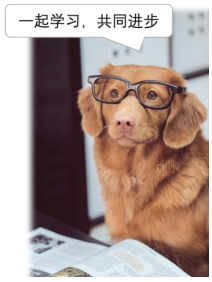
第0章 楔子　培训实习生——面向对象基础

本章主要是为阅读本书设计模式章节有困难的朋友提供的正餐前的开胃小菜。目的是不希望读者在阅读设计模式时，由于对Java语言中面向对象的知识了解匮乏或理解欠缺造成太大的障碍。所以本章仅对涉及阅读本书需要了解的Java语言中面向对象的知识做简单的介绍，比如继承、多态、接口、抽象、类、集合、泛型等。由于本书的重点不是讲面向对象基础，也不是讲Java语言，所以在本章中对很多技术细节都未提及或深入，希望有兴趣的朋友去阅读相关的专著，来补充对Java语言和面向对象的认识不足。

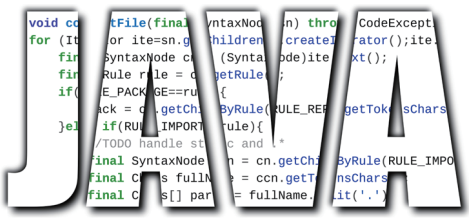
0.1 培训实习生



小菜，名叫蔡遥，24岁，上海人，上海某大学软件工程专业本科毕业。小菜工作两年多，因为勤思考、爱学习，成长很快，迅速成为公司里软件开发部门的骨干。

时间：9月3日 上午9点
地点：小菜办公室
人物：小菜、史熙、公司开发部经理

小菜的单位来了个大学实习生，叫史熙。开发部经理安排小菜有空教教他，小菜欣然接受。



“蔡老师，请多多关照了。”史熙很诚恳。

“哈，不敢当，我也工作没几年，比你大几届。以后叫我名字吧，我叫蔡遥。”

“哦，那叫遥哥。”

“唉！不跟你客气了，你是学计算机专

业的吗？”

“是的，今年大四。不过老实说，在学校里真的没学到什么东西。所谓‘公司一日，校园一年。’来这一定会比学校学的东西要多。”

“哈，夸张了，应该是‘公司一日抵自学一月。’在实践当中，自然会学得快一些。不过话说回来，一些基本的东西还是要知道的。Java语言学过吧？对面向对象又了解多少？”

“Java是学过的，什么变量、常量、判断、循环，我都知道。面向对象，好像也学过的。什么类呀、方法呀的，太久了，当时就为了应付考试了，具体如何用，根本不记得。你要不给我讲讲吧。”

“啊，都不记得了，不就等于白学了吗？不过没关系，今天我正好有空，我们争取来个快速入门吧。”

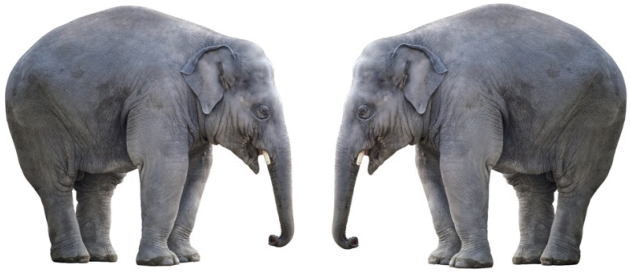
“太好了，遥哥，不对，还是得叫蔡老师，先谢谢了。”

0.2 类与实例

“先问问你，对象是什么？类是什么？”小菜问道。

“准确定义不知道。类大概就是对东西分类的意思。”史熙答得很勉强。

“啊，看来你是实实在在的菜鸟呀。一切事物皆为对象，即所有的东西都是对象，对象就是可以看到、感觉到、听到、触摸到、尝到或闻到的东西。准确地说，**对象是一个自包含的实体，用一组可识别的特性和行为来标识**。面向对象编程，英文叫Object-Oriented Programming，其实就是针对对象来进行编程的意思。”



“至于类，待会儿再讲，先从最简单的开始，你可以用Java编写一个小程序，最终将实现一个‘动物运动会’的软件小例子。”

“动物运动会？有意思。”

“首先实现这样一个功能，用Java实现一声猫叫，我们暂时无法模拟出真实声音，那就在控制台显

示叫声‘喵’就可以了。”

“这个非常简单。”

```
public class Test {
    public static void main(String[] args){
        System.out.println("喵");
    }
}
```

“好了，是这个意思吗？” 史熙问道。

“是，程序是编出来了，现在的问题就是，如果我们需要在另一个按钮中来让小猫叫一声，或者需要小猫多叫几声，怎么办？”

“那就多写几个‘System.out.println(" 喵 "); ’呀。”

“那不就重复了吗？可不可以想个办法？”

“我知道你的意思，写个函数就可以解决了。其他需要猫叫的地方都可以调用”

```
public class Test {
    public static void main(String[] args){
        System.out.println(shout());
    }
    String shout(){
        return "喵";
    }
}
```

“很好，现在问题是，万一别的地方，我指程序的其他地方也需要猫叫shout()，如何处理？”

“这个我好像学过的，在shout()方法前面加一个public，别的场合就可以访问了。”

“对，没错，这样是可以达到访问的目的了，但是你觉得这个shout(猫叫)放在这个Test.java的代码中合适吗？这就好比，居委会的公用电视放在你家，而别人家都没有，于是街坊邻居都来你家看电视。你喜欢这样吗？”

“这样好呀，邻居关系好了。不过这确实不是办法，公用电视应该放在居委会。”

“所以说，这猫叫的函数应该放在一个更合适的地方，这就是‘类’。**类就是具有相同的属性和功能的对象的抽象的集合**，我们来看代码。”

```
public class Cat {
    public String shout(){
        return "喵";
    }
}
```

“这里‘class’**是表示定义类的关键字**，‘Cat’就是类的名称，‘shout’就是类的方法。”

“哈，定义类这玩意还是很简单的嘛。”

“这里有两点要注意

（1）类名称首字母记着要大写。多个单词则各个首字母大写；

（2）对外公开的方法需要用‘public’修饰符。”

“明白，那怎么应用这个类呢？”

“很简单，只要将类实例化一下就可以了。”

“什么叫实例化？”

“实例，就是一个真实的对象。比如我们都是‘人’，而你和我其实就是‘人’类的实例了。**而实例化就是创建对象的过程，使用new关键字来创建。”**

```
public class Test {

    public static void main(String[] args){

        Cat cat = new Cat();
        System.out.println(cat.shout());

    }
}
```

“注意，‘Cat cat = new Cat();’其实做了两件事。”

```
Cat cat;              //声明一个Cat对象，对象名为cat

cat = new Cat();      //将此cat对象实例化
```

“Cat实例化后，等同于出生了一只小猫cat，此时就可以让小猫cat.shout()了。在任何需要小猫叫的地方都可以实例化它。”

“明白，这下调用它确实就方便很多了。”

0.3 构造方法

“下面，我们希望出生的小猫应该有个姓名，比如叫‘咪咪’，当咪咪叫的时候，最好是能说‘我的名字叫咪咪，喵’。此时就需要考虑用构造方法。”

“构造方法？这是做什么用的？”

“构造方法，又叫构造函数，其实就是对类进行初始化。构造方法与类同名，无返回值，也不需要void，在new的时候调用。”

“那就是说，在类创建时，就是调用构造方法的时候了？”

“是呀，在‘Cat cat=new Cat();’中，new

后面的Cat()其实就是构造方法。”

“不对呀，在类当中没有写过构造方法Cat()，怎么可以调用呢？”

“问得好，实际情况是这样，**所有类都有构造方法，如果你不编码则系统默认生成空的构造方法，若你有定义的构造方法，那么默认的构造方法就会失效了。**也就是说，由于你没有在Cat类中定义过构造方法，所以Java语言会生成一个空的构造方法Cat()。当然，这个空的方法什么也不做，只是为了让你能顺利地实例化而已。”

“那不是很好吗？我们还需要构造方法做什么呢？”

“刚才不是说过吗，构造方法是为了对类进行初始化。比如我们希望每个小猫一诞生就有姓名，那么就应该写一个有参数的构造方法。”

```
public class Cat {

    //声明Cat的私有字符串变量name
    private String name = "";
    //定义Cat类的构造方法，参数是输入一个字符串
    public Cat(String name){
        //将参数赋值给私有变量name
        this.name = name;
    }

    public String shout(){
        return "我的名字叫"+ name + " 喵";
    }
}
```

