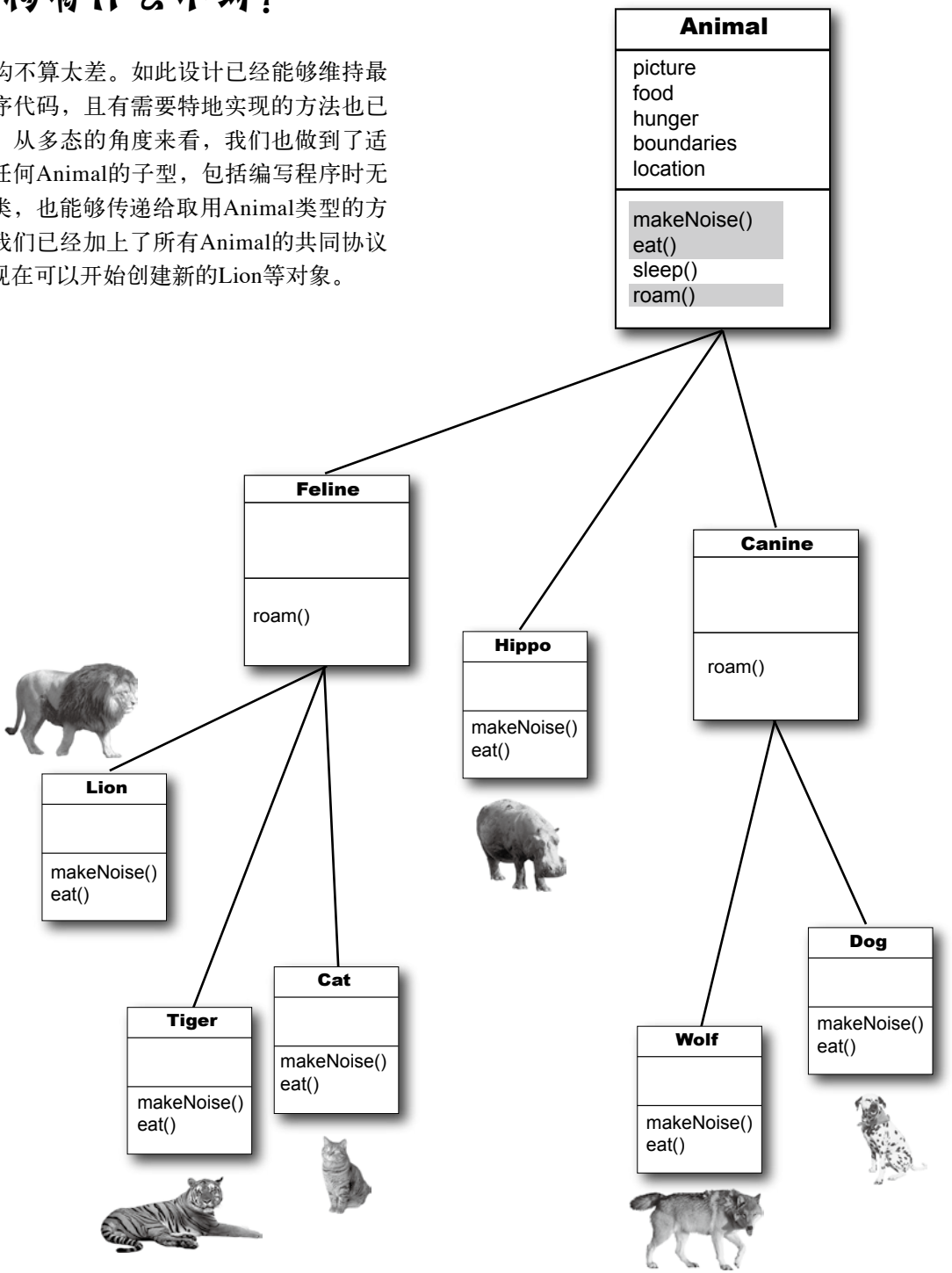
8 接口与抽象类

深入多态

继承只是个开始。要使用多态，我们还需要接口（这个接口的意义不是GUI的I所代表的接口）。我们需要超越简单的继承并前进到只有通过设计与编写接口规格才能达成的适应性与扩展性。很多Java功能若没有接口就无法工作，因此就算你不会去设计接口，也还是会使用到。最后你会怀疑如果没有接口机制要怎么活下去。到底接口是什么呢？它是一种100%纯抽象的类。什么是抽象类？它是无法初始化的类。抽象类有什么用途？马上就会告诉你。前一章让Vet这个方法能够运用Animal的所有子类只是多态的基本招式而已。接口是多态和Java的重点。

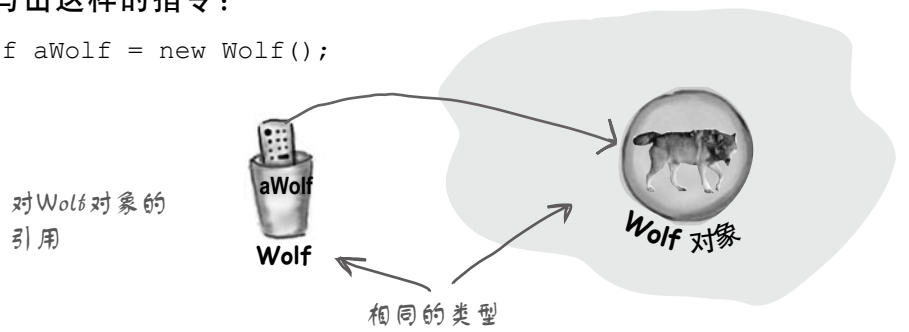
这个结构有什么不对？

这个class结构不算太差。如此设计已经能够维持最少的重复程序代码，且有需要特地实现的方法也已经被覆盖过。从多态的角度来看，我们也做到了适应性，所以任何Animal的子型，包括编写程序时无法想象的种类，也能够传递给取用Animal类型的方法来执行。我们已经加上了所有Animal的共同协议到父类上，现在可以开始创建新的Lion等对象。



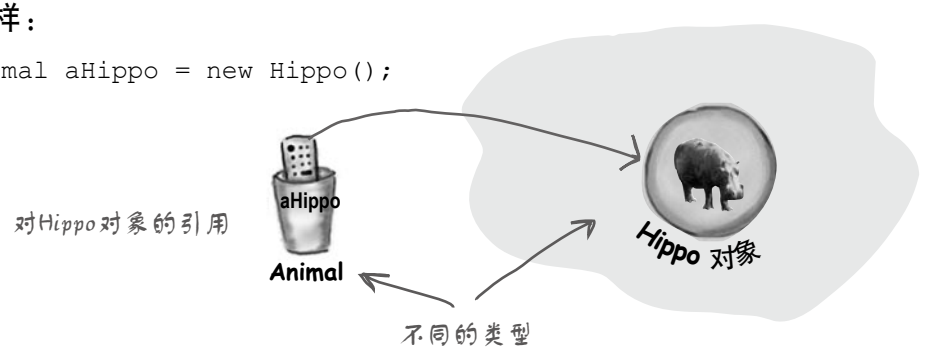
我们可以写出这样的指令：

```
Wolf aWolf = new Wolf();
```



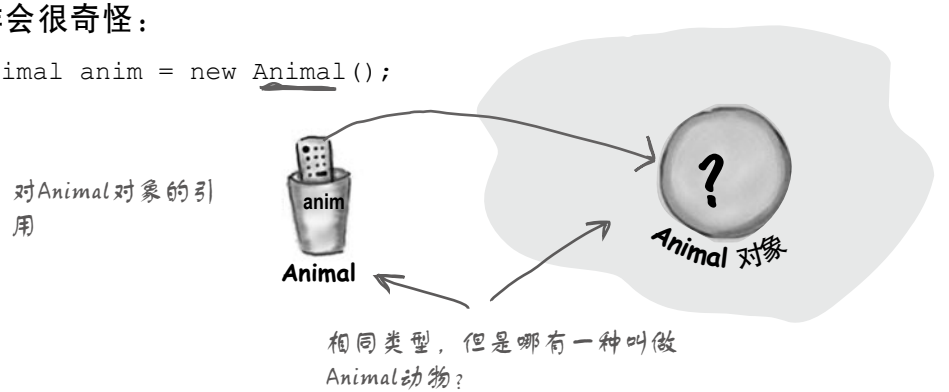
也可以这样：

```
Animal aHippo = new Hippo();
```

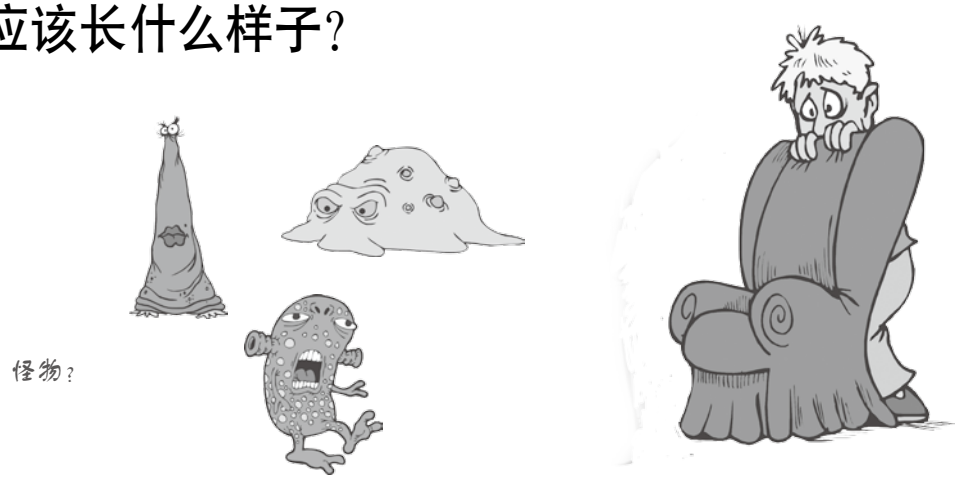


但是这样会很奇怪：

```
Animal anim = new Animal();
```



Animal对象应该长什么样子？



实例变量的值会是……？

有些类不应该被初始化！

创建出Wolf对象或Hippo对象或Tiger对象是很合理的，但是Animal对象呢？它应该长什么样子？什么颜色、大小、几条腿？

尝试创建出Animal类型的对象就好像出人意料之外，在传送组合的过程中发生了一点问题。

那要如何处理这个问题呢？我们一定要有Animal这个类来继承和产生多态。但是要限制只有它的子类才能够被初始化。我们要的是Lion、Hippo对象，而不是Animal对象。

