是经过编译之后，内部类会形成单独的字节码文件

### 3.1 内部类的分类

```
1  public class OutClass {
2      // 成员位置定义：未被static修饰 ----> 实例内部类
3      public class InnerClass1{
4      }
5
6      // 成员位置定义：被static修饰 ----> 静态内部类
7      static class InnerClass2{
8
9      }
10
11
12     public void method(){
13         // 方法中也可以定义内部类 ----> 局部内部类：几乎不用
14         class InnerClass5{
15
16         }
17     }
18 }
```

根据内部类定义的位置不同，一般可以分为以下几种形式：

1. 成员内部类(普通内部类：未被static修饰的成员内部类 和 静态内部类：被static修饰的成员内部类)
2. 局部内部类(不谈修饰符)、匿名内部类

注意：内部类其实日常开发中使用并不是非常多，大家在看一些库中的代码时候可能会遇到的比较多，日常开始中使用最多的是匿名内部类。

### 3.2 静态内部类

被static修饰的内部成员类称为静态内部类

```
1  public class OutClass {
2      public int a;
3      public static int b;
4
5
6      // 静态内部类：被static修饰的成员内部类
7      static class InnerClass{
8          public void methodInner(){
```

```

9          // 在内部类中只能访问外部类的静态成员
10         // a = 100;          // 编译失败，因为a不是类成员变量
11         b = 200;
12     }
13 }
14
15     public static void main(String[] args) {
16         // 静态内部类对象创建 & 成员访问
17         OutClass.InnerClass innerClass = new OutClass.InnerClass();
18         innerClass.methodInner();
19     }
20 }

```

### 【注意事项】

#### 1. 在静态内部类中只能访问外部类中的静态成员

如果确实想访问，我们该如何做？

#### 2. 创建静态内部类对象时，不需要先创建外部类对象

### 3.3 实例内部类

即未被static修饰的成员内部类。

```

1     public class OutClass {
2         public int a;
3         public static int b;
4         public int c;
5
6
7         // 实例内部类：未被static修饰
8         class InnerClass{
9             int c;
10            public void methodInner(){
11                // 在实例内部类中可以直接访问外部类中：任意访问限定符修饰的成员
12                a = 100;
13                b = 200;
14                // 如果外部类和实例内部类中具有相同名称成员时，优先访问的是内部类自己的
15                c = 300;
16                System.out.println(c);
17
18                // 如果要访问外部类同名成员时候，必须：外部类名称.this.同名成员名字
19                OutClass.this.c = 400;
20                System.out.println(OutClass.this.c);
21            }
22        }

```

```

23
24     public static void main(String[] args) {
25         // 外部类: 对象创建 以及 成员访问
26         OutClass outClass = new OutClass();
27         System.out.println(outClass.a);
28         System.out.println(OutClass.b);
29         System.out.println(outClass.c);
30
31         System.out.println("=====实例内部类的访问=====");
32         // 要访问实例内部类中成员, 必须要创建实例内部类的对象
33         // 而普通内部类定义与外部类成员定义位置相同, 因此创建实例内部类对象时必须借助外
    部类
34
35         // 创建实例内部类对象
36         OutClass.InnerClass innerClass1 = new OutClass().new InnerClass();
37
38         // 上述语法比较怪异, 也可以先将外部类对象先创建出来, 然后再创建实例内部类对象
39         OutClass.InnerClass innerClass2 = outClass.new InnerClass();
40         innerClass2.methodInner();
41     }
42 }

```

### 【注意事项】

1. 外部类中的任何成员都可以在实例内部类方法中直接访问
2. 实例内部类所处的位置与外部类成员位置相同, 因此也受public、private等访问限定符的约束
3. 在实例内部类方法中访问同名的成员时, 优先访问自己的, 如果要访问外部类同名的成员, 必须: 外部类名称.this.同名成员 来访问
4. 实例内部类对象必须在先有外部类对象前提下才能创建
5. 实例内部类的非静态方法中包含了一个指向外部类对象的引用
6. 外部类中, 不能直接访问实例内部类中的成员, 如果要访问必须先要创建内部类的对象。

## 3.4 局部内部类

定义在外部类的方法体或者{}中, 该种内部类只能在其定义的位置使用, 一般使用的非常少, 此处简单了解下语法格式。

```

1     public class OutClass {
2         int a = 10;
3         public void method(){
4             int b = 10;
5
6             // 局部内部类: 定义在方法体内部