“这样一来，我们在客户端要生成小猫时，就必须要给小猫起名字了。”

```
public class Test {

    public static void main(String[] args){

        Cat cat = new Cat("咪咪");
        System.out.println(cat.shout());

    }
}
```

结果显示：

0.4 方法重载

“但是，遥哥，如果我事先没有起好小猫的名字，难道这个实例就创建不了了吗？”史熙又有疑问。

“是的，有些父母刚生下孩子时，姓名没有起好也是很正常的事。就目前的代码，你如果写‘Cat cat = new Cat();’是会直接报‘Cat方法没有采用0个参数的重载’的错误，原因就是必须要给小猫起名字。如果当真需要不起名字也要生出小猫来,可以用‘方法重载’。”

“方法重载？好像也学过，具体如何说？”

“**方法重载提供了创建同名的多个方法的能力，但这些方法需使用不同的参数类型。**注意并不是只有构造方法可以重载，普通方法也是可以重载的。”

```
public class Cat {

    private String name = "";
    public Cat(String name){
        this.name = name;
    }

    //将构造方法重载
    public Cat(){
        this.name = "无名";
    }

    public String shout(){
        return "我的名字叫"+ name + "喵";
    }
}
```

“哦，这样的话，如果写‘Cat cat = new Cat();’的话，就不会报错了。而猫叫时会是‘我的名字叫无名 喵’。”

“对的，注意**方法重载时**，两个方法必须要**方法名相同，但参数类型或个数必须要有所不同**，否则重载就没有意义了。你觉得方法重载的好处是什么？”小菜问道。

“哈，我想应该是**方法重载可在不改变原方法的基础上，新增功能。**”

“说得很好，方法重载算是提供了函数可扩展的能力。比如刚才这个例子，有的小猫起好名字了，就用带string参数的构造方法，有的没有名字，就用不带参数的，这样就达到了扩展的目的。”

“如果我需要分清楚猫的姓和名，还可以再重载一个public Cat(string firstName，string lastName)，对吧？”

“对的。非常好。下面，我们觉得小猫叫的次数太少，希望是我让它叫几声，它就叫几声，如何做？”

“那是不是在构造方法里再加一个叫的次数？”

“那样当然是可以，但叫几声并不是必须要实例化的时候就声明的，我们可以之后再规定叫几声，所以这时应该考虑用‘属性’。”

0.5 属性与修饰符

“**属性是一个方法或一对方法，即属性适合于以字段的方式使用方法调用的场合。**这里还需要解释一下字段的意思，**字段是存储类要满足其设计所需要的数据，字段是与类相关的变量。**比如刚才的Cat类中的‘private string name = " "；’name其实就是一个字段，它通常是私有的类变量。那么属性是什么样呢？我们现在增加一个猫叫次数ShoutNum的属性。”

```
public class Cat {

    //声明一个内部字段，注意是private，默认叫的次数为3
    private int shoutNum = 3;
    //表示外界可以给内部的shoutNum赋值
    public void setShoutNum(int value){
        this.shoutNum=value;
    }
    //表示外界调用时可以得到shoutNum的值
    public int getShoutNum(){
        return this.shoutNum;
    }

}
```

“刚才没有强调public和private的区别，它们都是修饰符，public表示它所修饰的类成员可以允许其他任何类来访问，俗称公有的。而private表示只允许同一个类中的成员访问，其他类包括它的子类无法访问，俗称私有的。如果在类中的成员没有加修饰符，则被认为是private的。修饰符还有其他三个，以后再讲。通常字段都是private，即私有的变量，而属性都是public，即公有的变量。那么在这里shoutNum就是私有的字段，而ShoutNum就是公有的对外属性。由于是对外的，所以属性的名称一般首字母大写，而字段则一般首字母小写或前加‘_’。”

“属性的get和set是什么意思？”

“属性有两个方法get和set。get返回与声明的属性相同的数据类型，表示的意思是调用时可以得到内部字段的值或引用；set有一个参数，用关键字value表示，它的作用是调用属性时可以给内部的字段或引用赋值。”

“那又何必呢？我把字段的修饰符改为public，不就可以做到对变量的既读又写了吗？”

“是的，如果仅仅是可读可写，那与声明了public的字段没什么区别。但是对于对外界公开的数据，我们通常希望能做更多的控制，这就好像我们的房子，我们并不希望房子是全透明的，那样你在家里的所有活动全部都被看得清清楚楚，毫无隐私可言。通常我们的房子有门有窗，但更多的是不透明的墙。这门和窗其实就是public，而房子内的东西，其实就是private。而对于这个房子来说，门窗是可以控制的，我们并不是让所有的人都可以从门随意进出，也不希望蚊子苍蝇来回出入。这就是属性的作用了，如果你把字

段声明为public，那就意味着不设防的门窗，任何时候，调用者都可以读取或写入，这是非常糟糕的一件事。如果把对外的数据写成属性，那情况就会好很多。”

```
public class Cat {

    private int shoutNum = 3;
    public int getShoutNum(){
        return this.shoutNum;
    }

    //去掉set,表示ShoutNum
    //属性是只读的

}
```

```
public class Cat {

    private int shoutNum = 3;
    public void setShoutNum(int value){

        //控制叫声次数，最多只能叫10声
        if (value<=10)
            this.shoutNum=value;
        else
            this.shoutNum = 10;

    }
    public int getShoutNum(){
        return this.shoutNum;
    }

}
```

“我明白了，这就好比给窗子安装了纱窗，只让阳光和空气进入，而蚊子苍蝇就隔离。多了层控制就多了层保护。”

“说得很好。我们还没有做完，由于有了‘叫声次数’的属性，于是我们的shout方法就需要改进了。”

```
public String shout(){
    String result="";
    //用一个循环让小猫叫相应的次数
    for(int i=0;i<this.shoutNum;i++){
        result+= "喵 ";
    }
    return "我的名字叫"+ name + " " + result;
}
```

“此时调用的时候只需要给属性赋值就可以了。”

```
Cat cat = new Cat("咪咪");

cat.setShoutNum(5);    //给属性赋值

System.out.println(cat.shout());
```

结果显示：

“如果我们不给属性赋值，小猫会叫‘喵’吗？”

“当然会，应该是三声吧，因为字段shoutNum的初始值是3。”

“很好。另外需要强调的是，变量私有的叫字段，公有的是属性，那么对于方法而言，同样也就有私有方法和公有方法了，一般不需要对外界公开的方法都应该设置其修饰符为private（私有）。这才有利于‘封装’”

0.6 封装

“现在我们可以讲面向对象的三大特性之一‘封装’了。**每个对象都包含它能进行操作所需要的所有信息，这个特性称为封装，因此对象不必依赖其他对象来完成自己的操作。**这样方法和属性包装在类中，通过类的实例来实现。”

“是不是刚才提炼出Cat类，其实就是在做封装？”

“是呀，**封装有很多好处。第一，良好的封装能够减少耦合，**至少我们让Cat和Form1的耦合分离了。**第二，类内部的实现可以自由地修改，**这也是显而易见的，我们已经对Cat做了很大的改动。**第三，类具有清晰的对外接口，**这其实指的就是定义为public的ShoutNum属性和shout方法。”

“封装的好处很好理解，比如刚才举的例子。我们的房子就是一个类的实例，室内的装饰与摆设只能被室内的居住者欣赏与使用，如果没有四面墙的遮挡，室内的所有活动在外人面前一览无遗。由于有了封装，房屋内的所有摆设都可以随意地改变而不用影响他人。然而，如果没有门窗，一个包裹得严严实实的黑箱子，即使它的空间再宽阔，也没有实用价值。房屋的门窗，就是封装对象暴露在外的属性和方法，专门供人进出，以及流通空气、带来阳光。”

“现在我需要增加一个狗叫的功能，就是加一个按钮‘狗叫’，单击后会弹出‘我的名字叫XX 汪 汪 汪’如何做？”

“那简单呀，仿造Cat加一个Dog类。然后再用类似代码调用就好了。”

```
Dog dog = new Dog("旺财");

dog.setShoutNum(8);

System.out.println(dog.shout());
```

结果显示：

“这下就OK了，小狗旺财也会叫了。”

“很好，但你有没有发现，Cat和Dog有非常类似的代码？”