幸好，有个方法能够防止类被初始化。换句话说，就是让这个类不能被“new”出来。通过标记类为抽象类的，编译器就知道不管在哪里，这个类就是不能创建任何类型的实例。

你还是可以用这种抽象的类型作为引用类型。这也就是当初为何要有抽象类型的目的。

当你设计好继承结构时，你必须要决定哪些类是抽象的，而哪些是具体的。具体的类是实际可以被初始化为对象的。

设计抽象的类很简单 —— 在类的声明前面加上抽象类关键词abstract就好：

```
abstract class Canine extends Animal {

    public void roam() { }

}
```

编译器不会让你初始化抽象类

抽象类代表没有人能够创建出该类的实例。你还是可以使用抽象类来声明为引用类型给多态使用，却不用担心哪个创建该类型的对象，编译器会确保这件事。

```
abstract public class Canine extends Animal
{
   public void roam() { }
}
-------------------------------------------
public class MakeCanine {
   public void go() {
      Canine c;
      c = new Dog();
      c = new Canine();
      c.roam();
    }
}
```

这是可以的，因为你可以赋值子类对象给父类的引用，即使父类是抽象的

这个类已经被标记为abstract，所以过不了编译器这一关

```
File Edit Window Help BeamMeUp
% javac MakeCanine.java
MakeCanine.java:5: Canine is abstract;
cannot be instantiated
      c = new Canine();
          ^
1 error
```

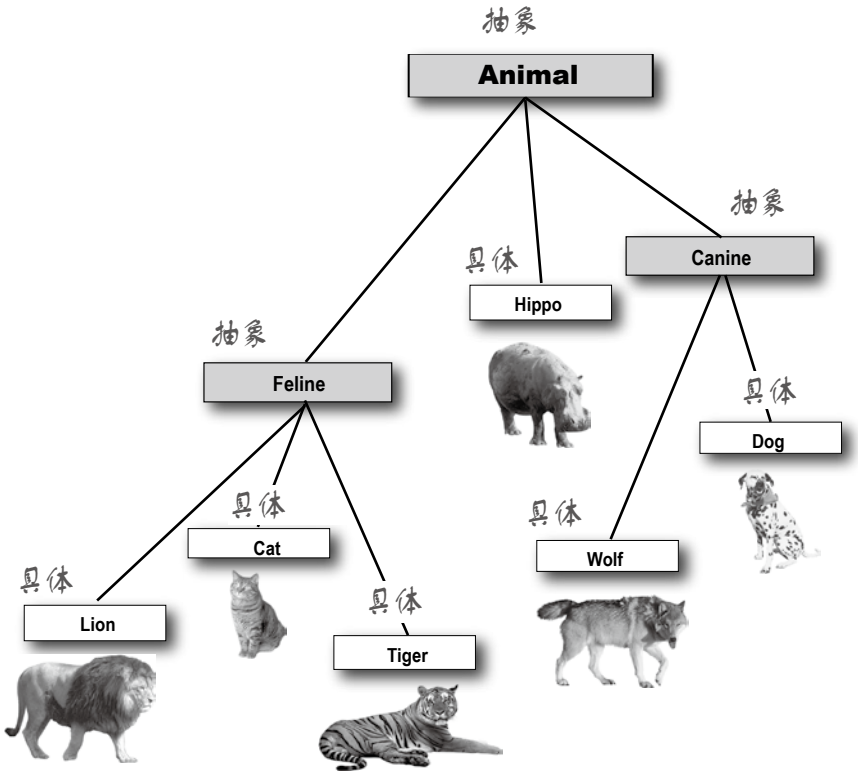
抽象类除了被继承过之外，是没有用途、没有值、没有目的*。

*除了抽象的class可以有static的成员之外，见第10章。

抽象与具体

不是抽象的类就被称为具体类。在Animal的继承树下，如果我们创建出Animal、Canine与Feline的抽象，则Hippo、Wolf等就是具体的类。

查阅Java API你就会发现其中有很多的抽象类，特别是GUI的函数库中更多。GUI的组件类是按钮、滚动条等与GUI有关类的父类。你只会对组件下的具体子类作初始化动作。



抽象或具体?

你怎么知道某个类应该是抽象的？饮料或许是抽象的。那大雕参茸或威士忌是否也应该是抽象的？在继承层次中从哪个点开始才算是具体的？

你会把“提神饮料”设计成具体，还是说它也是个抽象？看起来“保力达 B”才会是具体的。你认为呢？

观察上面的Animal继承层次，这些抽象或具体的决定是否合适呢？你会修改这个层次吗？

抽象的方法

除了类之外，你也可以将方法标记为abstract的。抽象的类代表此类必须要被extend过，抽象的方法代表此方法一定要被覆盖过。你或许认为抽象类中的某些行为在没有特定的运行时不会有任何的意义。也就是说，没有任何通用的实现是可行的。想象一下通用的eat()方法会有什么结果？

抽象的方法没有实体！

因为你已经知道编写出抽象方法的程序代码没有意义，所以不会含有方法。

```
public abstract void eat();
```

没有方法体！
直接以分号结束

如果你声明出一个抽象的方法，就必须将类也标记为抽象的。你不能在非抽象类中拥有抽象方法。

就算只有一个抽象的方法，此类也必须标记为抽象的。

there are no Dumb Questions

问： **为什么要有抽象的方法？我认为抽象类的重点就在于可以被子类继承的共同程序代码。**

答： 将可继承的方法体（也就是有内容的方法）放在父类中是个好主意。但有时就是没有办法作出给任何子类都有意义的共同程序代码。抽象方法的意义是就算无法实现出方法的内容，但还是可以定义出一组子型共同的协议。

问： **这样做的好处是……？**

答： 就是多态！记住，你想达成的目标是要使用父型作为方法的参数、返回类型或数组的类型。通过这个机制，你可以加入新的子型到程序中，却又不必重写或修改处理这些类型的程序。想象一下如果不是使用Animal作为Vet的方法参数程序会写成什么样子……你必须为每一种动物写出不同的方法！因此多态的好处就在于所有子型都会有那些抽象的方法。

你必须实现所有抽象的方法

实现抽象的方法就如同覆盖过方法一样

抽象的方法没有内容，它只是为了标记出多态而存在。这表示在继承树结构下的第一个具体类必须要实现出所有的抽象方法。

然而你还是可以通过抽象机制将实现的负担转给下层。例如说将Animal与Canine都标记为abstract，则Canine就无需实现出Animal的抽象方法。但具体的类，例如说Dog，就得实现出Animal和Canine的抽象方法。

记得抽象类可以带有抽象和非抽象的方法，因此Canine也可以实现Animal的抽象方法，让Dog不必实现这个部分。如果Canine没有对Animal的抽象类表示出任何意见，就表示Dog得自己实现出Animal的抽象方法。

当我们谈到“你必须实现所有抽象的方法”时，表示说你必须写出内容。你必须以相同的方法鉴名（名称与参数）和相容的返回类型创建出非抽象的方法。Java很注重你的具体子类有没有实现这些方法。

抽象类与具体类

让我们来对抽象修辞学产生些具体的用途。在中间这行列出一些类。你的任务是想象有哪些应用程序会把它们设计成具体的，又有哪些应用程序会把它们设计成抽象的。我们已经举出几个例子，例如Tree这个类在植物看护应用程序中应该是抽象的，而在高尔夫球场仿真程序中应该会是具体的。

具体	类	抽象
高尔夫模拟	**Tree**	植物看护
_____	**House**	建筑设计程序
卫星影像程序	**Town**	_____
_____	**Football Player**	教练程序
_____	**Chair**	_____
_____	**Customer**	_____
_____	**Sales Order**	_____
_____	**Book**	_____
_____	**Store**	_____
_____	**Supplier**	_____
_____	**Golf Club**	_____
_____	**Carburetor**	_____
_____	**Oven**	_____

多态的使用

假设我们不知道有ArrayList这种类而想要自行编写维护list的类以保存Dog对象。在第一轮我们只会写出add()方法。我们使用大小为5的简单Dog数组（Dog[]）来保存新加入的Dog对象。当Dog对象超过5个时，你还是可以调用add()方法，但是什么事情也不会发生。如果没有越界，add()会把Dog装到可用的数组位置中，然后递增可用索引（nextIndex）。

自己创建的Dog专用list

（这或许是有史以来自行尝试编写ArrayList类型类中最差劲的一个程序）


```
public class MyDogList {
    private Dog [] dogs = new Dog[5];
    private int nextIndex = 0;

    public void add(Dog d) {
       if (nextIndex < dogs.length) {
          dogs[nextIndex] = d;
          System.out.println("Dog added at " + nextIndex);
          nextIndex++;
       }
    }
}
```

实际上使用的是数组

增加新的Dog就加1

如果没有超出上限就把Dog加进去并列出信息

递增计数

糟了，也要写出Cat用的

我们有几个选项：

（1）另外创建一个单独的MyCatList类来处理Cat对象，这不好。

（2）创建一个单独的DogAndCatList类，用 addCat(Cat c) 与addDog(Dog d) 来同时处理两个不同的数组实例，这也不好。

（3）编写一个不同的AnimalList类让它处理Animal所有的子类。这应该是最好的办法，所以我们就这么处理，以更通用的Animal来取代个别的子类。程序变更得关键部分有特别标出来（逻辑还是一样，只是把Dog换成Animal）。

自己创建的Animal通用list

version 2

别紧张，这不是在创建Animal对象，只是个保存Animal的数组对象

```
public class MyAnimalList {

    private Animal[] animals = new Animal[5];
    private int nextIndex = 0;

    public void add(Animal a) {
       if (nextIndex < animals.length) {
          animals[nextIndex] = a;
          System.out.println("Animal added at " + nextIndex);
          nextIndex++;

       }
    }
}
```

```
public class AnimalTestDrive{
   public static void main (String[] args) {
      MyAnimalList list = new MyAnimalList();
      Dog a = new Dog();
      Cat c = new Cat();
      list.add(a);
      list.add(c);
   }
}
```

```
File Edit Window Help Harm
% java AnimalTestDrive
Animal added at 0
Animal added at 1
```

非Animal呢？何不写个万用类？

你知道这要怎么做。我们可以修改数组的类型，并且调整add()方法的参数，以处理Animal之上的类。那便是更通用、更抽象的一种类。但是真的有这种类吗？我们设计的Animal并没有 父类不是吗？

