《面向对象》之继承（下）

笔记本： JAVA
创建时间： 2018/8/2 9:55 **更新时间：** 2018/8/16 22:03
作者： Debao Xu

抽象类和抽象方法

描述一个事物，却没有足够的信息，这时就将这个事物称为抽象事物。例如，“水果”就是一个抽象事物，因为找不到足够的信息能够把水果是什么说清楚。面对抽象的事物，虽然不具体，但是可以将要处理的复杂的事物简单化，因为不用面对具体的事物。

特点：

1、抽象方法一定要定义在抽象类中，并且抽象方法和抽象类都需要用关键字abstract来修饰
2、抽象类不能实例化，即不能用new关键字来创建对象
3、只有子类覆盖了所有的抽象方法之后，子类才能被具体化（创建对象），如果没有覆盖所有的抽象方法，那么子类还是一个抽象类

抽象类也是不断地向上抽取而来的，只是越向上抽取越不具体，因为我们抽取了方法的声明，但是没有抽取方法的实现（具体的方法内容），即函数的主体不明确，但是功能声明是明确且存在的

问题：

1、抽象类中有构造函数吗？
有，抽象类的构造函数虽然不能给抽象类的对象实例化（因为抽象类不能创建对象），但是抽象类有子类，它的构造函数可以对创建出的对象实例化

抽象类和一般类的异同点：
相同：都是用来描述事物，都可以进行属性和行为的描述
不同：抽象类描述事物的信息不具体，而一般类描述事物的信息具体
代码的不同：抽象类中可以定义抽象方法，而一般类不行；抽象类不可以实例化，而一般类可以

2、抽象类一定是个父类吗？
是，必须需要子类覆盖抽象方法后，才可以实例化，然后使用这些方法

3、抽象类中可以定义抽象方法吗？
可以的，仅仅是为了让该类不能创建对象

4、抽象关键字abstract和哪些关键字不能共存呢？
final：final修饰的类不能有子类，而abstract修饰的类一定有子类；
private：因为数据私有化之后不能被访问
static：static依靠类名来调用

练习

需求：程序员有姓名、工号、薪水、工作；项目经理有姓名、工号、薪水、奖金、工作
要求：对需求进行数据建模

分析：
程序员
属性：姓名、工号、薪水
行为：工作

项目经理
属性：姓名、工号、薪水、奖金
行为：工作

两者不存在所属关系，但是有共性内容，可以向上抽取，这二者的共性就是“雇员”，
雇员
属性：姓名、工号、薪水
行为：工作

代码如下：

```
//雇员（抽象类）
abstract class Employee
{
	private String name;
	private String id;
	private int payment;
	Employee(String name,String id,int payment)// 抽象类构造函数
	{
		this.name = name;
		this.id = id;
		this.payment = payment;
	}
	public abstract void work(); // 注意，抽象类的方法没有“主体”
}
class Programmer extends Employee
{
	Programmer(String name,String id,int payment)// 子类构造函数
	{
		super(name,id,payment); // super访问父类的构造函数
	}
	public void work()
	{
		System.out.println("Coding!");
	}
}
class Manager extends Employee
{
	private double bonus; // Manager的特有属性
	Manager(String name,String id,int payment,double bonus)
	{
		super(name,id,payment);
		this.bonus = bonus;
	}
	public void work()
	{
		System.out.println("Management");
	}
}
public class Company
{
	public static void main(String[] args)
	{
		Programmer P1 = new Programmer("Rex","01",7000);
		Manager M1 = new Manager("Debao","02",10000,3000);
		P1.work();
		M1.work();
	}
}
```

接口（interface）

如果抽象类中所有的方法都是抽象的，这时，可以把抽象类用另一种形式来表示，即接口。
定义接口，需要用关键字 interface，如下：

```
interface Inter
{
    public static final NUM = 4;  //全局常量
    public abstract void show1(); //抽象方法
    public abstract void show2(); //抽象方法
}
```

接口中的成员和class中成员的定义的不同之处在于;
1、接口中常见的成员有两种：全局常量 、抽象方法；2、而且在接口中定义的数据都有固定的修饰符，成员都是用public修饰的（因为“接口”是一种“向外”提供的机制，因此必须要有足够大的访问权限，因此用public修饰）