“是呀，90%的代码是一样的，不过这些代码都是必需的，也没什么办法去除呀。”

“当然可以想办法，代码有大量重复不会是什么好事情。我们要用到面向对象的第二大特性‘继承’。”

0.7 继承

“我们还是先离开软件编程，来想想我们的动物常识，猫和狗都是什么？”小菜问道。

“都是给人添麻烦的东西。除了吃喝拉撒睡，什么也不干的家伙。”史熙调皮地答道。

“拜托，正经一些。猫和狗都是动物，准确地说，他们都是哺乳动物。哺乳动物有什么特征？”

“哦，这个小时候学过，哺乳动物是胎生、哺乳、恒温的动物。”

“OK，因为猫和狗是哺乳动物，所以猫和狗就同样具备胎生、哺乳、恒温的特征。所以我们可以这样说，由于猫和狗是哺乳动物，所以猫和狗与哺乳动物是继承关系。”

“哦，原来继承就是这个意思。”

“是的，回到编程上，**对象的继承代表了一种‘is-a’的关系，如果两个对象A和B，可以描述为‘B是A’，则表明B可以继承A。**‘猫是哺乳动物’，就说明了猫与哺乳动物之间是继承与被继承的关系。实际上，**继承者还可以理解为是对被继承者的特殊**

化，因为它除了具备被继承者的特性外，还具备自己独有的个性。例如，猫就可能拥有抓老鼠、爬树等‘哺乳动物’对象所不具备的属性。因而在继承关系中，继承者可以完全替换被继承者，反之则不成立。所以，我们在描述继承的‘is-a’关系时，是不能相互颠倒的。说‘哺乳动物是猫’显然有些莫名其妙。**继承定义了类如何相互关联，共享特性。继承的工作方式是，定义父类和子类，或叫作基类和派生类，其中子类继承父类的所有特性。子类不但继承了父类的所有特性，还可以定义新的特性。**”

“‘is-a’这个比较好理解。”

“学习继承最好是记住三句话，**如果子类继承于父类，第一，子类拥有父类非private的属性和功能；第二，子类具有自己的属性和功能，即子类可以扩展父类没有的属性和功能；第三，子类还可以以自己的方式实现父类的功能（方法重写）。**”

“这里有些不理解，什么叫非private，难道除了public还有别的修饰符吗？”

“当然有，刚才讲了private和public，现在再讲一个protected修饰符。**protected表示继承时子类可以对基类有完全访问权，**也就是说，用protected修饰的类成员，对子类公开，但不对其他类公开。所以子类继承于父类，则子类就拥有了父类的除private外的属性和功能，注意除这三个修饰符外还有两个，由于和目前所讲的内容无关，就留给你自己去查MSDN看吧。”

“那方法重写是什么意思？”

“这个留到后面讲多态的时候去说，现在我们来看看怎么做。对比观察Cat和Dog类。”

```
public class Cat{

    private String name = "";
    public Cat(String name){
        this.name = name;
    }

    public Cat(){
        this.name="无名";
    }

    private int shoutNum = 3;
    public void setShoutNum(int value){
        this.shoutNum=value;
    }
    public int getShoutNum(){
        return this.shoutNum;
    }

    public String shout(){
        String result="";
        for(int i=0;i<this.shoutNum;i++){
            result+= "喵";
        }
        return " 我的名字叫"+ name + " " + result;
    }
}
```

```
public class Dog {

    private String name = "";
    public Dog(String name){
        this.name = name;
    }

    public Dog(){
        this.name="无名";
    }

    private int shoutNum = 3;
    public void setShoutNum(int value){
        this.shoutNum=value;
    }
    public int getShoutNum(){
        return this.shoutNum;
    }

    public String shout(){
        String result="";
        for(int i=0;i<this.shoutNum;i++){
            result+= "汪";
        }
        return " 我的名字叫"+ name + " " + result;
    }
}
```

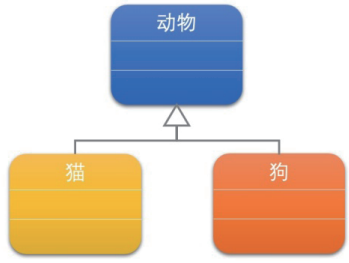
“我们会发现大部分代码都是相同的，所以我们现在建立一个父类，动物Animal类，显然猫和狗都是动物。我们把相同的代码尽量放到动物类中。”

```java
public class Animal {
    protected String name = "";
    public Animal(String name){
        this.name = name;
    }

    public Animal(){
        this.name = "无名";
    }
    protected int shoutNum = 3;
    public void setShoutNum(int value){
        this.shoutNum = value;
    }
    public int getShoutNum(){
        return this.shoutNum;
    }

    public String shout(){
        return "";
    }
}
```

注意修饰符改成了protected

注意修饰符改成了protected

“然后我们需要写Cat和Dog的代码。让它们继承Animal。这样重复的部分都可以不用写了，不过在Java中，**子类从它的父类中继承的成员有方法、属性等，但对于构造方法，**有一些特殊，**它不能被继承，只能被调用。对于调用父类的成员，可以用**base**关键字。**”




```java
public class Cat extends Animal {

    public Cat(){
        super();
    }

    public Cat(String name){
        super(name);
    }
    public String shout(){
        String result="";
        for(int i=0;i<this.shoutNum;i++){
            result += "喵 ";
        }
        return "我的名字叫"+ name + " " + result;
    }
}
```

继承格式就是使用extends让Cat继承Animal

子类构造方法需要调用与父类同样参数的构造方法，用super关键字

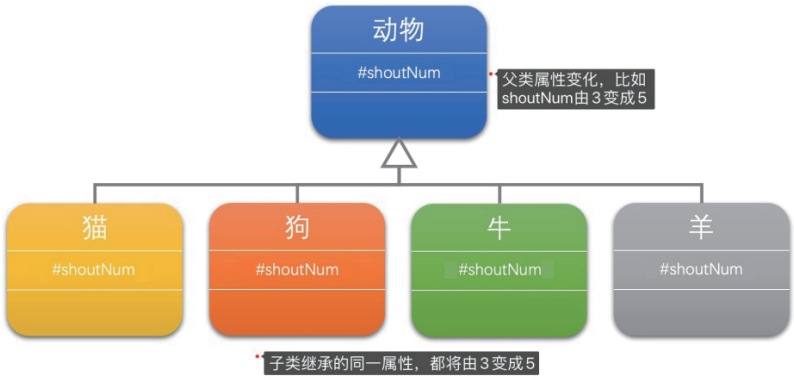
```java
public class Dog extends Animal {

    public Dog(){
        super();
    }

    public Dog(String name){
        super(name);
    }

    public String shout(){
        String result="";
        for(int i=0;i<this.shoutNum;i++){
            result += "汪 ";
        }
        return "我的名字叫"+ name + " " + result;
    }
}
```

“此时客户端的代码完全一样，没有受到影响，但重复的代码却因此减少了。”小菜说。

“差不太多嘛，子类还是有些复杂，没简单到哪去？”史熙说道。

“如果现在需要加牛、羊、猪等多个类似的类，按你以前的写法就需要再复制三遍，也就是有五个类。如果我们需要改动起始的叫声次数，也就是让shoutNum由3改为5，你需要改几个类？”

“我懂你的意思了，那需要改5个类，现在有了Animal，就只要改一个类就行了，继承可以使得重复减少。”



“说得很好。**不用继承的话，如果要修改功能，就必须在所有重复的方法中修改，代码越多，出错的可能就越大，而继承的优点是，继承使得所有子类公共的部分都放在了父类，使得代码得到了共享，这就避免了重复，另外，继承可使得修改或扩展继承而来的实现都较为容易。**”

“嗯，继承是个好东西，我以前时常Ctrl+C加Ctrl+V的，这样表面上很快，但其实重复的代码越多，以后要更改的难度越大。看来以后我要多多用继承。”

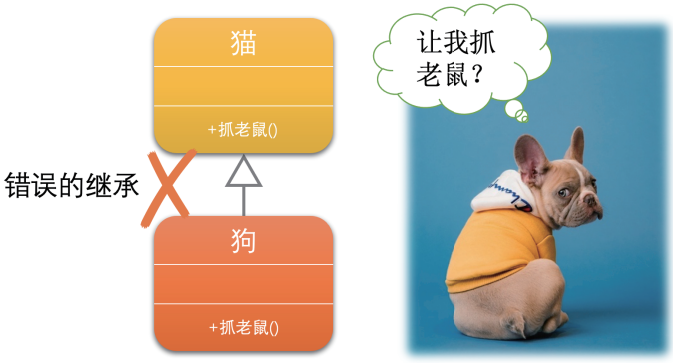
“慢，等等，你说以后要多多用继承？那可不一定是好事，用是可以，但一定要慎重。”