事实上是有的。

还记得ArrayList的方法吗？它们是通过对象这个类型来操纵所有类型的对象。

version 3 （这只是部分的ArrayList中的方法）

ArrayList

boolean remove(Object elem)
根据索引参数移动对象，如果list中没有元素返回true

boolean contains(Object elem)
如果和对象的参数相匹配的的话返回true

boolean isEmpty()
如果list中没有元素返回true

int indexOf(Object elem)
返回对象参数的索引值或-1

Object get(int index)
返回元素在list中的位置

boolean add(Object elem)
向list中增加元素

// more

许多ArrayList的方法都用到Object这个终极类型。因为每个类都是对象的子类，所以ArrayList可以处理任何类！

注意：Java 5.0的get()和add()方法与这里所显示的有所不同，但无损于此处要表达的概念。

在Java中的所有类都是从Object这个类继承出来的。

Object这个类是所有类的源头；它是所有类的父类。

如果Java中没有共同的父类，那将无法让Java的开发人员创建出可以处理自定义类型的类，也就是说无法写出像ArrayList这样可以处理各种类的类。

就算你不知道，但实际上所有的类都是从对象给继承出来的。你可以把自己写的类想象成是这样声明的：

```
public class Dog extends Object { }
```

但是Dog本来是从Canine给extends出来的啊！没关系，编译器会知道改成让Canine去继承对象。事实上是Animal去继承对象。

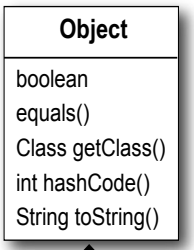
没有直接继承过其他类的类会是隐含的继承对象。

所以就算Dog或Canine没有直接extend对象，还是会通过Animal来继承对象。

终极对象有什么？

如果你是Java，那你会想要让每个对象都带有什么行为？嗯……来个可以判断某对象是否与其他对象相等的方法如何？再加上一个可以说明它是什么类的方法怎样？或许还会需要一个产生对象哈希代码的方法？你可以运用哈希表上的对象（我们会在第17章和附录B中讨论哈希表）。

你知道怎样吗？对象的确有上面所说的方法。那还不是全部的方法，但目前我们只关心这几个。



只列出类对象的一部分方法

所有的类都有继承这个类

① equals(Object o)

```
Dog a = new Dog();
Cat c = new Cat();

if (a.equals(c)) {
    System.out.println("true");
} else {
    System.out.println("false");
}
```

```
File Edit Window Help Stop
% java TestObject
false
```

这会让你知道是否两个对象可认为是“相等”的（见附录B）

② getClass()

```
Cat c = new Cat();
System.out.println(c.getClass());
```

```
File Edit Window Help Faint
% java TestObject
class Cat
```

告诉你此对象是从哪里被初始化的

③ hashCode()

```
Cat c = new Cat();
System.out.println(c.hashCode());
```

```
File Edit Window Help Drop
% java TestObject
8202111
```

列出此对象的哈希代码，你可以把它想成是一个唯一的ID

④ toString()

```
Cat c = new Cat();
System.out.println(c.toString());
```

```
File Edit Window Help LapseIntoComa
% java TestObject
Cat@7d277f
```

列出类的名称和一个我们不关心的数字

there are no Dumb Questions

问： Object这个类是抽象的吗？

答： 不是。至少不是正式的Java抽象类。因为它可以被所有类继承下来的方法都实现程序代码，所以没有必须被覆盖过的方法。

问： 那是否可以覆盖过Object的方法？

答： 部分可以。但是有些被标记为final，这代表你不能覆盖掉它们。强烈建议你用自己写的类去覆盖掉hastCode()、equals()以及toString()。

问： 如果ArrayList方法是通用的，那ArrayList <DotCom>是什么意思？

答： 限制它的类型。在Java 5.0之前无法限制它的类型。如果你写成ArrayList<Dog>，则此ArrayList受限只能保存Dog的对象。这种新型的语法会在后面的章节有更多的说明。

问： Object类是具体的。怎么会允许有人去创建Object的对象呢？这不就跟Animal对象一样不合理吗？

答： 好问题！为何要允许创建出Object的实例呢？因为有时候你就是会需要一个通用的对象，一个轻量化的对象。它最常见的用途是用在线程的同步化上面（见第15章）。你先当作不会用到这个对象。

问： 所以Object的主要目的是提供多态的参数与返回类型吗？

答： 这个Object类有两个主要的目的：作为多态让方法可以应付多种类型的机制，以及提供Java在执行期对任何对象都有需要的方法的实现程序代码（让所有的类都会继承到）。有一部分的方法是与线程有关，这会在后面的章节说明。

问： 即然多态类型这么有用，为什么不把所有的参数和返回类型都设定成Object类型哪？

答： 啊……想想看这会发生什么后果。考虑一下何谓“类型安全检查”。它是Java保护程序代码的一项重要机制。在此机制下，你不会意外地要求对象执行错误类型的动作。例如说防止你对Cefiro要求Ferrai的加速动作。

但事实上，你也不用担心会发生这件事，因为当某个对象是以Object类型来引用时，Java会把它当作Object类型的实例。这代表你只能调用由Object类中所声明的方法。若你像下面这样做：

```
Object o = new Ferrai( );
o.goFast( );//非法
```

则第二行会无法通过编译。

因为Java是类型检查很强的程序语言，编译器会检查你调用的是否是该对象确实可以响应的方法。换句话说，你只能从确实有该方法的类去调用。同样，这也会在后面的章节说明。

使用Object类型的多态引用是会付出代价的……

在你开始以Object类型使用所有超适用性参数和返回类型之前，你应该要考虑到使用Object类型作为引用的一些问题。注意此处的讨论不涉及制作出Object类型的实例，这是在说以Object类型作为引用的其他类型。

当你把对象装进ArrayList<Dog>时，它会被当作Dog来输入与输出：

```
ArrayList<Dog> myDogArrayList = new ArrayList<Dog>();
Dog aDog = new Dog();
myDogArrayList.add(aDog);
Dog d = myDogArrayList.get(0);
```

保存Dog的ArrayList

新建一个Dog

装到ArrayList中

将Dog赋值给新的Dog引用变量

但若你把它声明成ArrayList<Object>时会怎样？如果你打算创建出一个可以保存任何一种 对象的ArrayList 时，你会如此的声明：

```
ArrayList<Object> myDogArrayList = new ArrayList<Object>();
Dog aDog = new Dog();
myDogArrayList.add(aDog);
```

保存对象的ArrayList

新建一个Dog

装到ArrayList中

相同的步骤

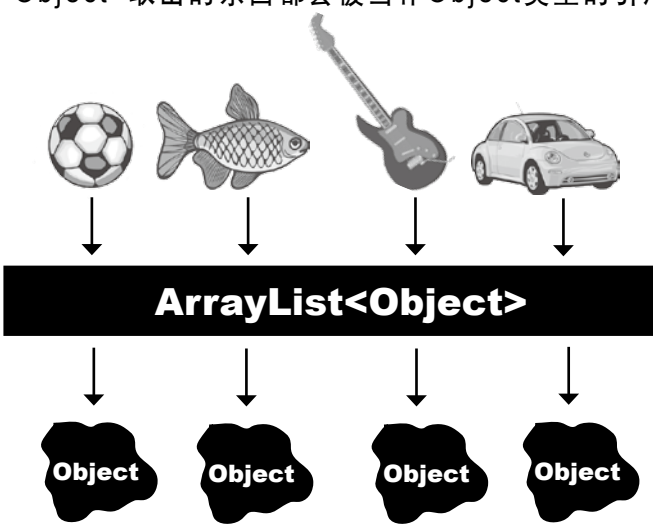
如果是这样，当你尝试要把Dog对象取出并赋值给Dog的引用时会发生什么事？

```
Dog d = myDogArrayList.get(0);
```

无法通过编译！对ArrayList<Object>调用get()方法会返回Object类型，编译器无法确认它是Dog！

任何从ArrayList<Object>取出的东西都会被当作Object类型的引用而不管它原来是什么。

放进去的对象原来是足球、鱼、吉他和汽车



出来后都变成Object

从ArrayList<Object>取出的Object都会被当作是Object这个类的实例。编译器无法将此对象识别为Object以外的事物。

当Dog不再是Dog时

问题在于把所有东西都以多态来当作是Object会让对象看起来失去了真正的本质（但不是永久性的）。Dog似乎失去了犬性。让我们来看一下当我们传入一个Dog给会返回同一个Dog对象的类型引用的方法时会有什么反应。

BAD

```
public void go() {
   Dog aDog = new Dog();
   Dog sameDog = getObject(aDog);
}

public Object getObject(Object o) {
   return o;
}
```

无法过关！虽然这个方法会返回同一个Dog，但编译器认为这只能赋值给Object类型的变量

返回同一个引用，但是类型已经转换成Object了

```
File Edit Window Help Remember
DogPolyTest.java:10: incompatible types
found   : java.lang.Object
required: Dog
      Dog sameDog = getObjects(aDog);
1 error                          ^
```

编译器无法得知方法返回的其实是Dog，因此不会同意这项赋值

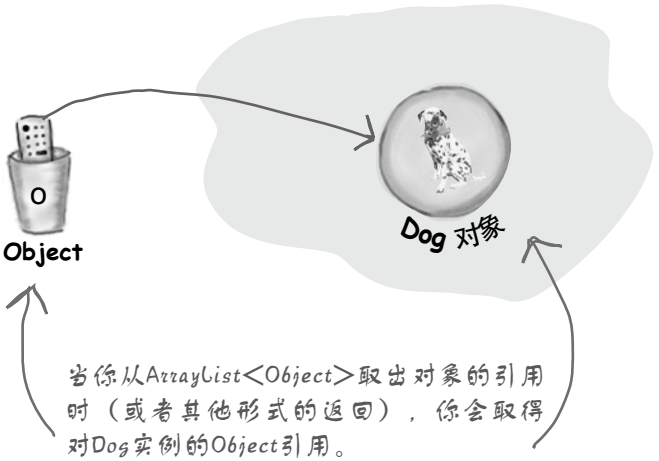
GOOD

```
public void go() {
   Dog aDog = new Dog();
   Object sameDog = getObject(aDog);
}

public Object getObject(Object o) {
   return o;

}
```