接口的特点：

1、接口不可以实例化，因为其中包含了抽象类方法
2、需要覆盖了接口中的所有的抽象方法的子类，才可以实例化，否则，该子类还是一个抽象类
3、接口是用来被实现的，而之前抽象类的操作是用一个类继承抽象类并覆盖抽象类中的抽象方法，从而达到实例化的效果的

类与接口之间的关系是：实现关系

代码如下：

```
interface Inter  // 先定义功能
{
    public static final NUM = 4;  //全局常量
    public abstract void show1(); //抽象方法
    public abstract void show2(); //抽象方法
}
class Demo implements Inter //再用子类去实现这个功能（用关键字implements）
{
        public void show1()  //实现接口中的功能
        {

        }
        public void show2()  //实现接口中的功能
        {

        }
}
class InterfaceDemo
{
        public static void main(String[] args)
        {
                Demo d = new Demo();//可以创建对象
        }
}
```

简言之，接口的特点就是：先把需要的功能是什么给定义出来，接着子类照着这个功能去实现（用关键字 implements）就行了。

接口的好处是什么?

前面说过“多继承”的好处是让子类具备更多的功能；但是它也有弊端，即存在调用的不确定性，这种不确定性产生的原因是在于方法主体的内容是不同的。Java中不直接支持多继承，而是对该机制进行改良，通过接口来解决问题。

```
interface InterA
{
        public void show1();
}
interface InterB
{
        public void show2();
}
class SubInter implements InterA,InterB //多实现
{
        public void show1()
        {
                System.out.println("InterA");
        }
        public void show2()
        {
```

```
                System.out.println("InterB");
        }
}
public class Company
{
        public static void main(String[] args)
        {
                SubInter S = new SubInter();
                S.show1();
                S.show2();
        }
}
```

在上面的代码中，在两个接口中定义了两个不同的功能（show1和show2），然后再子类中进行"多实现"，最后在主函数中分别调用show1()和show2()。那么，如果接口遇到了多继承中调用不确定的问题时，该如何解决呢？见如下代码，

```
interface InterA
{
        public void show();
}
interface InterB
{
        public void show();
}
class SubInter implements InterA,InterB //多实现
{
        public void show()
        {
                System.out.println("Inter");
        }
}
public class Company
{
        public static void main(String[] args)
        {
                SubInter S = new SubInter();
                S.show();
        }
}
```

在上面的代码中，两个接口中定义了同样的功能（show），那么在子类中进行实现时是如何处理的呢？
答：我们看到最后主函数中的S调用的是show()方法，那么这个show()方法是来自哪个接口呢？这个并不重要，重要的是我们已经在子类中对该方法进行了“覆盖”，所以最后S调用的是子类中的覆盖后的show()方法，而这个方法只有一个，因此解决了“多继承中调用不确定性的问题”。

通过继承和接口实现能够解决“单继承的局限性”，见如下代码，

```
class Fu // 父类
{
        void show1()
        {

        }
}
interface Inter1 // 接口
{
        public void method();
}
class Zi extends Fu implements Inter1 //继承父类得到基本功能，并且实现接口得到扩展功能
{
        public void show2() //Zi本身特有的功能
        {
```

```
        }
        public void method() // 写出函数体覆盖接口中的功能，
        {

        }
}
public class InterfaceClass
{
        public static void main(String[] args)
        {
              Zi z = new Zi();
              z.show1(); // 继承得到的基本功能
              z.show2(); // 对象本身特有的功能
              z.method(); // 实现接口得到的扩展功能
        }
}
```

上面的代码中，Zi 通过继承 Fu ，具备了Fu中的基本功能（show1()），接着由于Zi想要再扩展一些其他的功能，然而Java不支持多继承，因此可以以通过接口实现的方式来达到扩展功能（method()），这样就避免了单继承的局限性。简言之，继承是为了获取体系的基本功能，接口实现是获取扩展功能。