“有这么严重吗？反正只要是有重复的时候，就继承一个子类不就行了吗？”

“哈，那真是大错特错了，那我问你，你先写了Cat，而后要你写一个Dog，由于都差不多，你有没有考虑过直接让狗去继承猫呢？”

“咦，这其实也差不多哦，没什么问题呀。如果再有了羊呀、牛呀也都分别继承猫就可以了。”

“问题就在这里了。这就使得猫会的行为，狗都会。现在编写的这只猫只会叫，以后它应该还可以会抓老鼠、爬树等行为，你让狗继承了猫，就意味着狗也就会抓老鼠、会爬树。你觉得这合理吗？”



“狗拿耗子，那是多管闲事了。看来不能让狗继承猫，那样很容易造成麻烦。”

“**继承是有缺点的，那就是父类变，则子类不得不变。**让狗去继承于猫，显然不是什么好的设计。另外，**继承会破坏包装，父类实现细节暴露给子类，**这其实是增大了两个类之间的耦合性。”

“什么叫耦合性？”

“严格定义你自己去查吧，简单理解就是藕断丝连，两个类尽管分开，但如果关系密切，一方的变化都会影响到另一方，这就是耦合性高的表现，**继承显然是一种类与类之间强耦合的关系。**”

“明白，你说了这么多，那什么时候用继承才是合理的呢？”

“我最先不是说过吗？**当两个类之间具备‘is-a’的关系时，就可以考虑用继承了，**因为这表示一个类是另一个类的特殊种类，而当两个类之间是‘has-a’的关系时，表示某个角色具有某一项责任，此时不适合用继承。比如人有两只手，手不能继承人；再比如飞机场有飞机，飞机也不能去继承飞机场。”

“OK，也就是说，只有合理的应用继承才能发挥好的作用。”

0.8 多态

“下面我们再来增加需求，如果我们要举办一个动物运动会，来参加的有各种各样的动物，其中有一项是‘叫声比赛’。界面就是放两个按钮，一个是‘动物报名’，就是确定动物的种类和报名的顺序；另一个是‘叫声比赛’，就是报名的动物挨个地叫出声音来比赛。注意来报名的都是什么动物，我们并不知道。可能是猫、可能是狗，也可能是其他的动物，当然它们都需要会叫才行。”

“有点复杂，我除了会加两个按钮外，不知道如何做了。”

“先分析一下，来报名的都是动物，参加叫声比赛必须会叫。这说明什么？”

“说明都有叫的方法，哦，也就是Animal类中有shout方法。”

“是呀，所谓的‘动物报名’，其实就是建立一个动物对象数组，让不同的动物对象加入其中。所谓的‘叫声比赛’，其实就是遍历这个数组来让动物们‘shout()’就可以了。”

“哦，我大概明白你的意思了，那看看我写的代码，我觉得应该是类似的样子，但问题是我们不知道是哪个动物来报名，最终叫的时候到底是猫在叫还是狗在叫呢？”

“是呀，就之前讲到的知识，是不足以解决这个问题的，所以我们引入**面向对象的第三大特性——多态。**”

```java
public class Test {

    public static void main(String[] args){

        //有五个动物报名的资格
        Animal[] arrayAnimal=new Animal[5];

        //报名代码
        
        //开始叫声比赛，遍历这个数组，让每个动物对象都shout
        for(int i=0;i<5;i++){
            System.out.println(arrayAnimal[i].shout());
        }
    }
}
```

什么动物来报名参赛是事先不知道的

什么动物怎么叫也是不确定的

“啊，多态，多态是我大学里听得很多，但一直都不懂的东西，实在不明白它是什么意思。”

同样是鸟，同样展开翅膀的动作，但老鹰、鸵鸟和企鹅之间，是完全不同的作用。老鹰展开翅膀用来更高更远地飞翔，鸵鸟用来更快更稳地奔跑，而企鹅则是更急更流畅地游泳。这就是生物多态性表现。在面向对象中，**“多态表示不同的对象可以执行相同的动作，但要通过它们自己的实现代码来执行。**

我们都是鸟

看定义显然不太明白，我再来举个例子。我们的国粹‘京剧’以前都是子承父业，代代相传的艺术。假设有这样一对父子，父亲是非常有名的京剧艺术家，儿子长大成

人，模仿父亲的戏也惟妙惟肖。有一天，父亲突然发高烧，上不了台表演，而票都早就卖出，退票显然会大大影响声誉。怎么办呢？由于京戏都是需要化妆才可以上台的，于是就决定让儿子代父亲上台表演。”

“化妆后谁认识谁呀，只要唱得好就可以糊弄过去了。”

“是呀，这里面有几点注意，**第一，子类以父类的身份出现，**儿子代表老子表演，化妆后就是以父亲身份出现了。**第二，子类在工作时以自己的方式来实现，**儿子模仿得再好，那也是模仿，儿子只能用自己理解的表现方式去模仿父亲的作品；**第三，子类以父类的身份出现时，子类特有的属性和方法不可以使用，**儿子经过多年学习，其实已经有了自己的创作，自己的绝活，但在此时，代表父亲表演时，绝活是不能表现出来的。当然，如果父亲还有别的儿子会表演，也可以在此时代表父亲上场，道理也是一样的。这就是多态。”

“听听好像都懂，怎么用呢？”

“是呀，怎么用呢，我们还需要了解一些概念，方法重写。**子类可以选择使用override关键字，将父类实现替换为它自己的实现，这就是方法重写Override，或者叫作方法覆写。**我们来看一下例子。”

“由于Cat和Dog都有shout的方法，只是叫的声音不同，所以我们可以让Animal有一个shout的方法，然后Cat和Dog去重写这个shout，用的时候，就可以用猫或狗代替Animal叫唤，来达到多态的目的。”

```
public class Animal {

    ...

    public String shout(){
        return "";
    }
}
```

```
public class Cat extends Animal {

    public Cat(){
        super();
    }

    public Cat(String name){
        super(name);
    }

    public String shout(){
        String result="";
        for(int i=0;i<this.shoutNum;i++){
            result += "喵 ";
        }
        return "我的名字叫"+ name + " " + result;
    }
}
```

```
public class Dog extends Animal {

    public Dog(){
        super();
    }

    public Dog(String name){
        super(name);
    }

    public String shout(){
        String result="";
        for(int i=0;i<this.shoutNum;i++){
            result += "汪 ";
        }
        return "我的名字叫"+ name + " " + result;
    }
}
```

“再回到你刚才写的客户端代码上。”

```
public class Test {

    public static void main(String[] args){

        //有五个动物报名的资格
        Animal[] arrayAnimal=new Animal[5];

        //报名代码
        arrayAnimal[0] = new Cat("小花");
        arrayAnimal[1] = new Dog("阿毛");
        arrayAnimal[2] = new Dog("小黑");
        arrayAnimal[3] = new Cat("娇娇");
        arrayAnimal[4] = new Cat("咪咪");

        //开始叫声比赛，遍历这个数组，让每个动物对象都shout
        for(int i=0;i<5;i++){
            System.out.println(arrayAnimal[i].shout());
        }
    }
}
```

报名的是猫、狗、狗、猫、猫，根据需要还可以是牛、羊、猪等任何动物

结果显示，先单击“动物报名”，然后“叫声比赛”，将有五个对话列出。

“我明白了，Animal相当于京剧表演的老爸，Cat和Dog相当于儿子，儿子代表父亲表演shout，但Cat叫出来的是‘喵’，Dog叫出来的是‘汪’，这就是所谓的**不同的对象可以执行相同的动作，但要通过它们自己的实现代码来执行。**”

“说得好，是这个意思，不过一定要注意了，这个对象的声明必须是父类，而不是子类，实例化的对象是子类，这才能实现多态。**多态的原理是当方法被调用时，无论对象是否被转换为其父类，都只有位于对象继承链最末端的方法实现会被调用。也就是说，虚方法是按照其运行时类型而非编译时类型进行动态绑定调用的。**[AMNFP]”

```
动物 animal = new 猫();
```

```
猫 cat = new 猫();
动物 animal= cat;
```

