

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;           } // 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
            c = new Dog();
            c = new Canine(); // 这个类已经被标记为 abstract，所以过了编译器这一关
            c.roam();
        }
    }
}
```

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

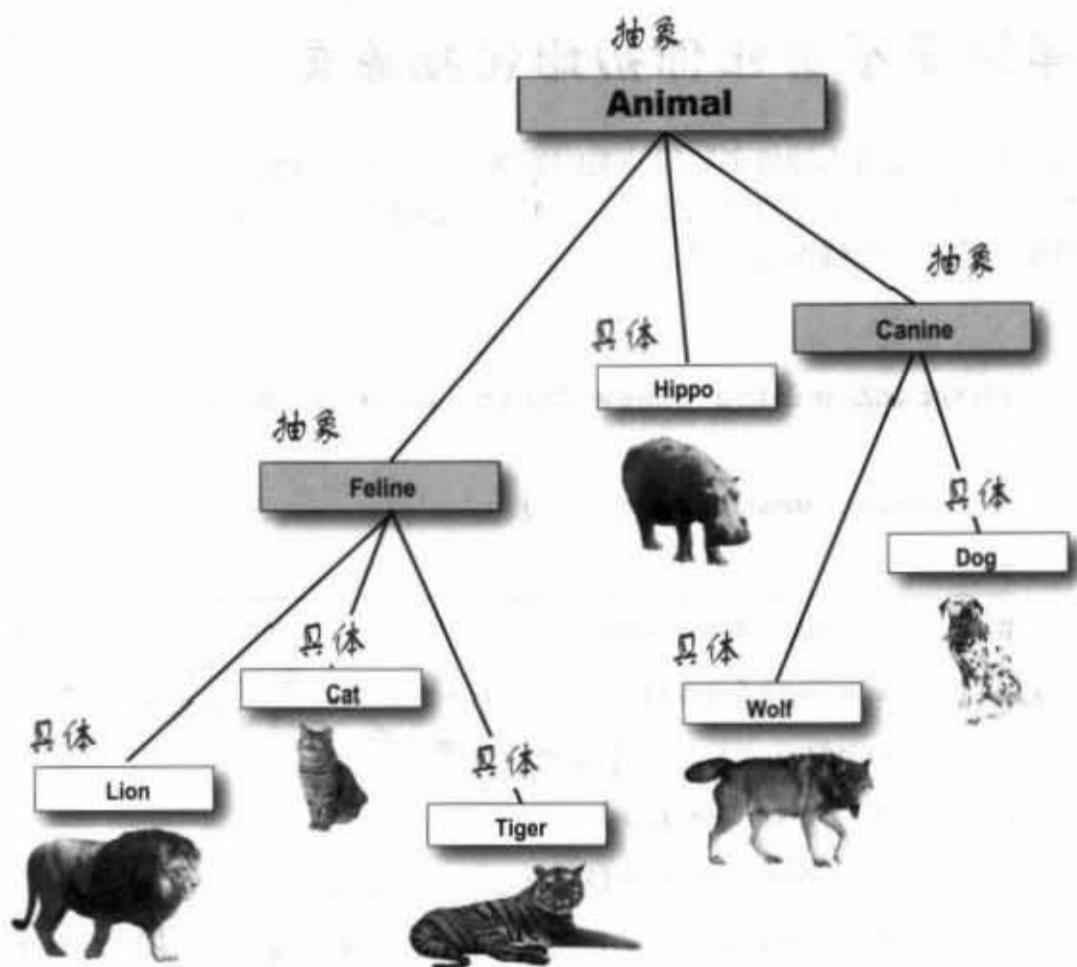
抽象类除了被继承过之外，是没有用途、没有值、没有目的*。

*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！

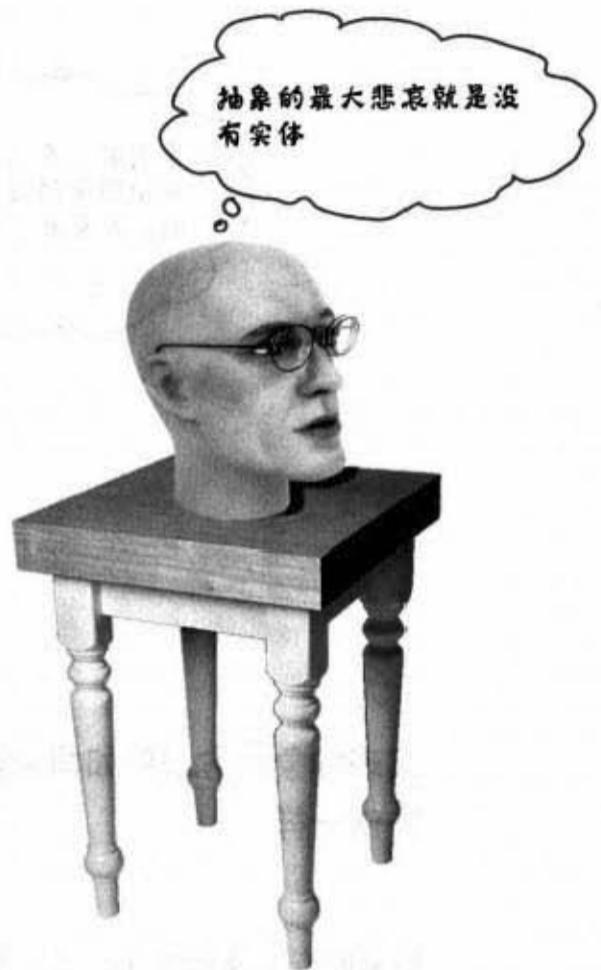
因为你已经知道编写出抽象方法的程序代码没有意义，所以不会含有方法。

```
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。比如说将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，比如说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也可以实现出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签名（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)

version 1

```
public class MyDogList {
    private Dog [] dogs = new Dog[5];
    private int nextIndex = 0; ← 增加新的Dog就加1

    public void add(Dog d) {
        if (nextIndex < dogs.length) { } 如果没有超出上限就把Dog加进去并列出信息
            dogs[nextIndex] = d;
            System.out.println("Dog added at " + nextIndex);

            nextIndex++; ← 递增计数
    }
}
```

实际上使用的是数组

MyDogList
Dog[] dogs
int nextIndex
add(Dog d)

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

```

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++;
        }
    }
}

```

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 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。

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。

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

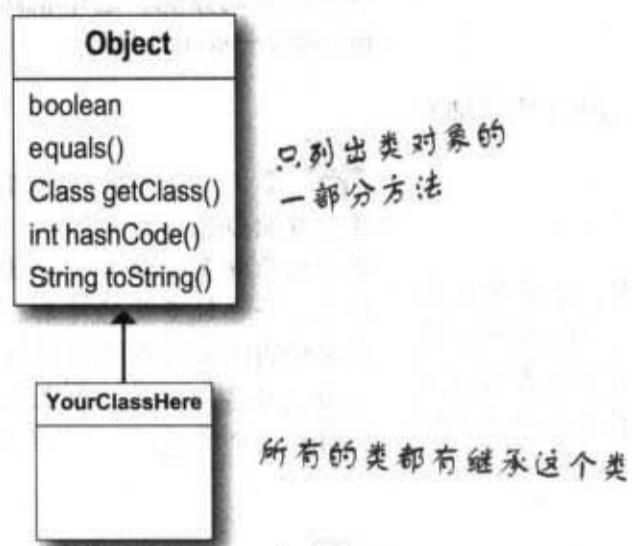
许多ArrayList的方法都用到Object这个终极类型。因为每个类都是对象的子类，所以ArrayList可以处理任何类！