这样会过关，因为你可以赋值任何东西给Object类型的引用，并且每个东西都能对Object通过IS-A测试。

Object不会吠

我们已经知道当一个对象被声明为Object类型的对象所引用时，它无法再赋值给原来类型的变量。我们也知道这会发生在返回类型被声明为Object类型的时候，例如前面所提过的ArrayList<Object>。但这意味着什么呢？使用Object引用变量来引用Dog对象会是个问题吗？让我们试着对被编译器认为是Object的Dog调用Dog才有的方法看看：




```
Object o = al.get(index);
int i = o.hashCode();
o.bark();
```

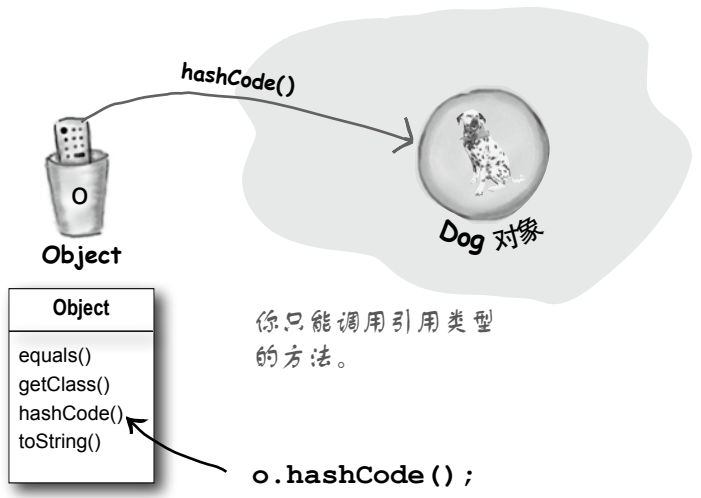
没问题。Object本来就有hashCode()这个方法

有问题！

不能这么做！Object根本就不知道什么是bark()。就算天知地知你知我知但编译器就是不知……

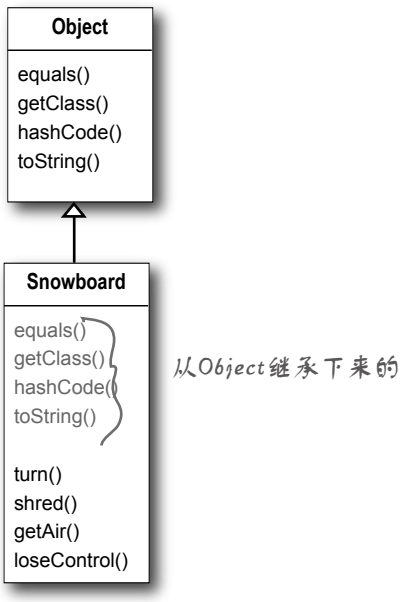
编译器是根据引用类型来判断有哪些method可以调用，而不是根据Object确实的类型。

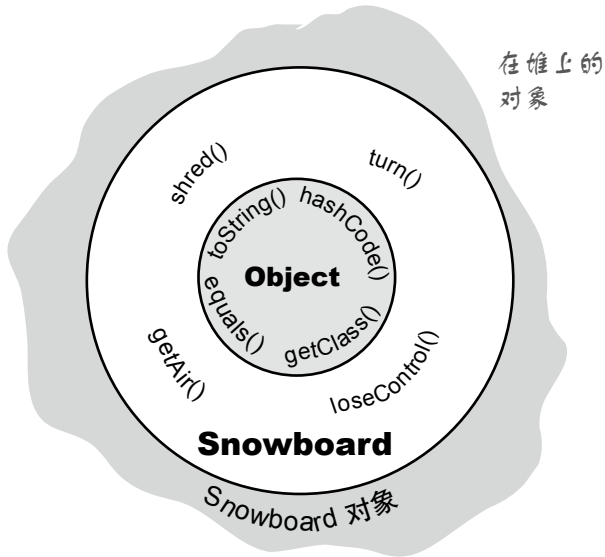
就算你知道对象有这个功能，编译器还是会把它当作一般的Object来看待。编译器只管引用的类型，而不是对象的类型。



探索内部Object

对象会带有从父类继承下来的所有东西。这代表每个对象，不论实际类型，也会是个Object的实例。所以Java中的每个对象除了真正的类型外，也可以当作是Object来处理。当你执行new Snowboard()命令时，除了在堆上会有一个Snowboard对象外，此对象也包含了一个Object在里面。





这在堆上只有一个对象，但它实际上带着Snowboard与Object的内容

“多态”意味着“很多形式”

你可以把Snowboard当作Snowboard或者Object

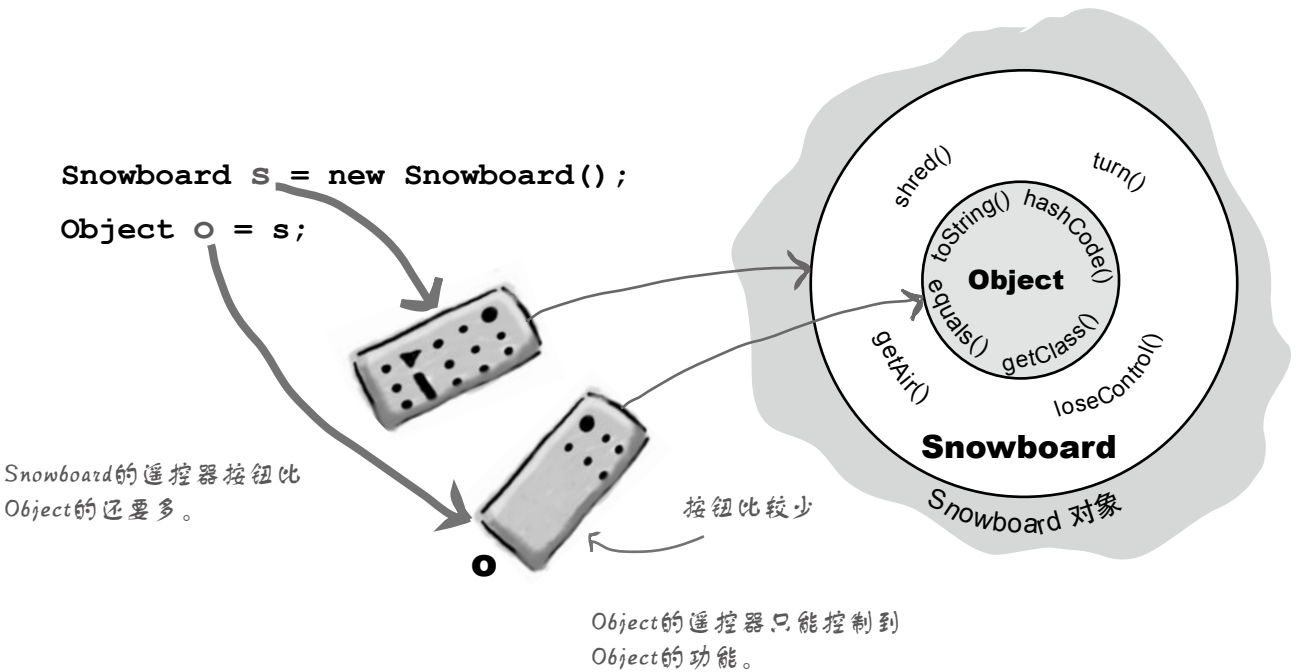
如果引用是个遥控器，则当你在继承树往下走时，会发现遥控器的按钮越来越多。Object的遥控器只有几个按钮而已，但Snowboard的遥控器就会包含有来自Object和自己定义的按钮。越接近具体的类会有越多的按钮。

当然这也不是绝对的，子类也有可能不会加入任何新的方法，而只是覆盖过一些方法罢了。重点在于如果对象的类型是Snowboard，而引用它的却是Object，则它不能调用Snowboard的方法。

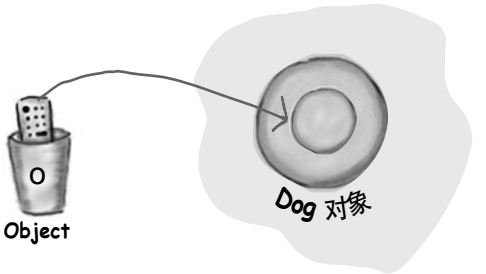
当你把对象装进ArrayList<Object>时，不管它原来是什么，你只能把它当作是Object。

从ArrayList<Object>取出引用时，引用的类型只会是Object。

这代表你只会取得Object的遥控器。



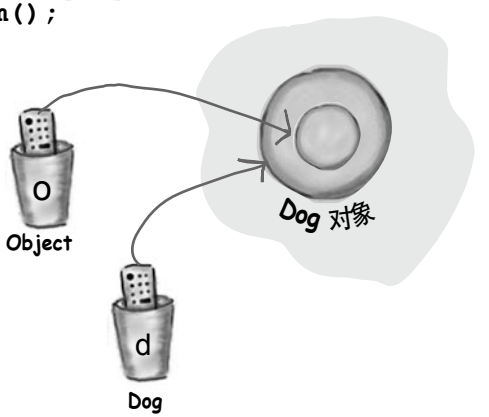
转换回原来的类型



它还是个Dog对象，但如果你想要调用Dog特有的方法，就必须要将类型声明为Dog。如果你真的确定它是个Dog，那么你就可以从Object中拷贝出一个Dog引用，并且赋值给Dog引用变量。

```
Object o = al.get(index);
Dog d = (Dog) o;
d.roam();
```

将类型转换成Dog



如果不能确定它是Dog，你可以使用instanceof这个运算符来检查。若是类型转换错了，你会在执行期遇到ClassCastException异常并且终止。

```
if (o instanceof Dog) {
   Dog d = (Dog) o;
}
```

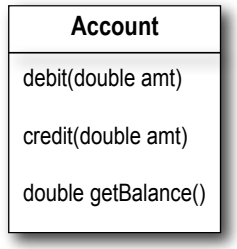
现在你知道Java是多么注重引用变量的类型。

你只能在引用变量的类确实有该方法才能调用它。

把类的公有方法当作是合约的内容，合约是你对其他程序的承诺协议。

在编写类时，大多数情况下你一定会显露出一些方法给类以外的程序使用。要让方法显露就代表你会让方法能够存取得到，通常这会通过标记成公有来完成。

想象这样的情境：你在编写一个小型会计总账系统给“猪标服装社”使用。你发现有个称为Account的类已经写好并且符合你的需求（上一次帮“宏昌水电行”开发程序时写出来的），因此就把它拿过来用。