“不过老实说，即使这样，我也还是不太理解这样做有多大的好处。多态被称为面向对象三大特性，我感觉不到它有和封装、继承同样的作用。”

“慢慢来，要深刻理解并会合理利用多态，不去研究设计模式是很难做到的。也可以反过来说，**没有学过设计模式，那么对多态乃至对面向对象的理解多半都是肤浅和片面的。**我相信会有那种天才，可以听一知十，刚学的东西就可以灵活自如地应用，甚至

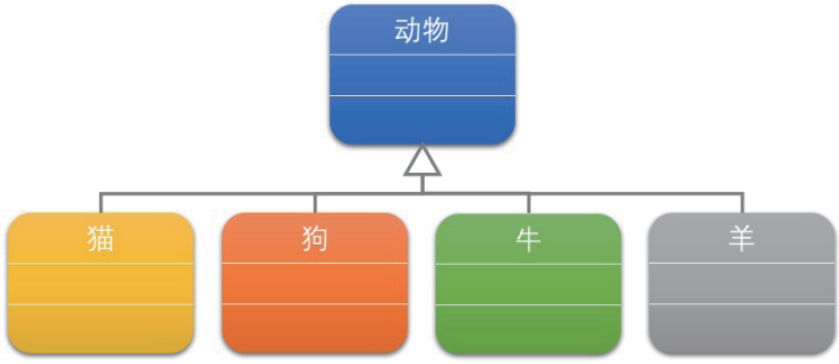
要造汽车，他都能再去发明轮子。但对于绝大多数程序员，还是需要踏踏实实地学习基本的东西，并在不断的实践中成长，最终成为高手。”

“蔡老师，受教了。下面我们做什么？”

0.9 重构

“现在又来了小牛和小羊来报名，需要参加‘叫声比赛’，你如何做？”

“这个简单了，我现在再实现牛Cattle和羊Sheep的类，让它们都继承Animal就可以了。”




```
public class Cattle extends Animal {

    public Cattle (){
        super();
    }

    public Cattle (String name){
        super(name);
    }

    public String shout(){
        String result="";
        for(int i=0;i<this.shoutNum;i++){
            result+= "哞 ";
        }
        return "我的名字叫"+ name + " " + result;
    }
}
```

```
public class Sheep extends Animal {

    public Sheep (){
        super();
    }

    public Sheep (String name){
        super(name);
    }

    public String shout(){
        String result="";
        for(int i=0;i<this.shoutNum;i++){
            result+= "咩 ";
        }
        return "我的名字叫"+ name + " " + result;
    }
}
```

“等等，你有没有发现，猫狗牛羊四个类，除了构造方法之外，还有重复的地方？”

“是呀，我发现了，shout里除了四种动物叫的声音不同外，几乎没有任何差异。”

“这有什么坏处？”

“重复呀，如果你有需求说，把‘我的名字叫XXX’改成‘我叫XXX’，我就得更改四个类的代码了。”

“非常好，所以这里有重复，我们还是应该要改造它。”

“我先把重复的这个shout的方法体放到Animal类中。”

“这样如何能行，动物叫什么声音呢？叫‘喵’？叫‘汪’？都不行，动物是个抽象的概念，它是不会有叫的声音的。”

“别急，我们把叫的声音部分改成另一个方法getShoutSound不就行了！”

```
public class Animal {

    ...

    public String shout(){

        String result="";

        for(int i=0;i<this.shoutNum;i++){
            result+= getShoutSound()+", ";
        }                    改成调用getShoutSound方法

        return "我的名字叫"+ name + " " + result;

    }

    protected String getShoutSound(){
        return "";           此方法留给继承的类具体实现，
    }                        所以用protected修饰

}
```

“此时的子类就极其简单了。除了叫声和构造方法不同，所有的重复都转移到了父类，真是漂亮之极。”

```
public class Cat extends Animal {

    public Cat (){
        super();
    }
    public Cat (String name){
        super(name);
    }

    protected String getShoutSound(){
        return "喵";
    }
}
```

```
public class Dog extends Animal {

    public Dog (){
        super();
    }
    public Dog (String name){
        super(name);
    }

    protected String getShoutSound(){
        return "汪";
    }
}
```

```
public class Sheep extends Animal {

    public Sheep (){
        super();
    }
    public Sheep (String name){
        super(name);
    }

    protected String getShoutSound(){
        return "咩";
    }
}
```

```
public class Cattle extends Animal {

    public Cattle (){
        super();
    }
    public Cattle (String name){
        super(name);
    }

    protected String getShoutSound(){
        return "哞";
    }
}
```

“有点疑问，这样改动，子类，比如Cat就没有Shout方法了，外面如何调用呢？”

“唉，你把继承的基本忘记了？继承的第一条是什么？”

“哈，是子类拥有所有父类非private的属性和方法。对的对的，由于子类继承父类，所以是public的Shout方法是一定可以为所有子类所用的。”史熙高兴地说，“我渐渐能感受到面向对象编程的魅力了，的确是非常简洁。由于不重复，所以需求的更改都不会影响到其他类。”

“这里其实就是在用一个设计模式，叫模板方法。（详见第10章）”

“啊，原来就是设计模式呀，Very Good，太棒了，哈，我竟然学会了设计模式。”

“疯了？发什么神经呀。”小菜同样微笑道，“这才是知道了皮毛，得意什么，还早着呢。”

0.10 抽象类

“我们再来观察，你会发现，Animal类其实根本就不可能实例化的，你想呀，说一只猫长什么样，可以想象，说new Animal(); 即实例化一个动物。一个动物长什么样？”

“不知道，动物是一个抽象的名词，没有具体对象与之对应。”

“是呀，所以我们完全可以考虑把实例化没有任何意义的父类，改成抽象类，同样地，对于Animal类的getShoutSound方法，其实方法体没有任何意义，所以可以将修饰符改为abstract，使之成为抽象方法。**Java允许把类和方法声明为abstract，即抽象类和抽象方法。**”

```
//声明一个抽象类，在class前增加abstract关键字
public abstract class Animal {

    ...

    //声明一个抽象方法，在返回值类型前加abstract
    //抽象方法没有方法体，直接在括号后加“;”
    protected abstract String getShoutSound();

}
```

“这样一来，Animal就成了抽象类。抽象类需要注意几点，**第一，抽象类不能实例**

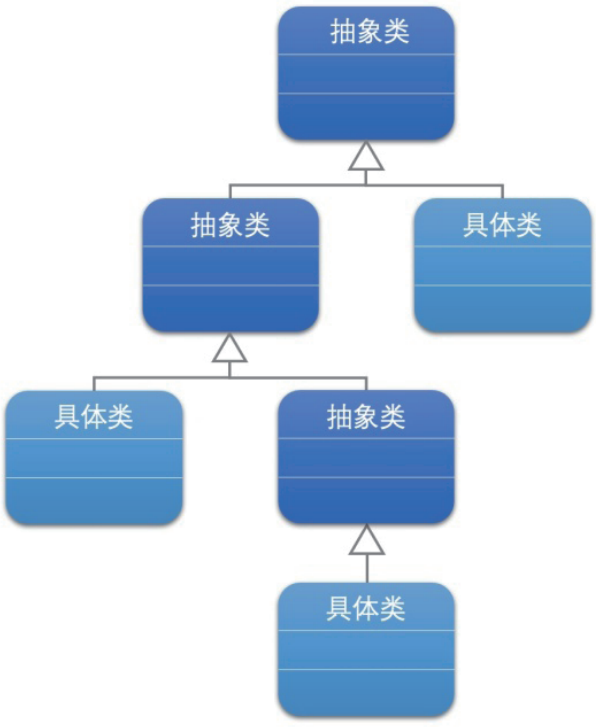
化，刚才就说过，‘动物’实例化是没有意义的；**第二，抽象方法是必须被子类重写的方法**，不重写的话，它的存在又有什么意义呢？其实抽象方法可以被看成是没有实现体的虚方法；**第三，如果类中包含抽象方法，那么类就必须定义为抽象类，不论是否还包含其他一般方法。**”