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。

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些什么行为？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。



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以认为是“相等”的（见附录B）

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

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

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，所以没有必要被覆盖过的方法。

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hashCode()、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……想想看这会发生在什么后果。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。

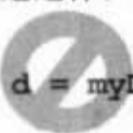
当你把对象装进 ArrayList<Dog> 时，它会被当作 Dog 来输入与输出：

```
ArrayList<Dog> myDogArrayList = new ArrayList<Dog>(); ← 保存 Dog 的 ArrayList
Dog aDog = new Dog(); ← 新建一个 Dog
myDogArrayList.add(aDog); ← 装到 ArrayList 中
Dog d = myDogArrayList.get(0); ← 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：

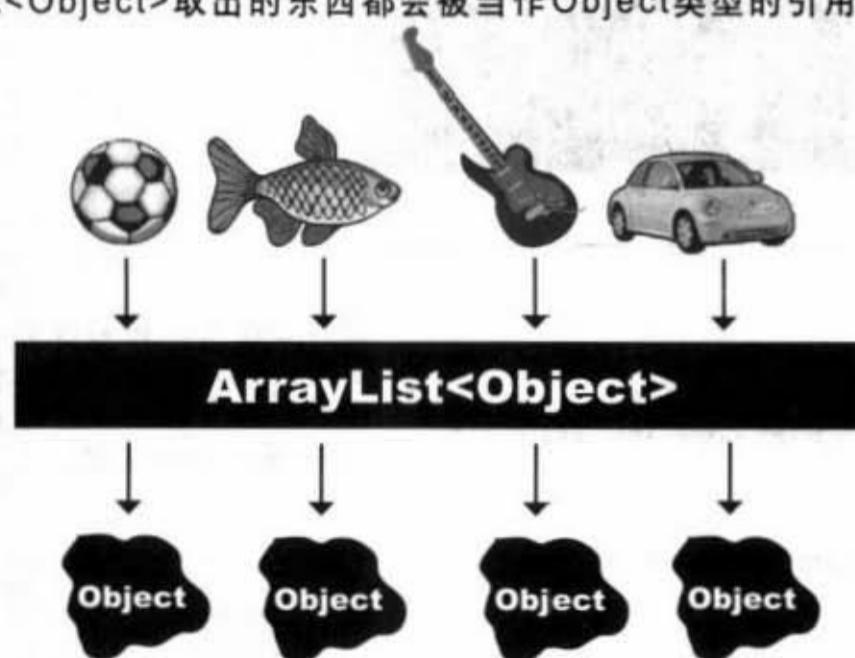
```
ArrayList<Object> myDogArrayList = new ArrayList<Object>(); ← 保存对象的 ArrayList
Dog aDog = new Dog(); ← 新建一个 Dog
myDogArrayList.add(aDog); ← 装到 ArrayList 中
```

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。

失去狗性

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);
}
```

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

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

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 = getObject(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

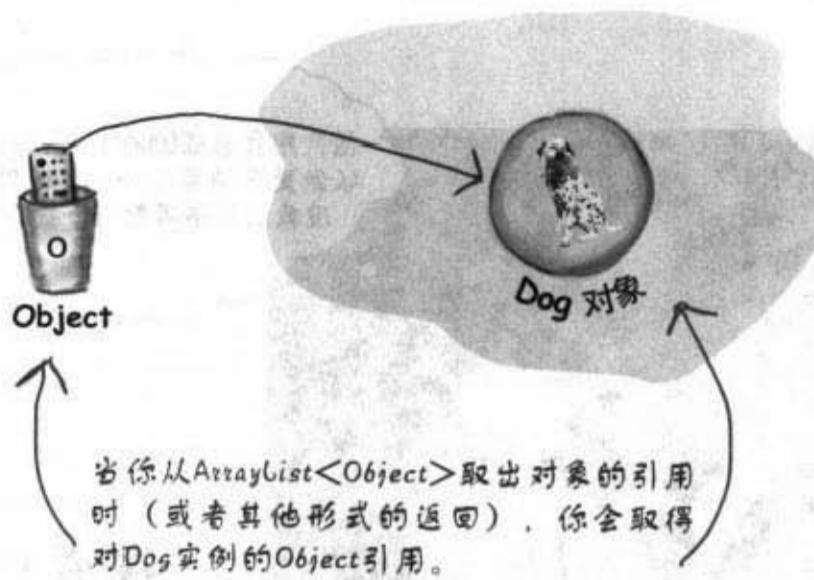
```
public void go() {
    Dog aDog = new Dog();
    Object sameDog = getObject(aDog);
}
```