```

```

7          // 不能被public、static等访问限定符修饰
8      class InnerClass{
9          public void methodInnerClass(){
10              System.out.println(a);
11              System.out.println(b);
12          }
13      }
14
15          // 只能在该方法体内部使用，其他位置都不能用
16      InnerClass innerClass = new InnerClass();
17      innerClass.methodInnerClass();
18  }
19
20      public static void main(String[] args) {
21          // OutClass.InnerClass innerClass = null; 编译失败
22      }
23  }

```

### 【注意事项】

1. 局部内部类只能在所定义的方法体内部使用
2. 不能被public、static等修饰符修饰
3. 编译器也有自己独立的字节码文件，命名格式：外部类名字\$数字内部类名字.class
4. 几乎不会使用

## 3.5 匿名内部类

匿名内部类是Java中一种特殊的嵌套类，它没有类名，在声明的同时完成实例化。匿名内部类通常用于创建只需使用一次的类。

语法形式：

```

1  new SuperType(constructor-arguments) {      // 类体 };

```

SuperType 可以是接口、抽象类或具体类。

示例代码：

```

1  interface Greeting {
2      void greet();
3  }
4
5  public class Test {

```



```

6      public static void main(String[] args) {
7          Greeting greeting = new Greeting() {
8              @Override
9              public void greet() {
10                 System.out.println("Hello!");
11             }
12         };
13
14         greeting.greet();
15     }
16 }

```

- 匿名内部类当中可以定义和正常类一样的成员变量，但是和正常类一样都不能直接包含执行语句。

```

1
2  interface Greeting {
3      void greet();
4  }
5
6  public class Derived extends Base {
7      public static void main(String[] args) {
8          Greeting greeting = new Greeting() {
9              public static int a = 10;
10             a = 100; //error
11             @Override
12             public void greet() {
13                 System.out.println("Hello!" + a);
14             }
15         };
16
17         greeting.greet();
18     }
19 }

```

## 4. Object类

Object是Java默认提供的一个类。Java里面除了Object类，所有的类都是存在继承关系的。默认会继承Object父类。即所有类的对象都可以使用Object的引用进行接收。

**范例：使用Object接收所有类的对象**

```

1  class Person{}
2  class Student{}

```

```

3  public class Test {
4      public static void main(String[] args) {
5          function(new Person());
6          function(new Student());
7      }
8      public static void function(Object obj) {
9          System.out.println(obj);
10     }
11 }
12 //执行结果:
13 Person@1b6d3586
14 Student@4554617c

```

所以在开发之中，Object类是参数的最高统一类型。但是Object类也存在有定义好的一些方法。如下：

Modifier and Type	Method and Description
protected Object	clone() 创建并返回此对象的副本。
boolean	equals(Object obj) 指示一些其他对象是否等于此。
protected void	finalize() 当垃圾收集器确定不再对该对象的引用时，垃圾收集器在对象上调用该对象。
类<?>	getClass() 返回此 Object 的运行时常类。
int	hashCode() 返回对象的哈希码值。
void	notify() 唤醒正在等待对象监视器的单个线程。
void	notifyAll() 唤醒正在等待对象监视器的所有线程。
String	toString() 返回对象的字符串表示形式。
void	wait() 导致当前线程等待，直到另一个线程调用该对象的 notify() 方法或 notifyAll() 方法。
void	wait(long timeout) 导致当前线程等待，直到另一个线程调用 notify() 方法或该对象的 notifyAll() 方法，或者指定的时间已过。
void	wait(long timeout, int nanos) 导致当前线程等待，直到另一个线程调用该对象的 notify() 方法或 notifyAll() 方法，或者某些其他线程中断当前线程，或一定量的实时时间。

对于整个Object类中的方法需要实现全部掌握。

本小节当中，我们主要来熟悉这几个方法：toString()方法，equals()方法，hashCode()方法

## 4.1 获取对象信息

如果要打印对象中的内容，可以直接重写Object类中的toString()方法，之前已经讲过了，此处不再赘赘。

```

1  // Object类中的toString()方法实现:
2  public String toString() {
3      return getClass().getName() + "@" + Integer.toHexString(hashCode());
4  }

```

## 4.2 对象比较equals方法

在Java中，==进行比较时：

- a.如果==左右两侧是基本类型变量，比较的是变量中值是否相同
- b.如果==左右两侧是引用类型变量，比较的是引用变量地址是否相同
- c.如果要比较对象中内容，必须重写Object中的equals方法，因为equals方法默认也是按照地址比较的：

```
1  // Object类中的equals方法
2  public boolean equals(Object obj) {
3      return (this == obj);    // 使用引用中的地址直接来进行比较
4  }
```

```
1  class Person{
2      private String name ;
3      private int age ;
4      public Person(String name, int age) {
5          this.age = age ;
6          this.name = name ;
7      }
8  }
9  public class Test {
10     public static void main(String[] args) {
11         Person p1 = new Person("gaobo", 20) ;
12         Person p2 = new Person("gaobo", 20) ;
13         int a = 10;
14         int b = 10;
15         System.out.println(a == b);           // 输出true
16         System.out.println(p1 == p2);         // 输出false
17         System.out.println(p1.equals(p2));    // 输出false
18     }
19 }
```