“这么说的话，一开始就可以把Animal类设成抽象类了，根本没有必要存在虚方法的父类。是这样吧？”史熙问道。

“的确是这样，我们应该考虑让**抽象类拥有尽可能多的共同代码，拥有尽可能少的数据**[J&DP]。”

“那到底什么时候应该用抽象类呢？”

“**抽象类通常代表一个抽象概念，它提供一个继承的出发点，当设计一个新的抽象类时，一定是用来继承的，所以，在一个以继承关系形成的等级结构里面，树叶节点应当是具体类，而树枝节点均应当是抽象类**[J&DP]。也就是说，具体类不是用来继承的。我们作为编程设计者，应该要努力做到这一点。比如，若猫、狗、牛、羊是最后一级，那么它们就是具体类，但如果还有更下面一级的金丝猫继承于猫、哈巴狗继承于狗，就需要考虑把猫和狗改成抽象类了，当然这也是需要具体情况具体分析的。”



“这个应该可以理解。”

“OK，我们继续下面的需求实现。”

0.11 接口

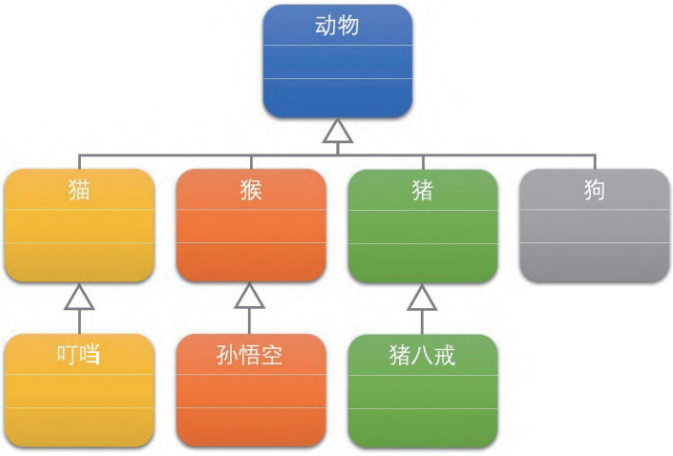
“在动物运动会里还有一项非常特殊的比赛是为了给有特异功能的动物展示其特殊才能的。”

“哈，有特异功能？有意思。不知是什么动物？”

“多的是呀，可以来比赛的比如有机器猫叮当、石猴孙悟空、肥猪猪八戒，再比如蜘蛛人、蝙蝠侠等。”

“啊，这都是什么动物呀，根本就是人们虚构之物。”

“让猫狗比赛叫声难道就不是虚构？你当它们会愿意相互攀比？其实我的目的只是为了让两个动物尽量不相干而已。现在叮当会从肚皮的口袋里变出东西，而孙悟空可以拔根毫毛变出东西，且有七十二般变化，八戒有三十六般变化。它们各属于猫、猴、猪，现在需要让它们比赛谁变东西的本领大。你来分析一下如何做？”



“‘变出东西’应该是叮当、孙悟空、猪八戒的行为方法，要想用多态，就得让猫、猴、猪有‘变出东西’的能力，而为了更具有普遍意义，干脆让动物具有‘变出东西’的行为，这样就可以使用多态了。”

“哈哈，史熙呀，你犯了几乎所有学面向对象的人都会犯的错误，‘变出东西’它是动物的方法吗？如果是，那是不是所有的动物都必须具备‘变出东西’的能力呢？”

“这个，确实不是，这其实只是三个特殊动物具备的方法。那应该如何做？”

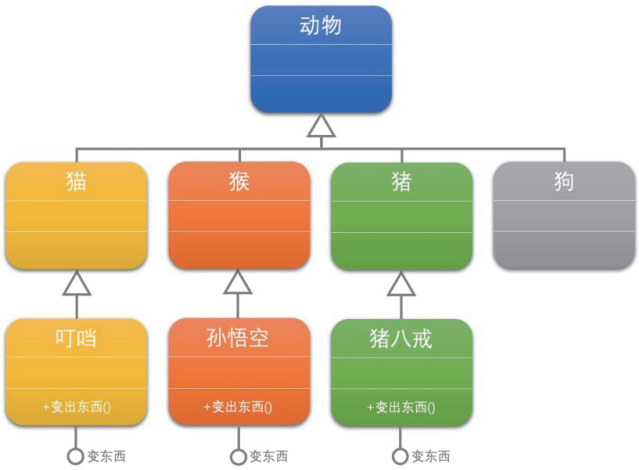
“这时候我们就需要新的知识了，那就是接口interface。通常我们理解的接口可能更多的是像电脑、音响等设备的硬件接口，比如用来传输电力、音视频、数据等接线的插口。而今天我们要提的，是面向对象编程里的接口概念。”

“**接口是把隐式公共方法和属性组合起来，以封装特定功能的一个集合。一旦类实现了接口，类就可以支持接口所指定的所有属性和成员。声明接口在语法上与声明抽象类完全相同，但不允许提供接口中任何成员的执行方式。**所以接口不能实例化，不能有构造方法和字段；不能有修饰符，比如public、private等；不能声明为虚拟的或静态的等。还有**实现接口的类就必须要实现接口中的所有方法和属性**。”

“怎么接口这么麻烦。”

“要求是多了点，但**一个类可以支持多个接口，多个类也可以支持相同的接口**。所以接口的概念可以让用户和其他开发人员更容易理解其他人的代码。哦，对了，**接口的命名，有些语言前面要加一个大写字母‘I’，**这是一种规范。”

“听不懂呀，不如讲讲实例吧。”



“我们先创建一个接口，它是用来‘变东西’的。注意**接口用interface声明，而不是class（接口名称前加‘I’会更容易识别），接口中的方法或属性前面不能有修饰符、方法没有方法体。**”

```
//声明一个IChange接口
public interface IChange {

    //此接口有一个方法changeThing,
    //参数是一个字符串变量，返回一字符
    public String changeThing(String thing);

}
```

“然后我们来创建机器猫的类。”

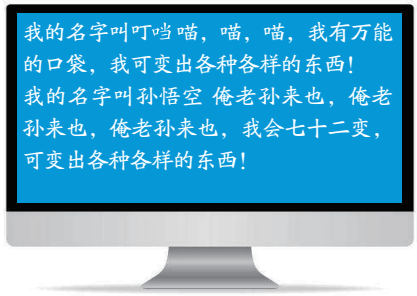
```java
//继承了Cat类，并实现了IChange接口
public class MachineCat extends Cat implements IChange {

    public MachineCat (){
        super();
    }
    public MachineCat (String name){
        super(name);
    }

    //实现了接口的方法
    public String changeThing(String thing){
        return super.shout()+ ", 我有万能的口袋，我可变出" + thing;
    }
}
```

super表示调用父类Cat的shout方法

“猴子的类Monkey和孙悟空的类StoneMonkey与上面非常类似，在此省略。此时我们的客户端，可以加一个‘变出东西’按钮，并实现下面的代码。”

```java
//创建两个类的实例
MachineCat mcat = new MachineCat("叮当");
StoneMonkey wukong = new StoneMonkey("孙悟空");

//声明了一个接口数组，将两个类的实例引用给接口数组
IChange[] array = new IChange[2];
array[0] = mcat;
array[1] = wukong;

//利用多态性，实现不同的changeThing
System.out.println(array[0].changeThing("各种各样的东西！"));
System.out.println(array[1].changeThing("各种各样的东西！"));
```

结果显示：



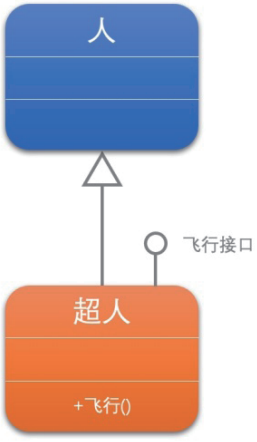
“哦，我明白了，其实这和抽象类是很类似的，由于我现在要让两个完全不相干的对象，叮当和孙悟空来做同样的事情‘变出东西’，所以我不得不让他们去实现做这件‘变出东西’的接口，这样的话，当我调用接口的‘变出东西’的方法时，程序就会根据我实现接口的对象来做出反应，如果是叮当，就是用万能口袋，如果是孙悟空，就是七十二变，利用了多态性完成了两个不同的对象本不可能完成的任务。”