它有一个debit()和credit()方法可以用来执行会计的借贷项目，还有getBlance()方法可以计算账户。

所以你可以声明一个变量a引用到Account的实例，然后通过圆点运算符调用a.debit()或a.credit()等。

因为类的合约是这么保证的，所以你在这个类的实例上面一定能找到这些方法。

万一想要修改合约呢？

好吧，假如你是个Dog（但是千万别去闻另外一个 Dog 的屁股，这样很不礼貌），你会发现不只有一份合约，你还继承了所有父类传递下来的方法。

类Canine中的所有元素是你合约中的一部分。

类Animal中的所有元素是你合约中的一部分。

类Object中的所有元素是你合约中的一部分。

根据IS-A测试，你就会是Canine、Animal和Object。

如果有人要编写类似的程序，你大可把定义好的class交给他使用。

但是，如果他还要加上亲热或耍宝等宠物特有的功能要怎么办呢？

现在假设你是设计Dog类的程序设计师。没问题吧？你可以直接把beFriendly()和play()这两个方法加进Dog这个类中。这样做不会让其他用到Dog的程序产生问题，因为你没有更改到其他现有的方法。

你觉得这样的做法（把Pet的方法直接加到Dog上）有没有缺点？

如果你是Dog类的程序设计师，且必须修改Dog类以让它能够执行Pet的动作，那你会怎么办？我们知道直接加入Pet的方法是可行的，并且这也不会对其他程序有影响。

但若Cat也要有Pet的功能怎么办？先不管Java的功能，想象一下你要怎样让Animal可以选择性地带有Pet的行为又不会强迫让狮子老虎都表现成宠物？

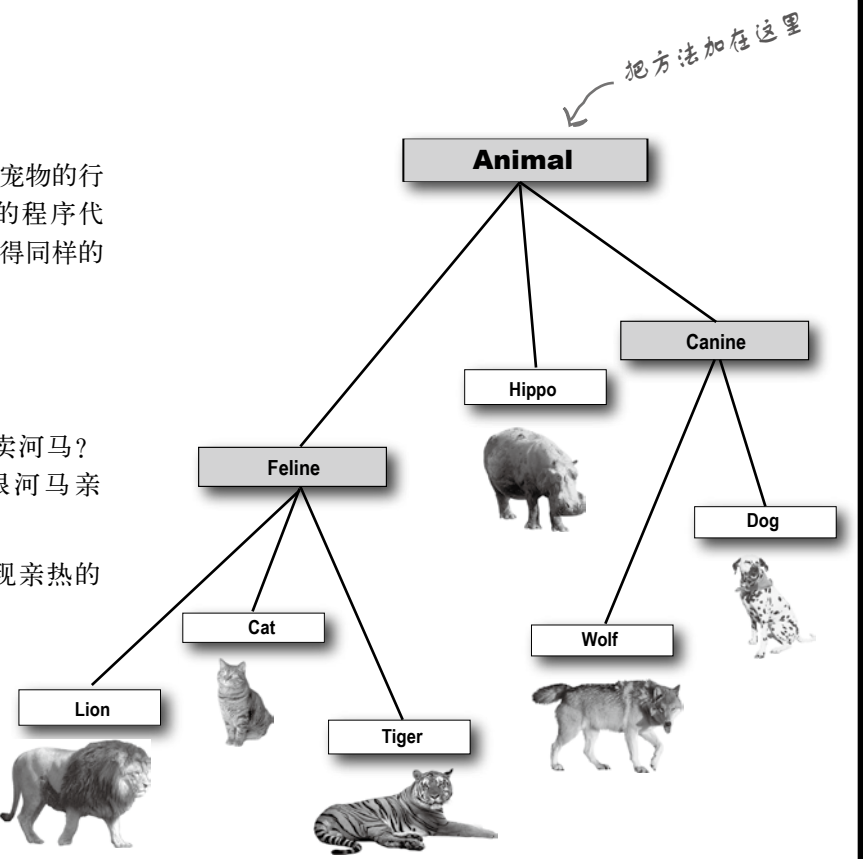
停下来想想看，动点脑筋会帮助你消耗能量。

有哪些方法可以在PetShop程序中重用现有的类？

在接下来的几页中，我们会对每种可能的方法逐个介绍。先不用管Java实际的功能是怎么做的。一旦知道各种方法的好处与坏处之后，我们就会知道怎么办了。

① 方法一

采用最简单的做法，把宠物方法加进Animal类中。



优点：

所有的动物马上就可以继承宠物的行为。不需要改变所有子类的程序代码，而新增加的动物也会取得同样的行为。

缺点：

你什么时候看过宠物店贩卖河马？又是什么时候看到饲主跟河马亲热，带河马去公园散步？

并且我们也知道狗跟猫表现亲热的方式不太一样啊。

② 方法二

采用方法一，但是把宠物的方法设定成抽象的，强迫每个动物子类覆盖它们。

优点：

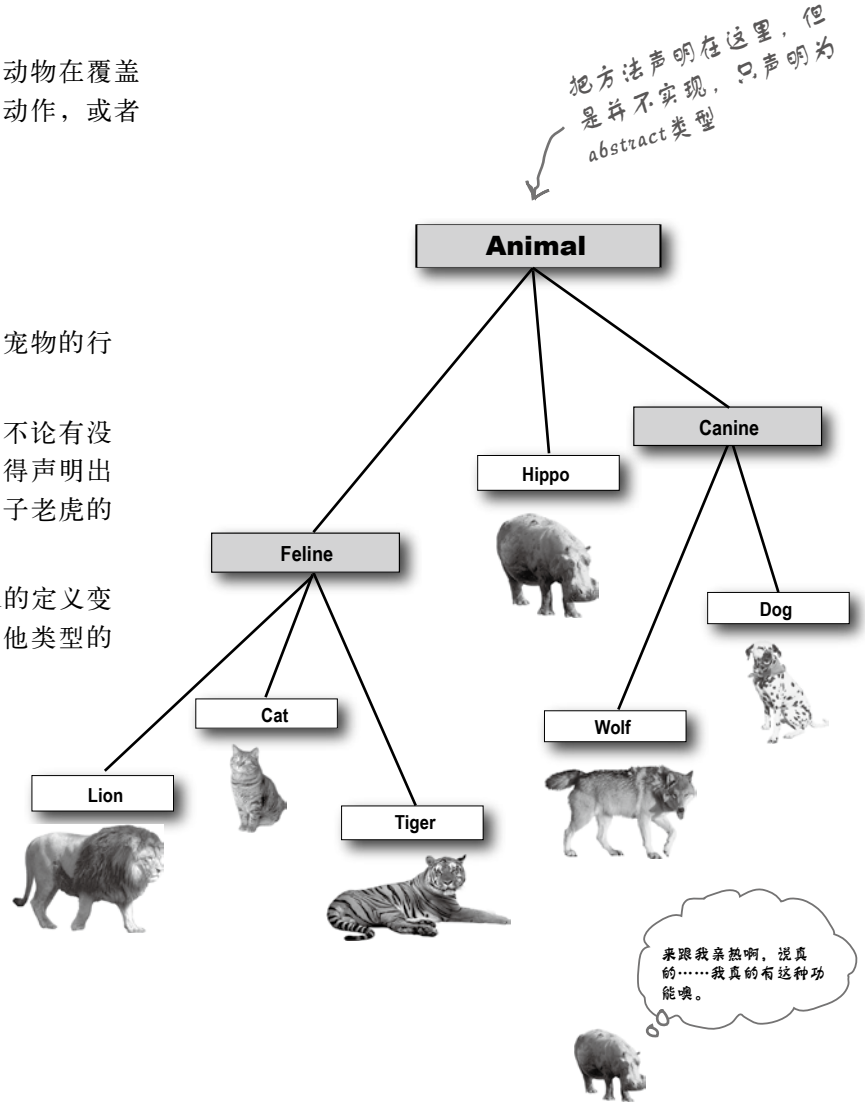
这样就可以让非宠物类的动物在覆盖这些方法时，作出合理的动作，或者是什么也不作。

缺点：