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JS-A测试。

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

Object不会吠

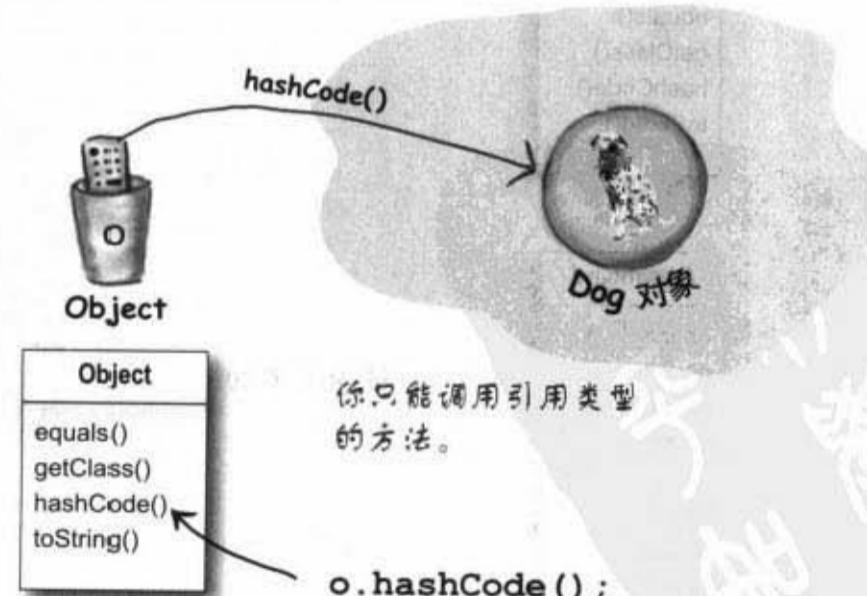
我们已经知道当一个对象被声明为Object类型的对象所引用时，它无法再赋值给原来类型的变量。我们也知道这会发生在返回类型被声明为Object类型的时候，例如前面所提过的ArrayList<Object>。但这意味着什么呢？使用Object引用变量来引用Dog对象会是个问题吗？让我们试着对被编译器认为是Object的Dog调用Dog才有的方法看看：



```
Object o = al.get(index);    没问题。Object本来就有
int i = o.hashCode();      hashCode()这个方法
                           ←
有问题！ → o.bark();      不能这么做！Object根本就不知道
                           什么是bark()。就算天知地知你知道
                           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。

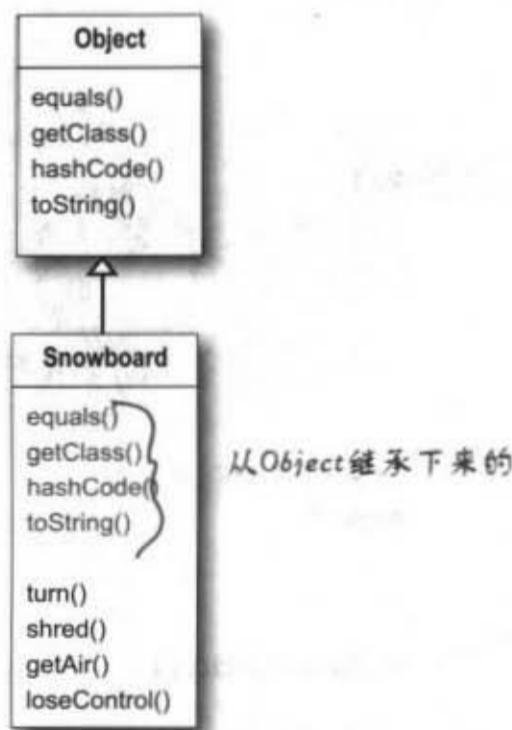


“o”被声明为Object的引用，所以只能通过o调用Object类中的方法。



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。



“多态”意味着“很多形式”

你可以把Snowboard当作Snowboard或者Object

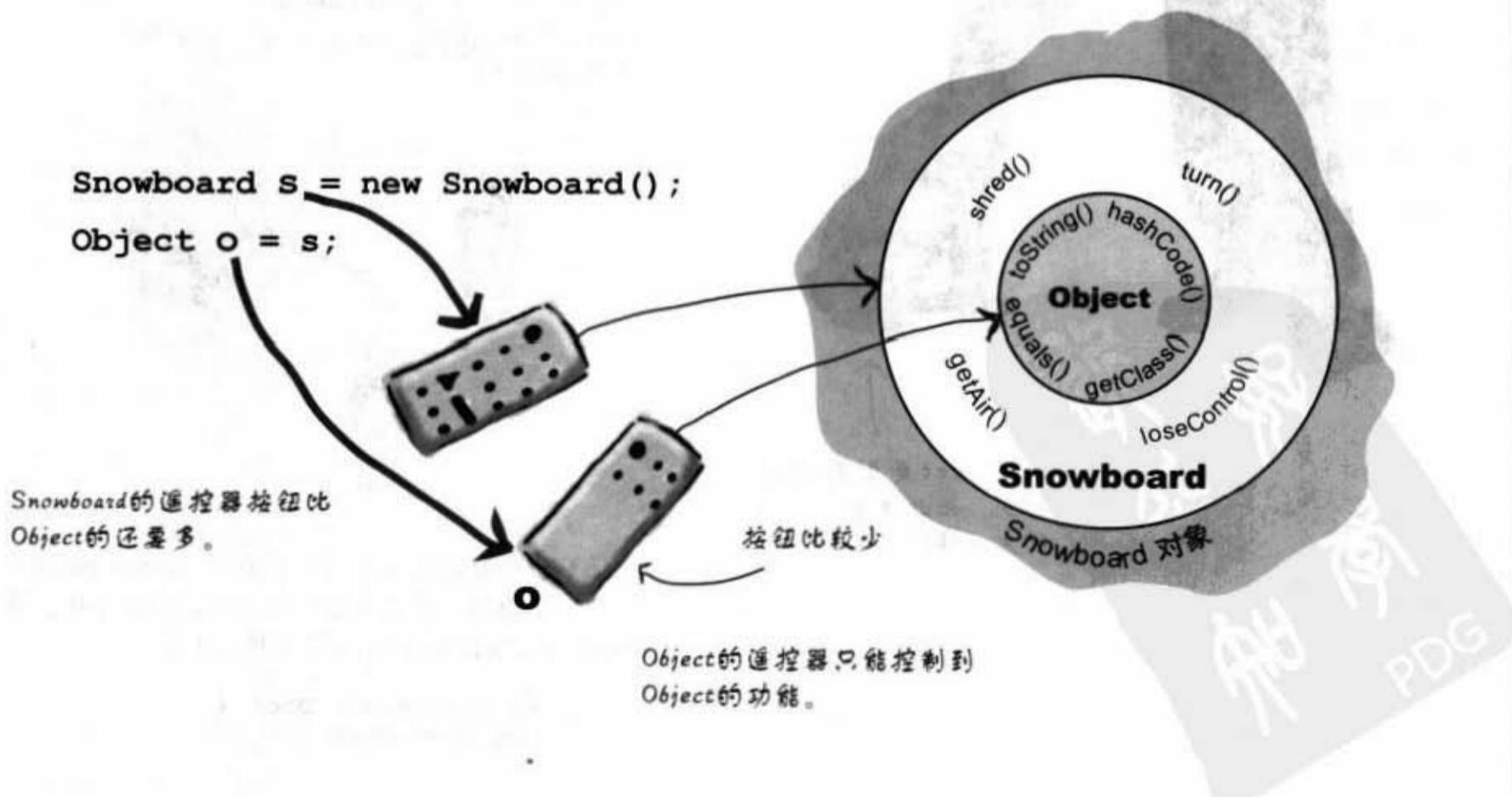
如果引用是个遥控器，则当你在继承树往下走时，会发现遥控器的按钮越来越多。Object的遥控器只有几个按钮而已，但Snowboard的遥控器就会包含有来自Object和自己定义的按钮。越接近具体的类会有越多的按钮。

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。

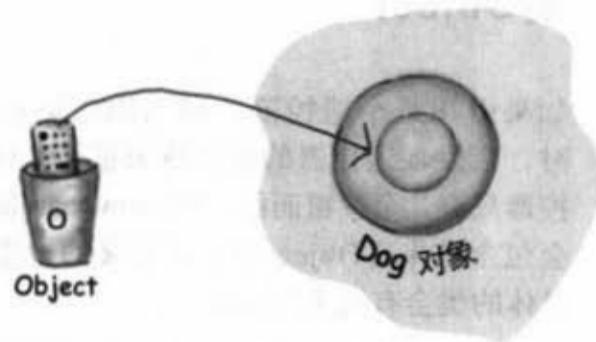
这代表你只会取得Object的遥控器。





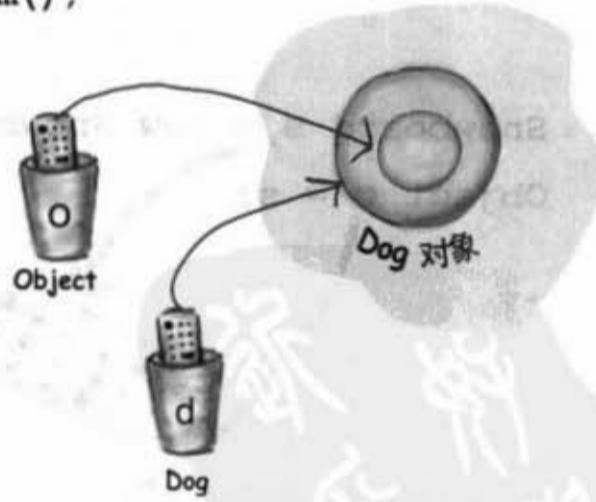
↑
将 Object 类型转换成 Dog, 如此才能用 Dog 的方法操作

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; ← 将类型转换成 Dog  
d.roam();
```