“说得非常好，同样是飞，鸟用翅膀飞，飞机用引擎加机翼飞，而超人呢？举起两手，握紧拳头就能飞，它们是完全不同的对象，但是，如果硬要把它们放在一起的话，用一个飞行行为的接口，比如命名为IFly的接口来处理就是非常好的办法。”

“但是我对抽象类和接口的区别还是不太清楚。”

“问到点子上了，这两个概念的异同点是网上讨论面向对象问题时讨论得最多的话题之一，从表象上来说，**抽象类可以给出一些成员的实现，接口却不包含成员的实现，抽象类的抽象成员可被子类部分实现，接口的成员需要实现类完全实现，一个类只能继承一个抽象类，但可实现多个接口等。**但这些都是从两者的形态上去区分的。我觉得还有三点是能帮助我们去区分抽象类和接口。**第一，类是对对象的抽象，抽象类是对类的抽象，接口是对行为的抽象。**接口是对类的局部（行为）进行的抽象，而抽象类是对类整体（字段、属性、方法）的抽象。如果只关注行为抽象，那么也可以认为接口就是抽象类。总之，不论是接口、抽象类、类甚至对象，都是在不同层次、不同角度进行抽象的结果，它们的共性就是抽象。**第二，如果行为跨越不同类的对象，可使用接口；对于一些相似的类对象，用继承抽象类。**比如猫呀狗呀它们其实都是动物，它们之间有很多相似的地方，所以我们应该让它们去继承动物这个抽象类，而飞机、麻雀、超人是完全不相关的类，叮当是动漫角色，孙悟空是古代神话人物，这也是不相关的类，但它们又是有共同点的，前三个都会飞，而后两个都会变出东西，所以此时让它们去实现相同的接口来达到我们的设计目的就很合适了。”

“哦，明白，其实**实现接口和继承抽象类并不冲突**，我完全可以让超人继承人类，再实现飞行接口，是吗？”

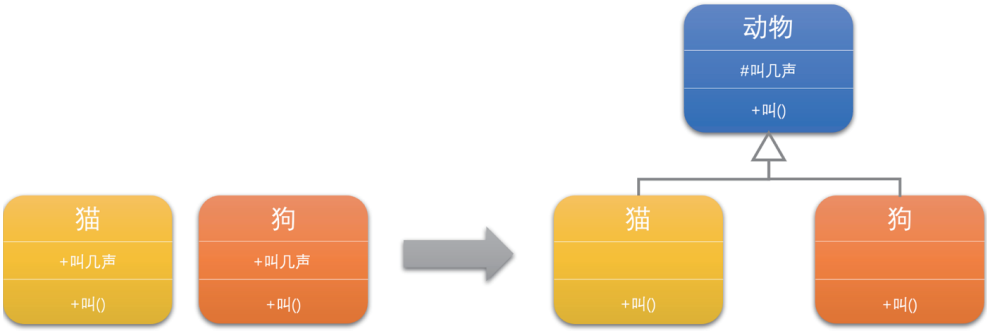


“对，超人除了内裤外穿以外，基本就是一个正常人的样子，让他继承人类是对

的，但他本事很大，除了飞天，他还具有刀枪不入、力大无穷等非常人的能力，而这些能力也可能是其他对象具备的，所以就让超人去实现飞行、力大无穷等行为接口，这就可以让超人和飞机比飞行，和大象比力气了，这就是一个类只能继承一个抽象类，却可以实现多个接口的做法。”

“那还有一点呢？”

“嗯，这一点更加关键，那就是**第三，从设计角度讲，抽象类是从子类中发现了公共的东西，泛化出父类，然后子类继承父类，而接口是根本不知子类的存在，方法如何实现还不确认，预先定义。**这里其实说明的是抽象类和接口设计的思维过程。回想一下我们今天刚开始讲的时候，先是有一个Cat类，然后再有一个Dog类，观察后发现它们有类似之处，于是泛化出Animal类，这也体现了敏捷开发的思想，**通过重构改善既有代码的设计。**事实上，只有小猫的时候，你就去设计一个动物类，这就极有可能会成为过度设计了。所以说，抽象类往往都是通过重构得来的，当然，如果你事先意识到多种分类的可能，那么事先就设计出抽象类也是完全可以的。”



“而接口就完全不是一回事。我们很早已经设计好了电源插座的接口，但在几十年前是无法想象未来会有什么样的电器需要电源插座的。只要事先把接口设计好，剩下的事就慢慢再说不着急了。”

“再比如我们是动物运动会的主办方，在策划时，大家坐在一起考虑需要组织什么样的比赛。大家商议后，觉得应该设置如跑得最快、跳得最高、飞得最远、叫得最响、力气最大等比赛项目。此时，主办方其实还不太清楚会有什么样的动物来参加运动会，所有的这些比赛项目都可能是完全不相同的动物在比，它们将如何去实现这些行为也不得而知，此时，能做的事就是事先定义这些比赛项目的行为接口。”

“啊，你的意思是不是说，抽象类是自底而上抽象出来的，而接口则是自顶向下设计出来的？”

“对，可以这么说。其实仅理解这一点是不够的，要想真正把抽象类和接口用好，还是需要好好用心地去学习设计模式。只有真正把设计模式理解好了，那么你才能算是真正会合理应用抽象类和接口了。”

0.12 集合

“下面我们再来看看，客户端的代码中，‘动物报名’用的是Animal类的对象数组，你设置了数组的长度为5，也就是说，最多只能有五个动物可以报名参加‘叫声比赛’，多了就不行了。这显然是非常不合理的，应该考虑改进。你能说说数组的优缺点吗？”

“数组的优点，比如说数组在内存中连续存储，因此可以快速而容易地从头到尾遍历元素，可以快速修改元素等。缺点嘛，应该是创建时必须要指定数组变量的大小，还有在两个元素之间添加元素也比较困难。”

“说得不错，的确是这样，这就可能使得数组长度设置过大，造成内存空间浪费，长度设置过小造成溢出。所以Java**提供了用于数据存储和检索的专用类，这些类统称集合。这些类提供对堆栈、队列、列表和哈希表的支持。大多数集合类实现相同的接口。**我们现在介绍当中最常用的一种，ArrayList。”

“集合？它和数组有什么区别？”

“别急，首先ArrayList**是程序包**java.util.ArrayList**下的一部分，它是使用大小可按需动态增加的数组实现**Collection**接口**。”

“哦，没学接口前不太懂，现在知道了，你的意思是说，Collection接口定义了很多集合用的方法，ArrayList对这些方法做了具体的实现？”

“对的，ArrayList**的容量是**ArrayList**可以保存的元素数。**ArrayList**的默认初始容量为0。随着元素添加到**ArrayList**中，容量会根据需要通过重新分配自动增加。使用整数索引可以访问此集合中的元素。此集合中的索引从零开始。”**

“是不是可以这样理解，数组的容量是固定的，而 ArrayList 的容量可根据需要自动扩充？”

“是的，由于实现了Collection，所以ArrayList 提供添加、插入或移除某一范围元素的方法。下面我们来看看如何做。”

```
//导入ArrayList所在的程序包
import java.util.ArrayList;

public class Test {

    public static void main(String[] args){

        //声明集合对象，并实例化对象
        ArrayList arrayAnimal=new ArrayList();

        //调用集合的add方法增加对象，参数是所有对象的抽象类Object，
        //所以new Cat()或new Dog()都是可以的
        arrayAnimal.add(new Cat("小花"));
        arrayAnimal.add(new Dog("阿毛"));
```

```
        arrayAnimal.add(new Dog("小黑"));
        arrayAnimal.add(new Cat("娇娇"));
        arrayAnimal.add(new Cat("咪咪"));

        //遍历集合
        for(Object item : arrayAnimal){
            Animal animal = (Animal)item;//此时需要强制将Object转换为Animal对象
            System.out.println(animal.shout());
        }

        System.out.println("动物个数: "+arrayAnimal.size());

    }
}
```

“如果有动物报完名后，由于某种原因（如政治、宗教、兴奋剂、健康等）放弃比赛，此时应该需要将其从名单中移除。例如，在报了名后，两只小狗需要退出比赛。我们查了一下它们的报名索引次序为1和2（从0开始计算），所以可以应用集合的remove方法，它的作用是移除指定索引处的集合项。”