关系

类与类之间是继承关系，类与接口之间是实现关系，那么接口与接口之间是什么关系呢？答：接口与接口之间是继承关系，而且支持多继承。刚才说了Java不支持多继承，那么到这了怎么又支持多继承了呢？前面之所以说不可以“多继承”是因为会造成调用的不确定性，而在接口与接口之间只定义了函数的功能，并没有定义其具体的内容，因此不会造成调用的不确定，因此接口可以支持多继承。

```
interface Inter1 // 接口
{
        public void method();
}
interface Inter2 extends Inter1 // 接口Inter2继承了接口Inter1
{
        public void method2();
}
class Fu implements Inter2 // 这个类要实现接口Inter2,必须要覆盖两个方法，即Inter1中的方法（method()），与Inter2中的方法（method()2）,这样才能实例化
{
        public void method()
        {

        }
        public void method2()
        {

        }
}
```

问题

如果我们只需要接口中的部分功能，但是为了实例化，还是必须覆盖接口中的全部方法，这样代码的复用性就会很差。怎么办呢？解决的方法是，定义一个抽象类，将接口中的方法全部进行“空实现”，然后在其他的类中继承该类，代码如下，

```
interface Inter1 // 接口
{
        public void method1();
        public void method2();
        public void method3();
        public void method4();
}
```

```
abstract class Demo implements Inter1 //将接口中的方法都进行“空实现”，由于函数体内部都是空的，所以该类创建对象是没有意义的，因此可以利用abstract将该类抽象,这就是所谓的“没有抽象方法的抽象类”
{
	public void method1() {}  // 空实现
	public void method2() {}
	public void method3() {}
	public void method4() {}
}
class DemoA extends Demo  // 继承抽象类中的空实现方法，并加以覆盖
{
	public void method1()
	{
		System.out.println("继承空实现中的方法，进行覆盖");
	}
}
```

接口的思想

1、接口的出现扩展了功能
2、接口其实就是暴露出来的规则
3、接口的出现降低了耦合性（解耦）
4、接口的出现，一方在使用接口，一方在实现接口

```
class Mouse
{

}
interface USB
{

}
class NewMouse extends Mouse implements USB //实现方式是USB
{

}
```

关于“接口”，我们更多的关注于实现接口，至于在哪个类中实现接口我们并不关心。例如USB接口，我们只想知道这台机器能否通过USB进行实现，至于这台机器是鼠标还是硬盘或者是充电宝，我们不关心这个具体的对象。

抽象类与接口的区别：

例子：
描述犬，里面有“吃”、“叫”，这都是抽象的，具体的由子类来完成，那么问题是，是定义成抽象类呢？还是定义成接口呢？先把两个都试一下

```
// 第一种，通过实现接口来完成
interface dog
{
	abstract void eat();
	abstract void shout();
}
class PoliceDog implements dog
{
	void eat() //方法覆盖
	{

	}
	void shout()
	{

	}
}
```

```
// 第二种，通过抽象类来完成
abstract class dog
{
	abstract void eat();
	abstract void shout();
}
class PoliceDog extends dog
{
	void eat() //方法覆盖
	{

	}
	void shout()
	{

	}
}
```

上面两种描述“犬”的方式都没有问题，那么接下来，如果想要再给犬添加一个“缉毒”的功能，这个时候就会发现如果把“缉毒”功能也通过抽象类的方式来完成就不行了，因为不是所有的犬都具备“缉毒”的功能的，因此，我们要把“缉毒”这个功能用接口实现的方式来完成，代码如下，

```
interface du  // 接口定义出额外功能
{
	abstract void jidu();
}
abstract class dog // 抽象类定义出基本功能
{
	abstract void eat();
	abstract void shout();
}
class PoliceDog extends dog implements du //继承基本功能，实现接口的额外功能
{
	void eat()
	{

	}
	void shout()
	{

	}
	public void jidu()
	{

	}
}
```

抽象类和接口的区别?

1、类与类之间是继承关系；类与接口之间是实现关系。
2、抽象类中可以定义抽象和非抽象方法，子类可以直接使用，或者覆盖使用；接口中定义的都是抽象方法，必须实现才能使用。

牢记

****
* 类是用于描述事物的共性基本功能。例如，“犬”类的基本功能有，吃、叫、睡 *
* 接口用于定义的都是事物的额外功能。例如，“缉毒” *
****