所有具体的动物都得实现宠物的行为，这样很浪费时间。

并且这种合约不太理想，不论有没有实质的行为，非宠物也得声明出有宠物行为的外观，对狮子老虎的尊严是个打击。

最重要的是这会让Animal的定义变得有些局限性，反而让其他类型的程序更难以重复利用。



③ 方法三

把方法加到需要的地方。

优点：

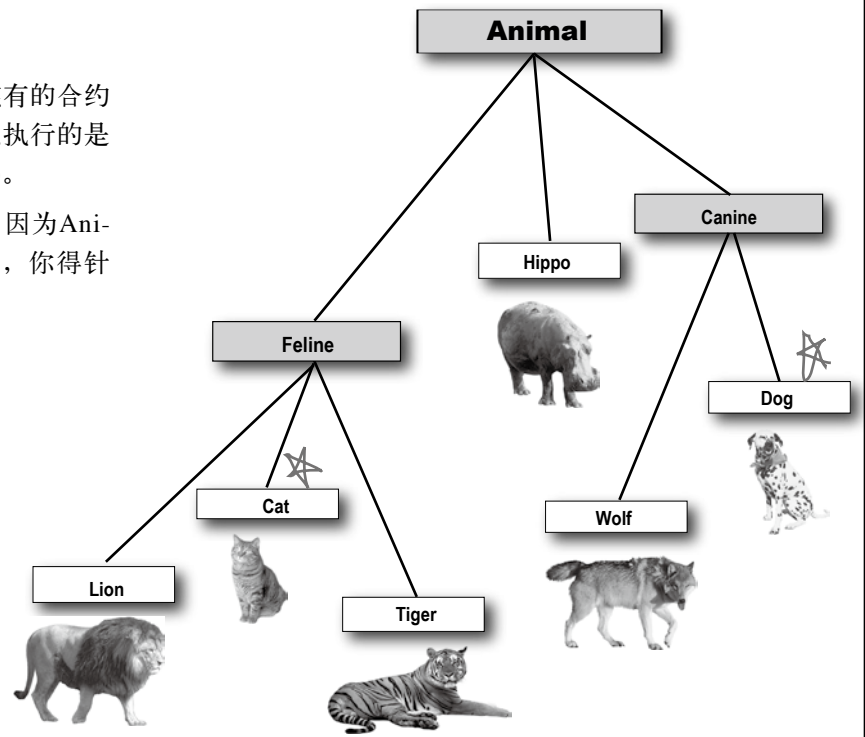
不必担心如何跟河马亲热。只有宠物才会有宠物的行为。

把方法写在各个类中

缺点：

首先，这样就会失去了物该有的合约保证。你无法确定宠物可以执行的是doFriendly()还是beFriendly()。

其次，多态将无法起作用，因为Animal不会有共同的宠物行为，你得针对个别宠物设计程序。



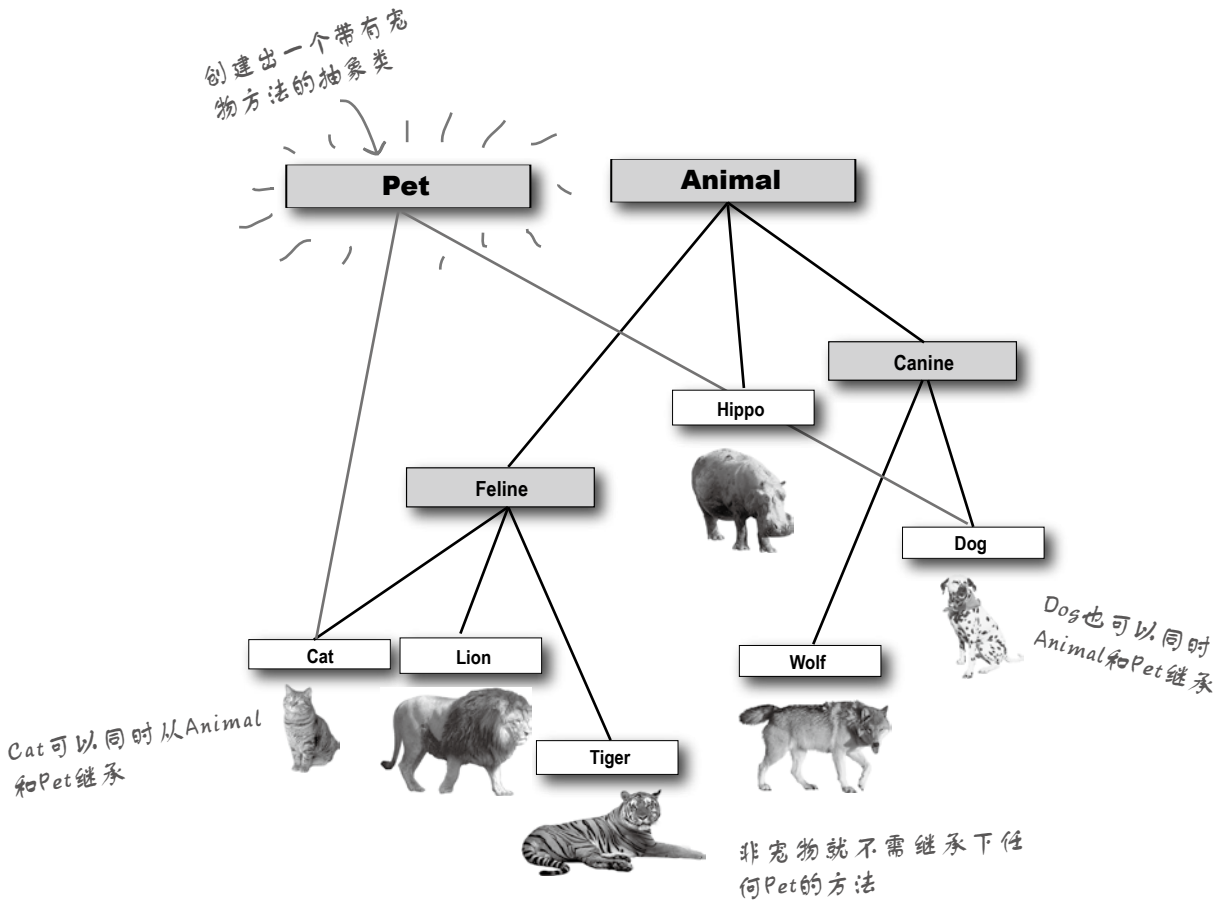
所以我们真正需要的方法是：

(1) 一种可以让宠物行为只应用在宠物身上的方法。

(2) 一种确保所有宠物的类都有相同的方法定义的方法。

(3) 一种是可以运用到多态的方法。

看起来，我们需要两个上层的父类



“两个父类”这个主意有一个问题……

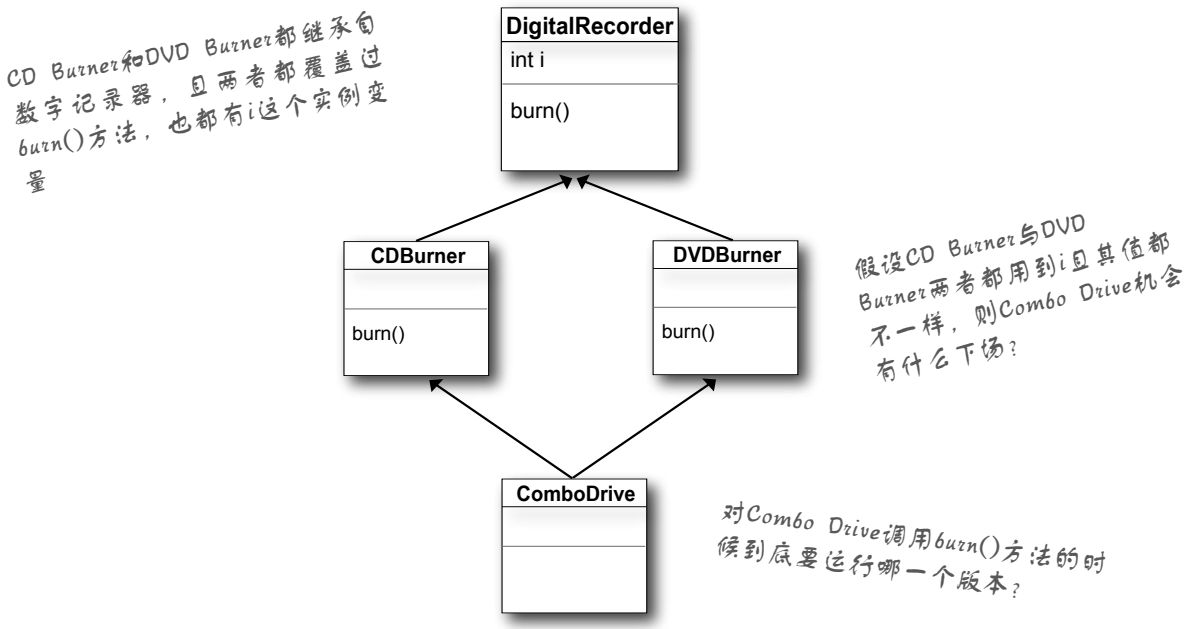
这种“多重继承”可能会很差。

其实Java不支持这种方式，

因为多重继承会有称为“致命方块”的问题

“致命方块”

（因为这个形状看起来就像扑克牌的方块）



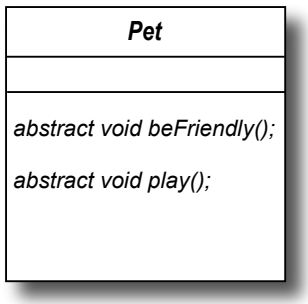
允许致命方块的程序语言会产生某种很糟糕的复杂性问题，因为你必须要有某种规则来处理可能出现的模糊性。额外的规则意味着你必须同时学习这些规则与观察适用这些规则的特殊状况。因此 Java 基于简单化的原则而不允许这种致命方块的出现。好吧，问题还是没有解决……

接口是我们的救星！

Java有个解决方案，使用接口。此处所讨论的不是GUI的接口，也不是“沟通管道”或“存取途径”的接口，我们说的是Java的interface关键词。

此接口可以用来解决多重继承的问题却又不会产生致命方块这种问题。

接口解决致命方块的办法很简单：把全部的方法设为抽象的！如此一来，子类就得要实现此方法，因此Java虚拟机在执行期间就不会搞不清楚要用哪一个继承版本。



Java的接口就好像是100%的纯抽象类。

所有接口的方法都是抽象的，所以任何Pet的类都必须要实现这些方法

接口的定义：

```
public interface Pet {...}
```

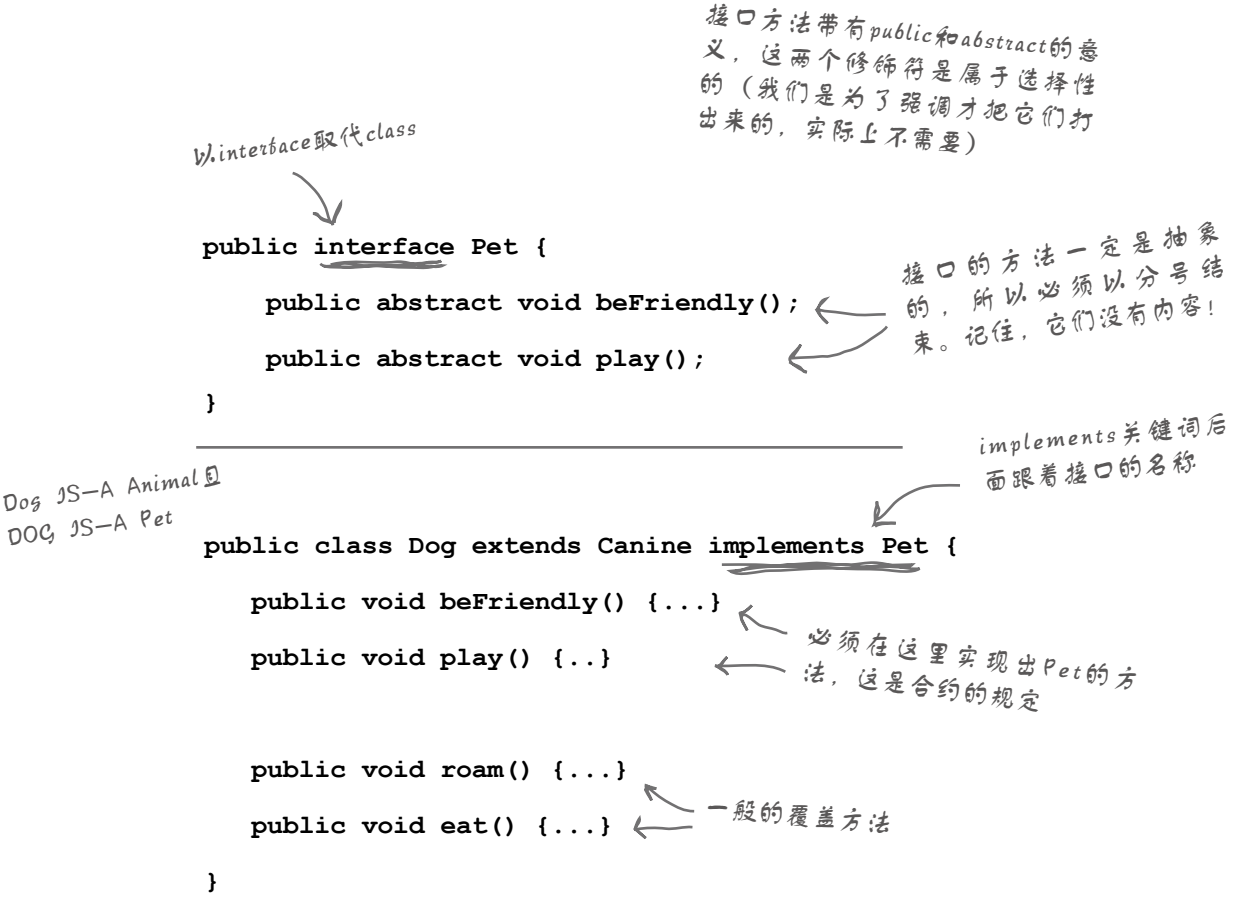
使用“interface”来取代“class”