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。要让方法显露出就代表你会让方法能够取得到，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getBalance()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。

Account
debit(double amt)
credit(double amt)
double getBalance()

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。

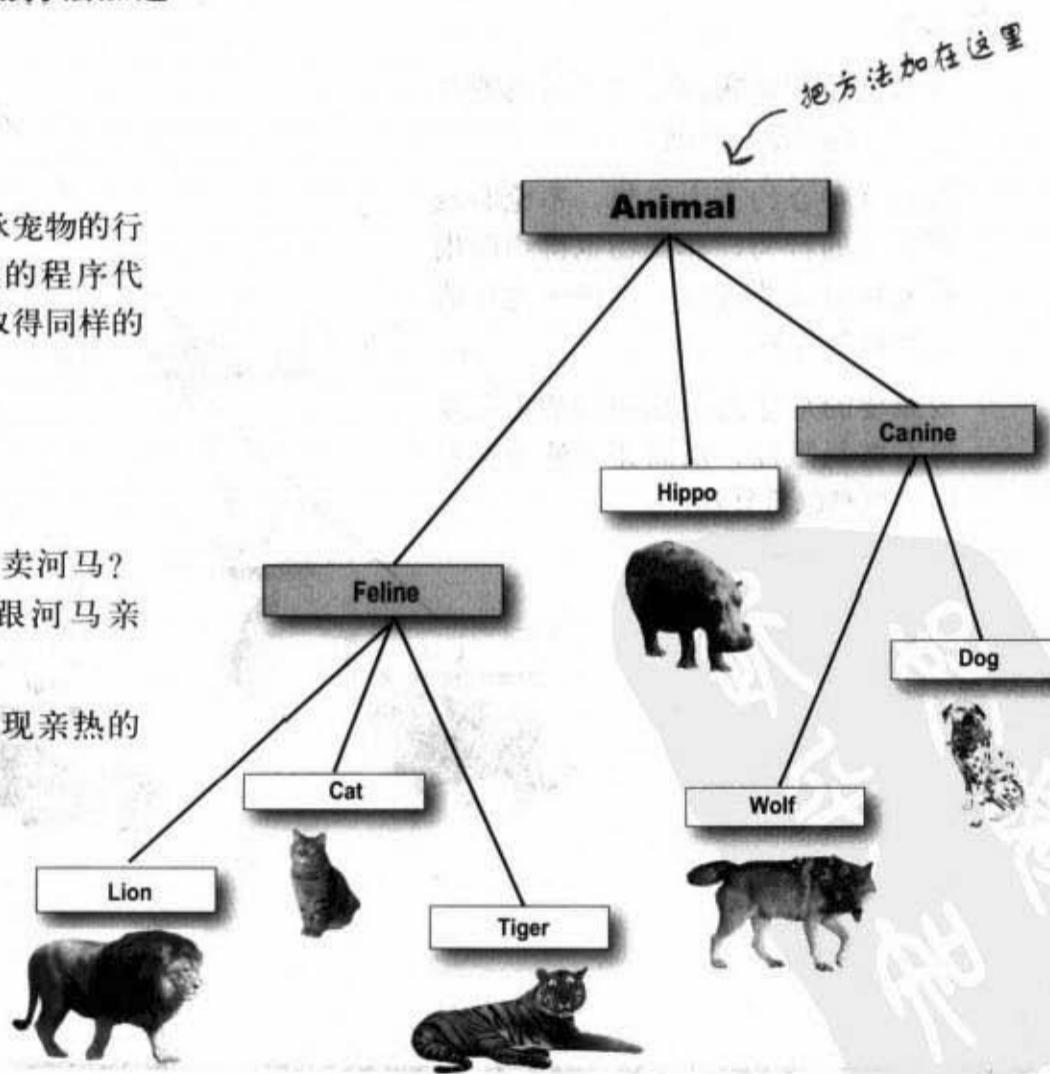
优点：

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：

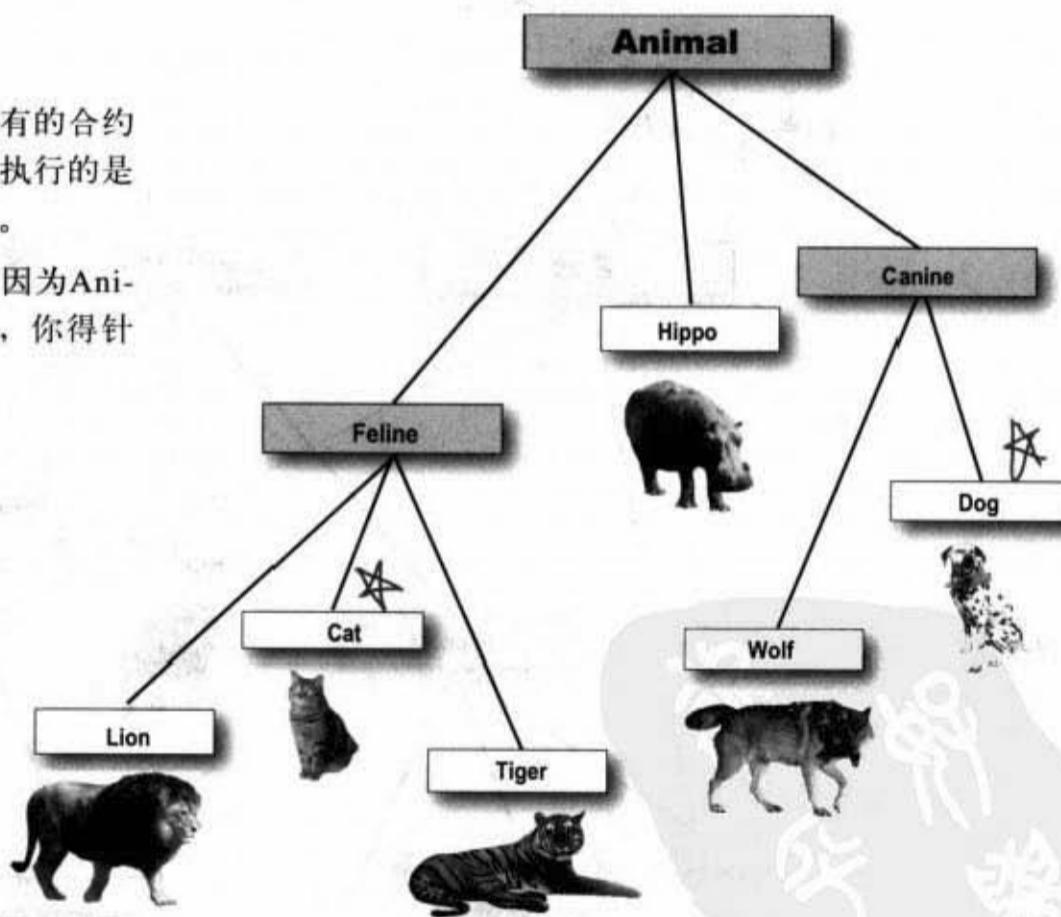
不必担心如何跟河马亲热。只有宠物才会有宠物的行为。

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。

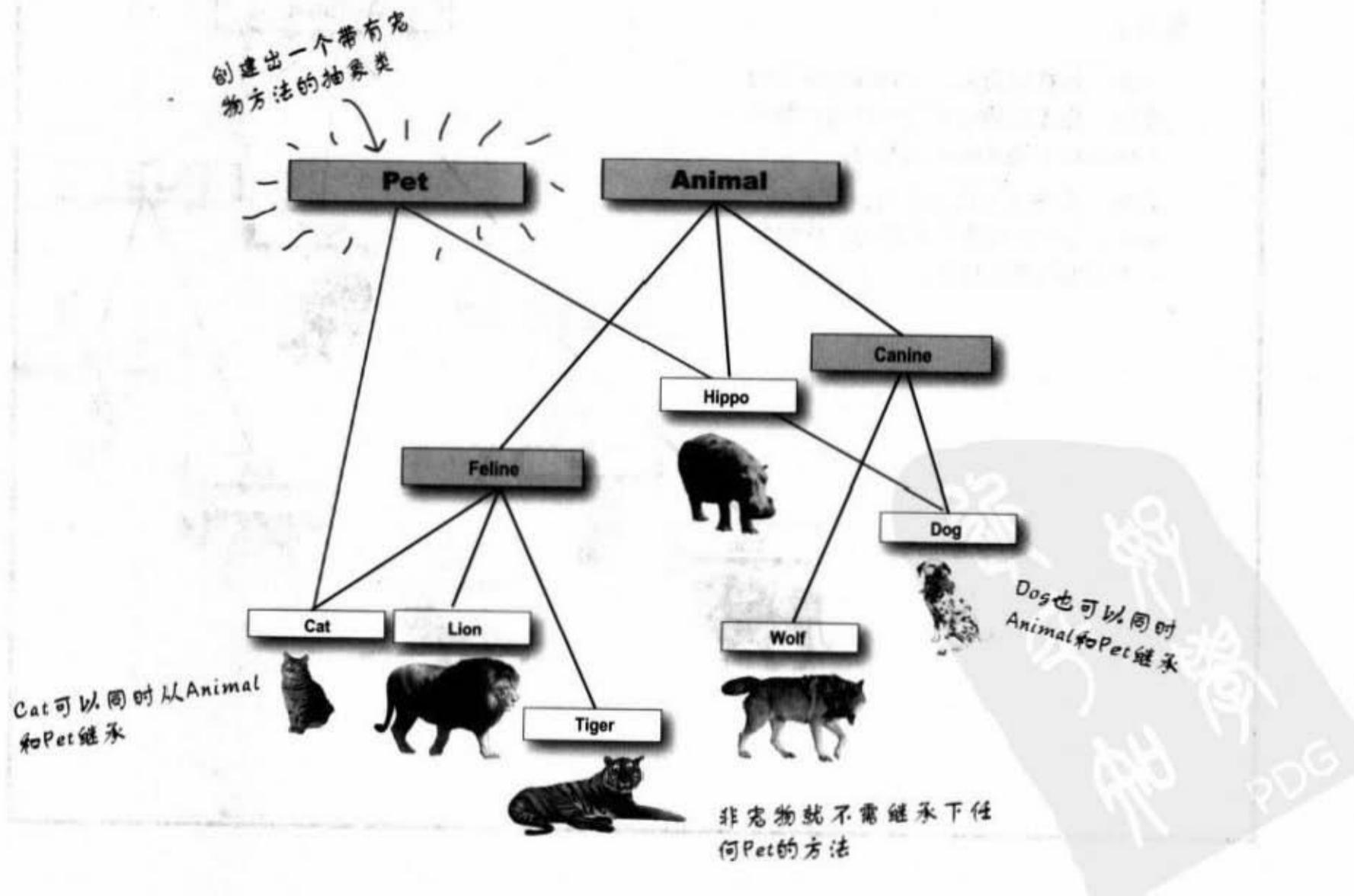


多重继承？

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……

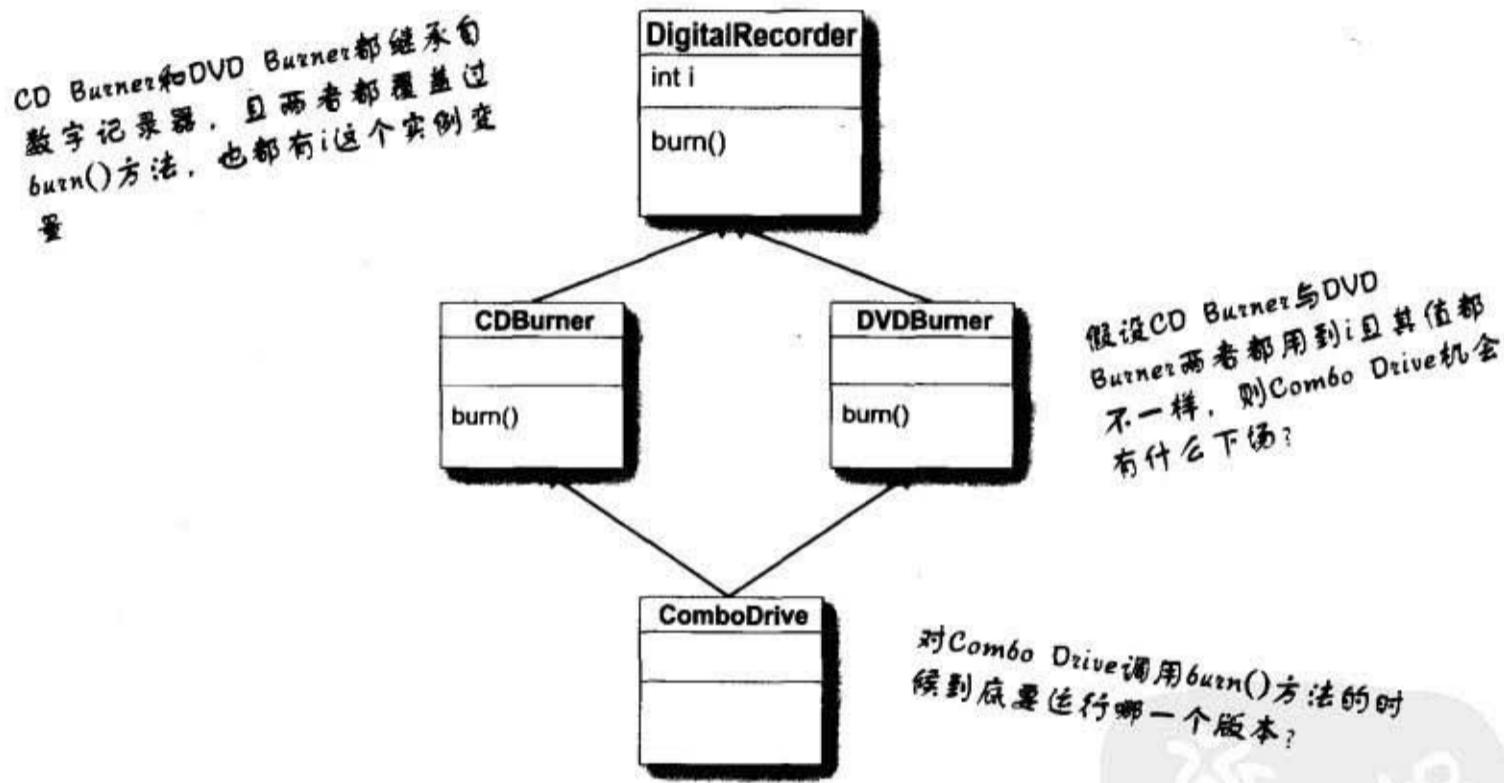