“我明白怎么做了。”

```
arrayAnimal.add(new Cat("小花"));
arrayAnimal.add(new Dog("阿毛"));
arrayAnimal.add(new Dog("小黑"));
arrayAnimal.add(new Cat("娇娇"));
arrayAnimal.add(new Cat("咪咪"));

//阿毛和小黑两条狗要退出比赛，所以要移除它们
arrayAnimal.remove(1);
arrayAnimal.remove(2);
```

“哈，你太着急，集合与数组的不同就在于此，程序在执行RemoveAt(1)的时候，也就是叫‘阿毛’的Dog被移除了集合，此时‘小黑’的索引次序还是原来的2吗？”

“哦，我明白了，等于整个后序对象都向前移一位了。应该是这样才对。也就是说，集合的变化是影响全局的，它始终都保证元素的连续性。”

```
arrayAnimal.remove(1);
arrayAnimal.remove(1);
```

“总结一下，集合ArrayList相比数组有什么好处？”

“主要就是它可以根据使用大小按需动态增加，不用受事先设置其大小的控制。还有就是可以随意地添加、插入或移除某一范围元素，比数组要方便。”

“对，这是ArrayList的优势，但它也有不足，ArrayList不管你是什么对象都是接受的，因为在它眼里所有元素都是Object，这就使得如果你‘arrayAnimal.add(123)；’或者‘arrayAnimal.add(" HelloWorld ")；’在编译时都是没有问题的，但在执行时，‘for (Animal item ： arrayAnimal)’需要明确集合中的元素是Animal类型，而123是整型，HelloWorld是字符串型，这就会在运行到此处时报错，显然，这是典型的类型不匹配错

误，换句话说，ArrayList**不是类型安全的**。还有就是ArrayList对于存放值类型的数据，比如int、string型（string是一种拥有值类型特点的特殊引用类型）或者结构struct的数据，用ArrayList就意味着都需要将值类型装箱为Object对象，使用集合元素时，还需要执行拆箱操作，这就带来了很大的性能损耗。”

“等等，我不太懂，装箱和拆箱是什么意思？”

“所谓**装箱就是把值类型打包到Object引用类型的一个实例中**。比如整型变量 i 被‘装箱’并赋值给对象o。”

“所谓**拆箱就是指从对象中提取值类型**。此例中对象 o 拆箱并将其赋值给整型变量 i。”

```
int i = 123;
Object o = (Object)i;  //装箱 boxing
```

```
o = 123;
i = (int)o;  //拆箱 unboxing
```

“**相对于简单的赋值而言，装箱和拆箱过程需要进行大量的计算。对值类型进行装箱时，必须分配并构造一个全新的对象。其次，拆箱所需的强制转换也需要进行大量的计算**[MSDN]。总之，装箱拆箱是耗资源和时间的。而ArrayList集合在使用值类型数据时，其实就是在不断地做装箱和拆箱的工作，这显然是非常糟糕的事情。”

“啊，那从这点上来看，它还不如数组来得好了，因为数组事先就指定了数据类型，就不会有类型安全的问题，也不存在装箱和拆箱的事情了。看来它们各有利弊呀。”

“说得非常对，Java在5.0版之前的确也没什么好办法，但5.0出来后，就推出了新的技术来解决这个问题，那就是泛型。”

0.13 泛型

“**泛型是具有占位符（类型参数）的类、结构、接口和方法，这些占位符是类、结构、接口和方法所存储或使用的一个或多个类型的占位符。泛型集合类可以将类型参数用作它所存储的对象的类型的占位符；类型参数作为其字段的类型和其方法的参数类型**

出现。我读给你的是泛型定义的原话，听起有些抽象，我们直接来看例子。在Java 5.0后有ArrayList 类的泛型等效类，该类使用大小可按需动态增加的数组实现Collection泛型接口。其实用法上关键就是在ArrayList后面加‘<T>’，这个‘T’就是你需要指定的集合的数据或对象类型。”

```
import java.util.ArrayList;

public class Test {

    public static void main(String[] args){

        //声明泛型集合变量，在<>中声明Animal，意味着此集合只接受Animal对象
        ArrayList<Animal> arrayAnimal = new ArrayList<Animal>();

        arrayAnimal.add(new Cat("小花"));
        arrayAnimal.add(new Dog("阿毛"));
        arrayAnimal.add(new Dog("小黑"));
        arrayAnimal.add(new Cat("娇娇"));
        arrayAnimal.add(new Cat("咪咪"));

        //此时循环可以直接明确集合中都是Animal的item
        for(Animal item : arrayAnimal){
            System.out.println(item.shout());
        }

        System.out.println("动物个数："+arrayAnimal.size());

    }
}
```

“此时，如果你再写‘arrayAnimal.add(123);’或者‘arrayAnimal.add("HelloWorld");’结果将是？”

“哈，编译就报错，因为add的参数必须是要Animal或者Animal的子类型才行。”

“我是这样想的，其实ArrayList和ArrayList<T>在功能上是一样的，不同点就在于，它在声明和实例化时都需要指定其内部项的数据或对象类型，这就避免了刚才讲的类型安全问题和装箱拆箱的性能问题了。强，够强，怎么想到的，真是厉害。”

“是呀，也就是说，我们一开始就明确了集合这个‘箱子’只能装啥，这个就不需要再考虑混乱的问题了。不过显然Java语言的设计者也并不是一开始就明白这一点，也是通过实践和用户的反馈才在Java 5.0版中改进过来的。巨人也会有走弯路的时候，何况我们常人。**通常情况下，都建议使用泛型集合，因为这样可以获得类型安全的直接优点而不需要从基集合类型派生并实现类型特定的成员。此外，如果集合元素为值类型，泛型集合类型的性能通常优于对应的非泛型集合类型（并优于从非泛型基集合类型派生的类型），因为使用泛型时不必对元素进行装箱。**”

“当然是泛型好呀，它可是集早期的ArrayList集合和Array数组优点于一身的好东西，有了它，早期的ArrayList就显得太老土了。”

“至于泛型的知识还有很多，这里就不细讲了，你自己去找资料研究吧。”

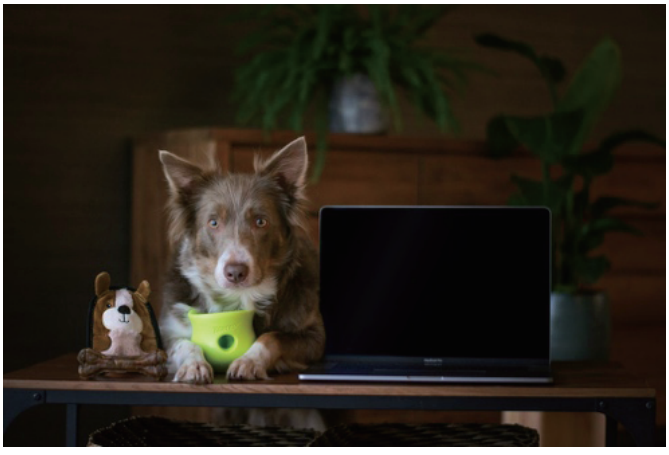
“好的好的，其实已经有些明白是怎么回事了。我自己去研究吧。”

0.14 客套

“要讲的东西太多了，我们的‘动物运动会’程序也只写了个开头，以后有的是机会。”小菜看了看表说，“现在都过了中午，食堂都快没菜了，走，我们先吃饭去吧。”

“好的，今天真的太感谢了，我觉得这半天的收获远远比上一个月课，看几本砖头书来得效果好呀。”史熙兴奋地说。

“哪里哪里，今天讲的都只是皮毛，要学习的内容还多着呢，不过话说回来，上午讲的这个未完成的‘动物运动会’的例子尽管简单，但却涵盖了面向对象的最重要的知识，你好好去体会一下。有机会我再跟你讲讲设计模式，你对封装、继承、多态的理解就会更深入一些，**学无止境，你需要不断地练习实践才可能真正成为优秀的软件工程师。**”



“嗯，我觉得我对编程有了很大的兴趣，面向对象的编程方式确实非常有意思。”

“师傅领进门，修行在个人，今后就看你的了，好好加油。不过现在我们还是先去为肚皮加点油哦。”

“对对对，走，我们去吃饭去。”

几个月后。研发总监对小菜的培训工作非常满意，准备提升小菜为技术培训经理，今后培训新员工都可以交给他做。小菜欣喜之余，也不觉感慨，如果不是两年前表哥大鸟的帮助，自己也不能有今天的成长。那段时间关于设计模式的学习经历，真是一段值得书写和回味的时光。