接口的实现：

```
public class Dog extends Canine implements Pet {...}
```

使用implements这个关键词。注意到实现interface时还是必须在某个类的继承之下

设计与实现 Pet接口



there are no Dumb Questions

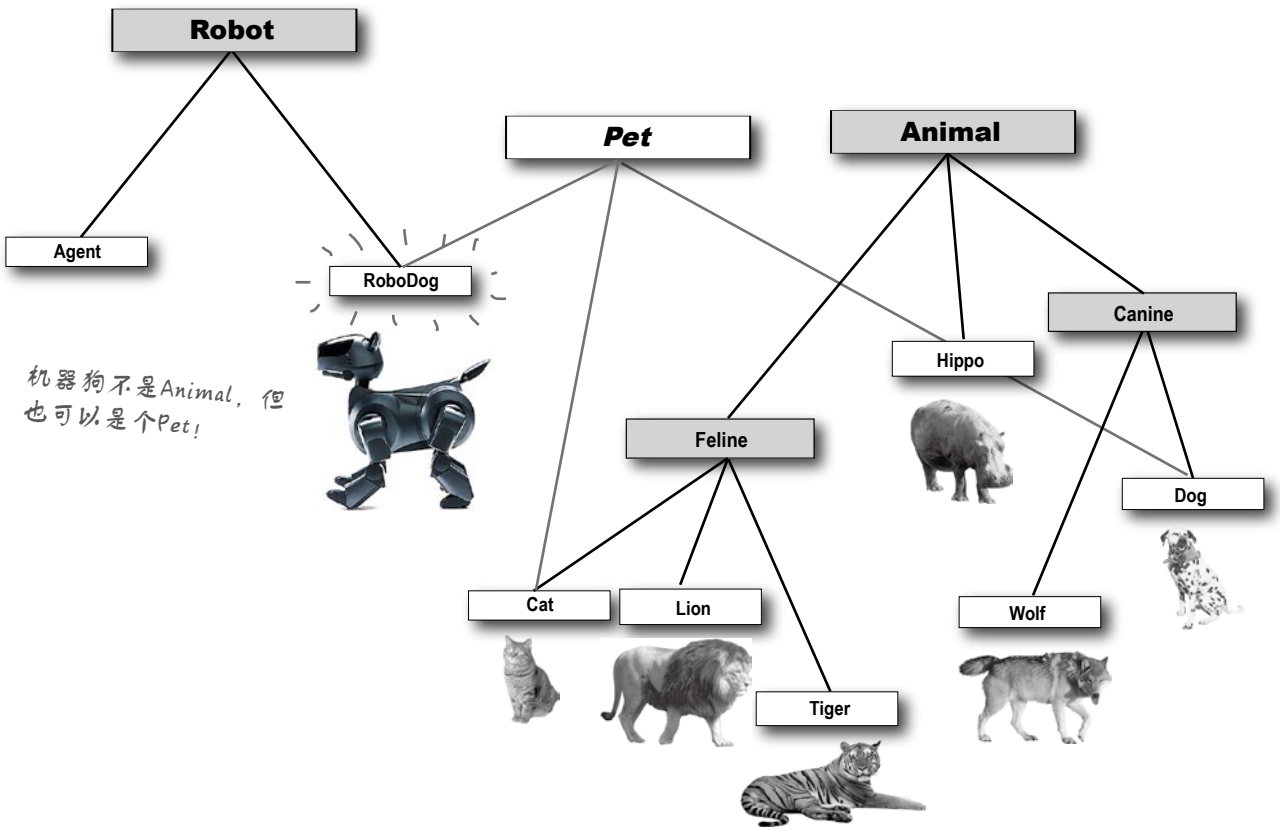
问： **等一下！接口并不是真正的多重继承，因为你无法在它里面实现程序代码，不是吗？如果是这样，那还要接口做什么？**

答： 多态、多态、多态。接口有无比的适用性，若你以接口取代具体的子类或抽象的父类作为参数或返回类型，则你就可以传入任何有实现该接口的东西。这么说吧，使用接口你就可以继承超过一个以上的来源。类可以 extend 过某个父类，并且实现其他的接口。同时其他的类也可以实现同一个接口。因此你就可以为不同的需求组合出不同的继承层次。

事实上，如果使用接口来编写程序，你就是在说：“不管你来自哪里，只要你实现这个接口，别人就会知道你一定会履行这个合约”。

大部分良好的设计也不需要在抽象的层次定义出实现细节，我们所需的只是个共同的合约定义。让细节在具体的子类上实现也是很合理的。

不同继承树的类也可以实现相同的接口



当你把一个类当作多态类型运用时，相同的类型必定来自同一个继承树，而且必须是该多态类型的子类。定义为Canine类型的参数可以接受Wolf与Dog，但无法忍受Cat或Hippo。

但当你用接口来作为多态类型时，对象就可以来自任何地方了。唯一的条件就是该对象必须是来自有实现此接口的类。允许不同继承树的类实现共同的接口对Java API来说是非常重要的。如果你想要将对象的状态保存在文件中，只要去实现Serializable这个接口就行。打算让对象的方法以单独的线程来执行吗？没问题，实现Runnable。有概念了吧。后面的章节会有关于Serializable与Runnable的讨论，现在只要先掌握住这个概念就行。

更棒的是类可以实现多个接口！

通过继承结构，Dog对象IS-A Canine、IS-A Animal、IS-A Object。但Dog IS-A Pet是通过接口实现的机制达成的，并同时也能够实现其他的接口：

```
public class Dog extends Animal implements
Pet, Saveable, paintable {...}
```

要如何判断应该是设计类、子类、抽象类或接口呢？

- 如果新的类无法对其他的类通过IS-A测试时，就设计不继承其他类的类。
- 只有在需要某类的特殊化版本时，以覆盖或增加新的方法来继承现有的类。
- 当你需要定义一群子类的模板，又不想让程序员初始化此模板时，设计出抽象的类给它们用。
- 如果想要定义出类可以扮演的角色，使用接口。

调用父类的方法

问： 如果创建出一个具体的子类且必须要覆盖某个方法，但又需要执行父类的方法时要怎么办？也就是说不打算完全地覆盖掉原来的方法，只是要加入额外的动作要怎么做？

答： 呃……想想看“extends”的字义。设计良好的面向对象要注意到如何编写出必须被覆盖的程序代码。换言之，就是在抽象的类中编写能够共同的实现，让子类加入其余特定的部分。super这个关键词能让你在子类中调用子类的方法。

父类的方法，带有子类可以运用的部分

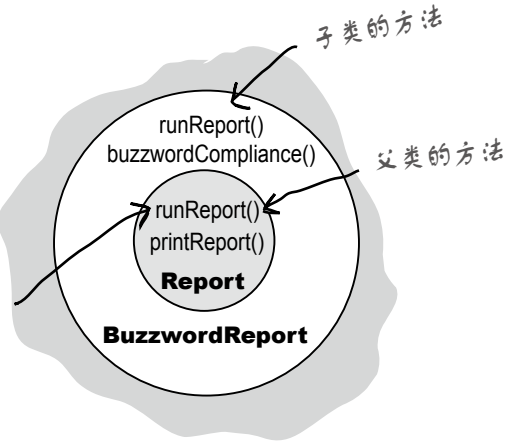
```
abstract class Report {
    void runReport() {
        // 设置报告
    }
    void printReport() {
       // 输出
    }
}

class BuzzwordsReport extends Report {

    void runReport() {
        super.runReport();
        buzzwordCompliance();
        printReport();

    }
    void buzzwordCompliance() {...}
}
```

调用父版的方法

如果在子类中指定下面的命令：

```
super.runReport();
```

父类的方法就会执行。



super.runReport();

对子类的对象调用会去执行子类覆盖过的方法，但在子类中可以去调用父类的方法

super关键字是用来引用父类对象

要点

- 如果不想让某个类被初始化，就以abstract这个关键词将它标记为抽象的。
- 抽象的类可以带有抽象和非抽象的方法。
- 如果类带有抽象的方法，则此类必定标识为抽象的。
- 抽象的方法没有内容，它的声明是以分号结束。
- 抽象的方法必须在具体的类中运行。
- Java所有的类都是Object（java.lang.Object）直接或间接的子类。
- 方法可以声明Object的参数或返回类型。
- 不管实际上所引用的对象是什么类型，只有在引用变量的类型就是带有某方法的类型时才能调用该方法。
- Object引用变量在没有类型转换的情况下不能赋值给其他的类型，若堆上的对象类型与所要转换的类型不兼容，则此转换会在执行期产生异常。

  类型转换的例子：`Dog d = (Dog) x.getObject(aDog);`
- 从ArrayList<Object>取出的对象只能被Object引用，不然就要用类型转换来改变。
- Java不允许多重继承，因为那样会有致命方块的问题。
- 接口就好像是100%纯天然抽象类。
- 以interface这个关键词取代class来声明接口。
- 实现接口时要使用implements这个关键词。

  例如：`Dog implements Pet`
- class可以实现多个接口。
- 实现某接口的类必须实现它所有的方法，因为这些方法都是public与abstract的。
- 要从子类调用父类的方法可以用super这个关键词来引用。