这种“多重继承”可能会很差。

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的类都必须重实现这些方法

接口的定义：

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

使用“interface”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

W/interface 取代 class

```

public interface Pet {
    public abstract void beFriendly();
    public abstract void play();
}

```

接口方法带有 `public` 和 `abstract` 的意义，这两个修饰符是属于选择性的（我们是为了强调才把它们打出来的，实际上不需要）

Dog IS-A Animal 且
Dog IS-A Pet

```

public class Dog extends Canine implements Pet {
    public void beFriendly() {...}
    public void play() {...}
    public void roam() {...}
    public void eat() {...}
}

```

接口的方法一定是抽象的，所以必须以分号结束。记住，它们没有内容！

implements 关键词后面跟着接口的名称

必须在这里实现出 Pet 的方法，这是合约的规定

一般的覆盖方法

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？

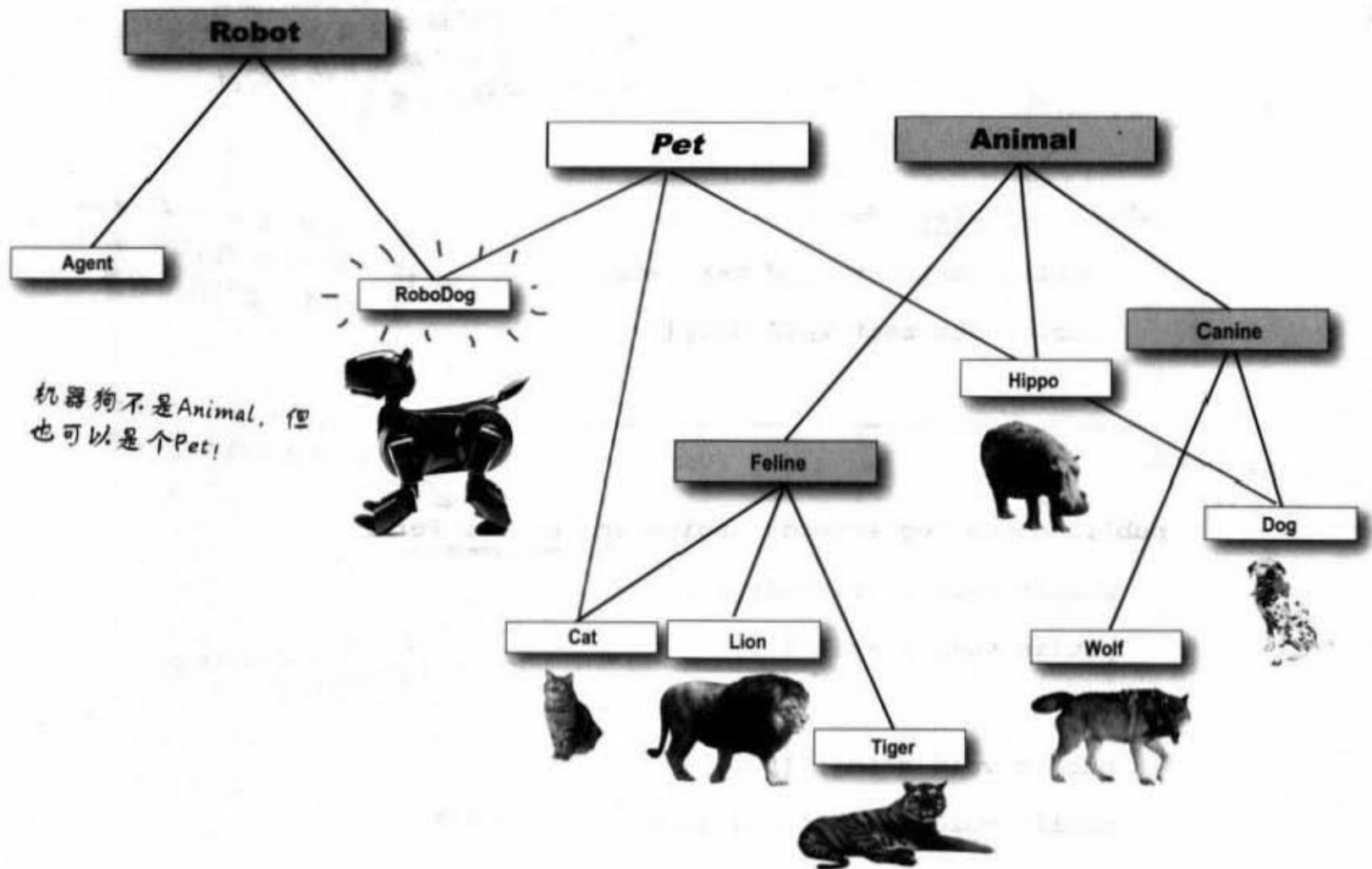
答： 多态、多态、多态。接口有无比的适用性，若你以接口取代具体的子类或抽象的父类作为参数或返回类型，则你就可以传入任何有实现

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。

super的使用

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。

```
abstract class Report {  
    void runReport() {  
        // 设置报告  
    }  
    void printReport() {  
        // 输出  
    }  
  
    class BuzzwordsReport extends Report {  
  
        void runReport() {  
            super.runReport(); ← 调用父版的方法  
            buzzwordCompliance();  
            printReport();  
        }  
        void buzzwordCompliance() {...}  
    }  
}
```