Person类重写equals方法后，然后比较：

```
1  class Person{
2      ...
3      @Override
4      public boolean equals(Object obj) {
```

```

5         if (obj == null) {
6             return false ;
7         }
8         if(this == obj) {
9             return true ;
10        }
11        // 不是Person类对象
12        if (!(obj instanceof Person)) {
13            return false ;
14        }
15
16        Person person = (Person) obj ; // 向下转型，比较属性值
17        return this.name.equals(person.name) && this.age==person.age ;
18    }
19 }

```

**结论：**比较对象中内容是否相同的时候，一定要重写equals方法

### 4.3 hashCode方法

回忆刚刚的toString方法的源码：

```

1    public String toString() {
2        return getClass().getName() + "@" + Integer.toHexString(hashCode());
3    }

```

我们看到了hashCode()这个方法，他帮我算了一个具体的**对象位置**，这里面涉及数据结构，但是我们还没学数据结构，没法讲述，所以我们只能说它是个内存地址。然后调用Integer.toHexString()方法，将这个地址以16进制输出。

**hashCode方法源码：**

```

1    public native int hashCode();

```

该方法是一个native方法，底层是由C/C++代码写的。我们看不到。

我们认为两个名字相同，年龄相同的对象，将存储在同一个位置，如果不重写hashCode()方法，我们可以来看示例代码：

```

1    class Person {
2        public String name;
3        public int age;

```

```

4     public Person(String name, int age) {
5         this.name = name;
6         this.age = age;
7     }
8 }
9 public class TestDemo4 {
10     public static void main(String[] args) {
11         Person per1 = new Person("gaobo", 20) ;
12         Person per2 = new Person("gaobo", 20) ;
13         System.out.println(per1.hashCode());
14         System.out.println(per2.hashCode());
15     }
16 }
17 //执行结果
18 460141958
19 1163157884

```

**注意事项：两个对象的hash值不一样。**

像重写equals方法一样，我们也可以重写hashCode()方法。此时我们再来看看。

```

1 class Person {
2     public String name;
3     public int age;
4     public Person(String name, int age) {
5         this.name = name;
6         this.age = age;
7     }
8
9     @Override
10    public int hashCode() {
11        return Objects.hash(name, age);
12    }
13 }
14 public class TestDemo4 {
15     public static void main(String[] args) {
16         Person per1 = new Person("gaobo", 20) ;
17         Person per2 = new Person("gaobo", 20) ;
18         System.out.println(per1.hashCode());
19         System.out.println(per2.hashCode());
20     }
21 }
22 //执行结果
23 460141958
24 460141958

```

注意事项：哈希值一样。

结论：

- 1、hashCode方法用来确定对象在内存中存储的位置是否相同
- 2、事实上hashCode() 在散列表中才有用，在其它情况下没用。在散列表中hashCode() 的作用是获取对象的散列码，进而确定该对象在散列表中的位置。

## 5. 小试牛刀

1. 关于Java抽象类，以下哪个陈述是错误的？
  - A. 抽象类可以包含非抽象方法
  - B. 抽象类可以被实例化
  - C. 抽象类可以有构造方法
  - D. 一个类可以继承多个抽象类
2. 关于Java接口，以下哪个说法是正确的？
  - A. 接口中的所有方法都必须是抽象的
  - B. 接口可以包含静态方法
  - C. 接口可以有私有方法
  - D. 一个类只能实现一个接口
3. 以下关于内部类的描述，哪个是错误的？
  - A. 静态内部类可以访问外部类的静态成员
  - B. 非静态内部类可以直接访问外部类的所有成员
  - C. 局部内部类可以访问所在方法的final局部变量
  - D. 匿名内部类可以继承其他类
4. 关于抽象类和接口的区别，以下哪个说法是正确的？
  - A. 抽象类可以有构造方法，而接口不能
  - B. 一个类可以实现多个抽象类，但只能实现一个接口
  - C. 接口中的所有方法都必须是public的，而抽象类没有这个限制
  - D. 抽象类中的方法必须都是抽象的

答案解析：

1. 1. B D
2. 2. B

- 3 3. D 匿名内部类只能继承一个类或实现一个接口，不能同时继承类和实现接口
- 4 4. A C

---

完

比特就业课