  例如：`super.RunReport( );`

问： 还是有点怪怪的，你没有解释为何ArrayList<DOG>返回的引用无需转换，却还是在方法中使用Object而不是Dog。使用ArrayList <Dog>时是否有什么怪招？

答： 说它是怪招一点也不为过。事实上ArrayList根本就不认识Dog，所以不必作类型转换是个怪招没错。

最简单的回答是：编译器帮你做了类型转换！<Dog>对编译器来说是个禁止将Dog类型以外的对象装进ArrayList的标记。就因为这样，所以编译器也很清楚将从此ArrayList中取出的对象转换为Dog类型是绝对安全的。

但这里面还有很多细节，我们会在讨论Collection的章节加以说明。

练习

这是挑战艺术天分的好机会。下面左方有一组类和接口的声明。你的任务是在右边画出类的图表。第一张图已经帮你画好了。使用虚线来表示实现并以实线来表示继承。

已知：

1)
```
public interface Foo { }
public class Bar implements Foo { }
```

2)
```
public interface Vinn { }
public abstract class Vout implements Vinn { }
```

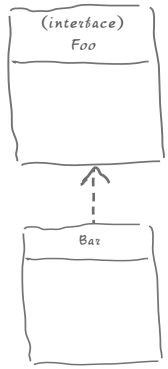
3)
```
public abstract class Muffie implements Whuffie { }
public class Fluffie extends Muffie { }
public interface Whuffie { }
```

4)
```
public class Zoop { }
public class Boop extends Zoop { }
public class Goop extends Boop { }
```

5)
```
public class Gamma extends Delta implements Epsilon { }
public interface Epsilon { }
public interface Beta { }
public class Alpha extends Gamma implements Beta { }
public class Delta { }
```

图呢？

1)



2)

3)

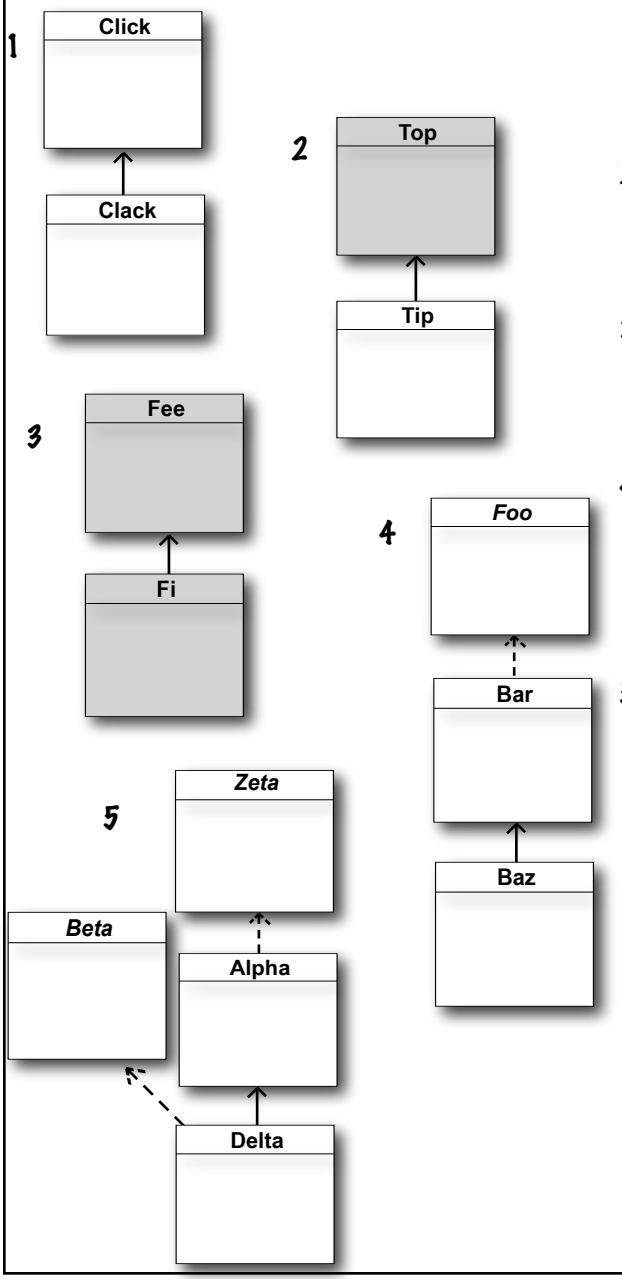
4)

5)

练习

下面左方有一组class和interface的图表。你的任务是在右边写出有效的Java声明。第一张图已经帮你写好声明。

已知：



Java声明呢？

1) *public class Click { }*

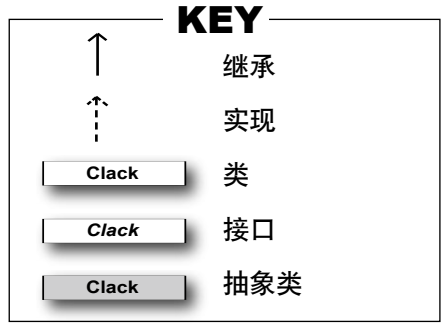
public class Clack extends Click { }

2)

3)

4)

5)



泳池迷宫

你的任务是从游泳池中挑出程序片段并将它们填入右边的空格中。同一个片段可以重复使用，且泳池中有些多余的片段。填完空格的程序必须要能够编译与执行并产生出下面的输出。

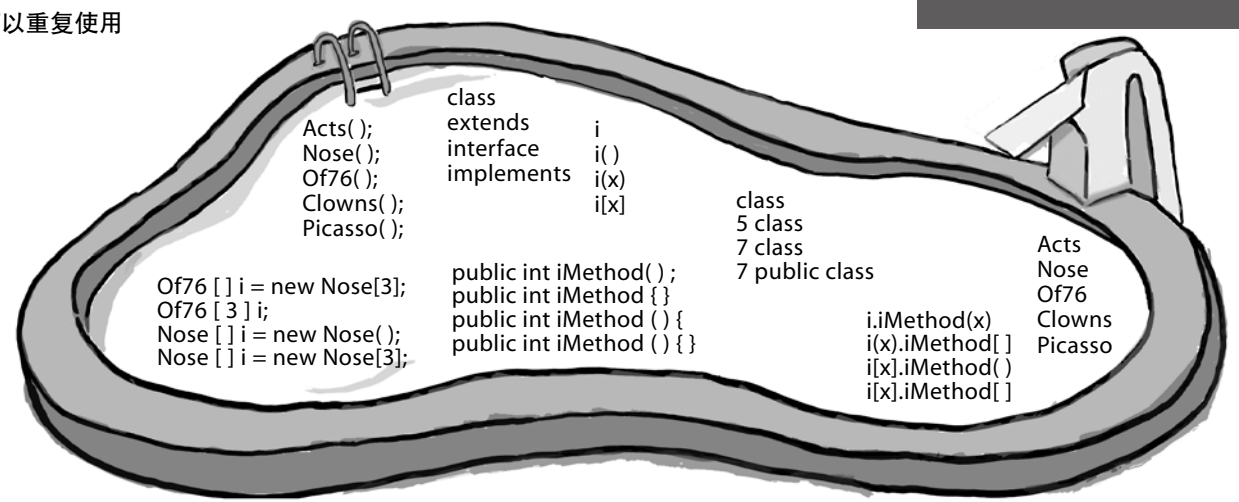
```
_____________ Nose {
  ____________________________
}

abstract class Picasso implements _______{
  ____________________________
    return 7;
  }
}

class __________  _________  ___________ { }

class __________  _________  ___________ {
  ____________________________
    return 5;
  }
}
```

```
public ____________  _________ extends Clowns {

  public static void main(String [] args) {
    ____________________________
    i[0] = new ___________
    i[1] = new ___________
    i[2] = new ___________
    for(int x = 0; x < 3; x++) {
      System.out.println(____________________
              + " " + ________.getClass( ) );
    }
  }
}
```

输出

```
File  Edit  Window  Help  BeAfraid
%java _________________
5 class Acts
7 class Clowns
_________Of76
```

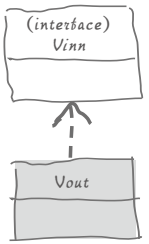
注意：每一个片段可以重复使用



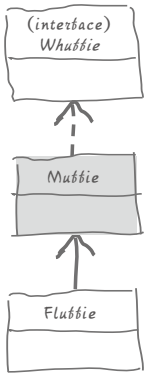
练习解答

艺术天分

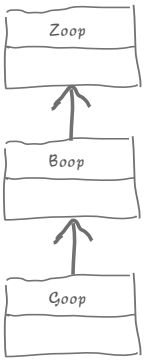
2)



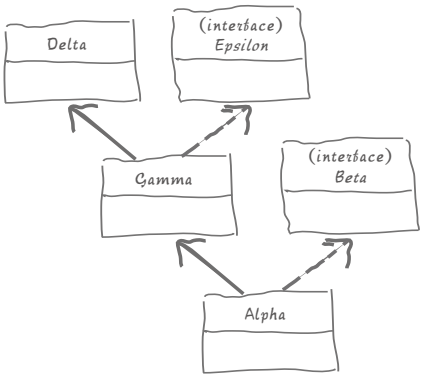
3)



4)



5)



Java 声明

2)

```
public abstract class Top { }
public class Tip extends Top { }
```

3)

```
public abstract class Fee { }
public abstract class Fi extends Fee { }
```

4)

```
public interface Foo { }
public class Bar implements Foo { }
public class Baz extends Bar { }
```

5)

```
public interface Zeta { }
public class Alpha implements Zeta { }
public interface Beta { }
public class Delta extends Alpha implements Beta { }
```

```
interface Nose {
  public int iMethod( ) ;
}
abstract class Picasso implements Nose {
  public int iMethod( ) {
    return 7;
  }
}
class Clowns  extends  Picasso { }

class Acts  extends  Picasso {
  public int iMethod( ) {
    return 5;
  }
}
```

```
public class  Of76 extends Clowns {
  public static void main(String [] args) {
    Nose [ ] i = new Nose [3] ;
    i[0] = new Acts( ) ;
    i[1] = new Clowns( ) ;
    i[2] = new Of76( ) ;
    for(int x = 0; x < 3; x++) {
      System.out.println( i [x] . iMethod( )
              + " " + i [x].getClass( ) );
    }
  }
}
```

输出

```
File  Edit  Window  Help  KillTheMime
%java Of76
5 class Acts
7 class Clowns
7 class  Of76
```