父类的方法，带有子类可以运用的部分

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



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。

习题：图呢？



练习

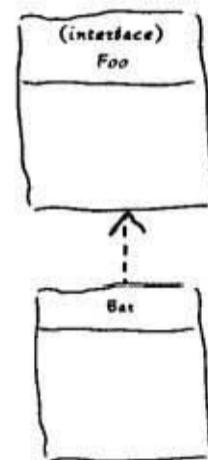
这是挑战艺术天分的好机会。下面左方有一组类和接口的声明。你的任务是在右边画出类的图表。第一张图已经帮你画好了。使用虚线来表示实现并以实线来表示继承。

已知：

图呢？

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

1)



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

2)

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

3)

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

4)

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

5)

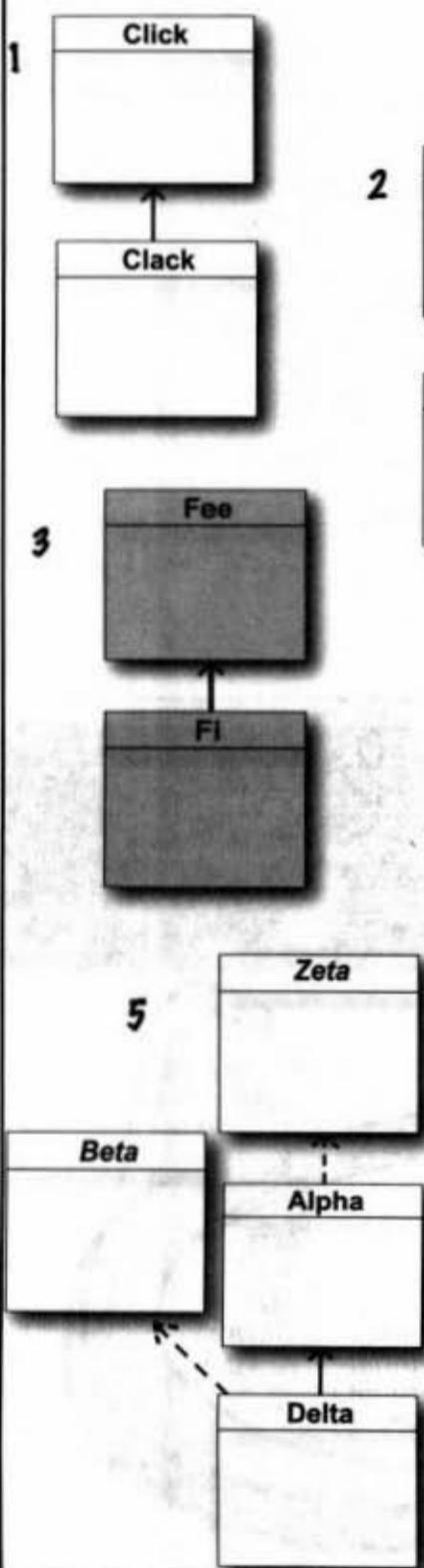


练习

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？



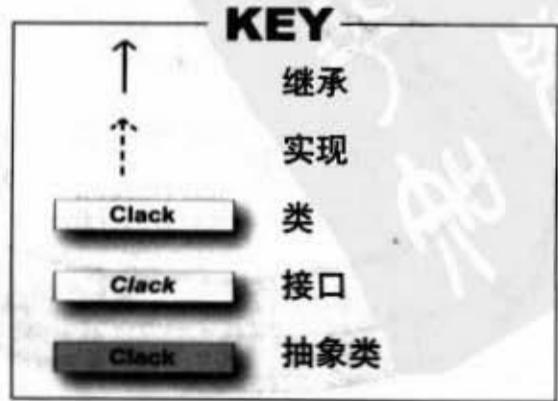
```
1) public class Click { }
public class Clack extends Click { }
```

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;
}

```

注意：每一个片段
可以重复使用

```

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( ) );
        }
    }
}

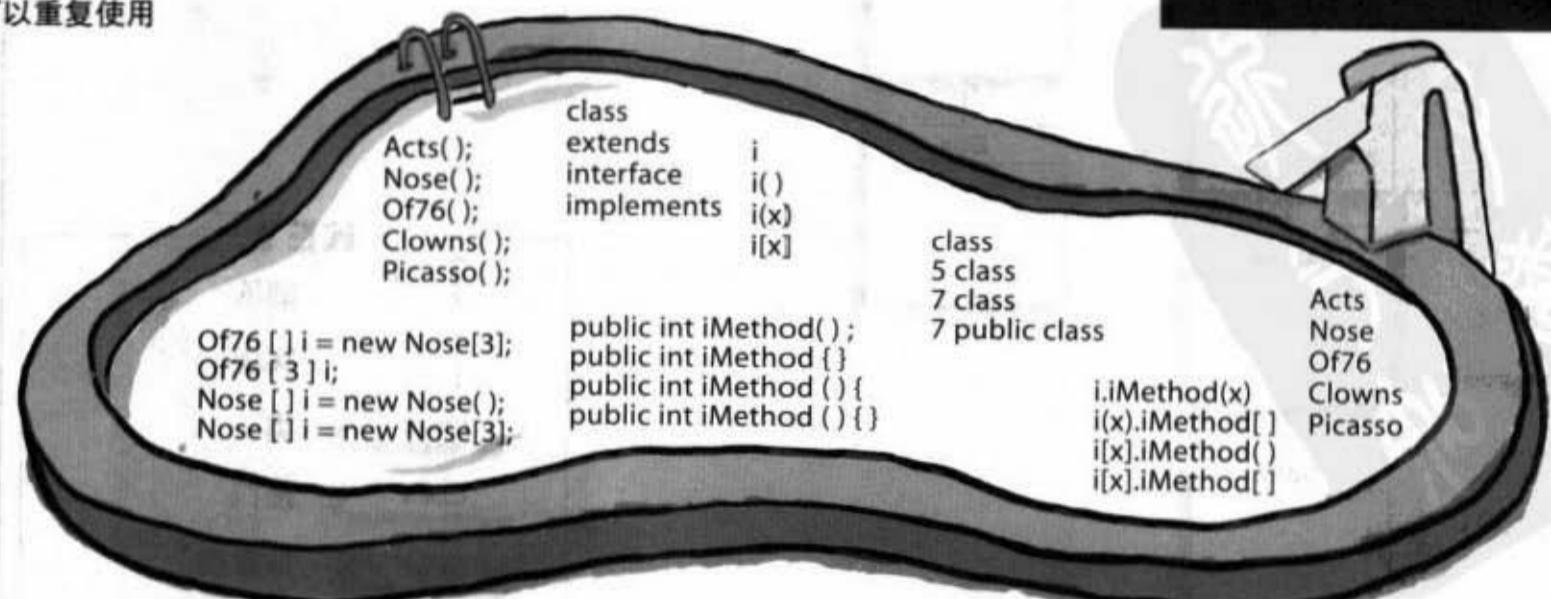
```

输出

```

File Edit Window Help BeAfraid
%java _____
5 class Acts
7 class Clowns
_____ Of76

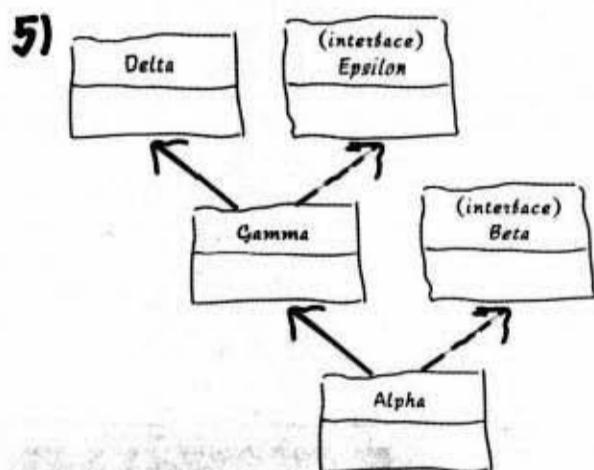
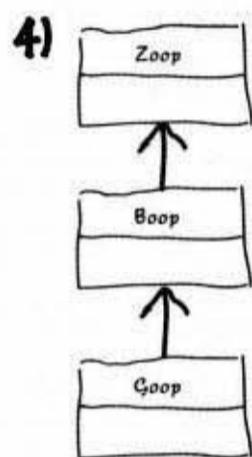
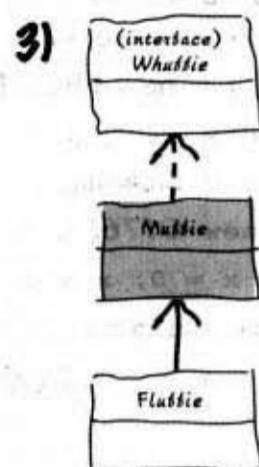
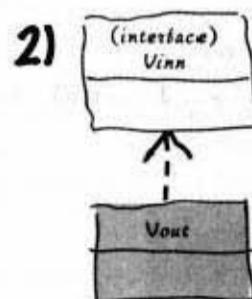
```





练习解答

艺术天分



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 KillTheMine
% java Of76
5 class Acts
7 class Clowns
7 class Of76
```

9 构造器与垃圾收集器

对象的前世今生



忽然一阵阴风吹过来，它还来不及开口，垃圾收集器马上就取走它的性命，我吓得两腿发软，掉底……

对象有生有死。你必须为对象的生命循环周期负责。你决定着对象何时创建，如何创建，也决定着何时销毁对象。其实你不是真的自消灭对象，只是声明要放弃它而已。一旦它被放弃了，冷血无情的垃圾收集器（GC）就会将它蒸发掉、回收对象所占用的内存空间。如果你要编写Java程序，就必须创建对象。早晚你得将它们释放掉，不然就会出现内存不足的问题。这一章会讨论对象如何创建、存在于何处以及如何让保存和抛弃更有效率。这代表我们会述及堆、栈、范围、构造器、超级构造器、空引用等。注意：内容含有死亡成份，12岁以下儿童需由家长陪同观赏。

栈与堆：生存空间

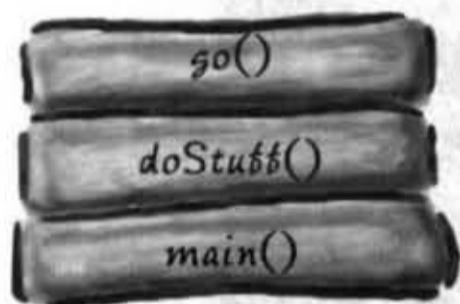
在我们能够了解创建对象真正发生的情况之前，我们必须先退回一步来看。我们需要对生存在Java中的事物更加了解。这代表我们必须对栈与堆有更多的认识。在Java中，程序员会在乎内存中的两种区域：对象的生存空间堆（heap）和方法调用及变量的生存空间（stack）。当Java虚拟机启动时，它会从底层的操作系统取得一块内存，并以此区段来执行Java程序。

至于有多少内存，以及你是否能够调整它都要看Java虚拟机与平台的版本而定。但通常你对这些事情无法加以控制。如果程序设计得不错的话，你或许也不太需要在乎。

我们知道所有的对象都存活于可垃圾回收的堆上，但我们还没看过变量的生存空间。而变量存在于哪一个空间要看它是哪一种变量而定。这里说的“哪一种”不是它的类型，而是实例变量或局部变量。后者这种区域变量又被称为栈变量，该名称已经说明了它所存在的区域。

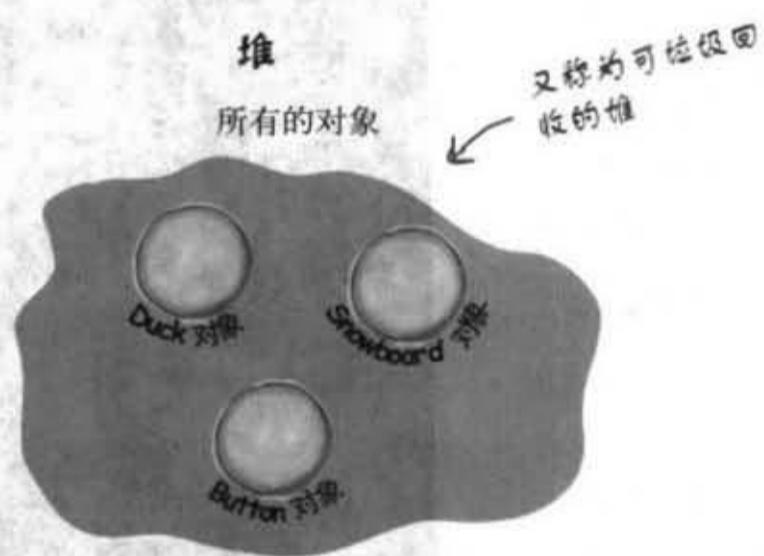
栈

方法调用和局部变量。



堆

所有的对象



实例变量

实例变量是被声明在类而不是方法里面。它们代表每个独立对象的“字段”（每个实例都能有不同的值）。实例变量存在于所属的对象中。

```
public class Duck {
    int size; // 每个Duck对象都会有独立的size
}
```

局部变量

局部变量和方法的参数都是被声明在方法中。它们是暂时的，且生命周期只限于方法被放在栈上的这段期间（也就是方法调用至执行完毕为止）。

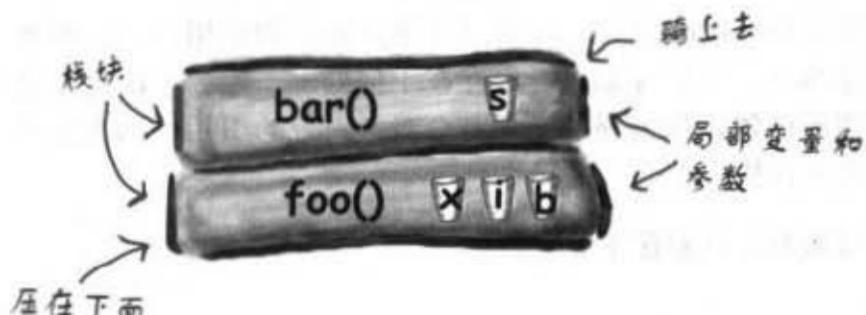
```
public void foo(int x) {
    int i = x + 3; // 参数x和变量i, b都是局部变量
    boolean b = true;
}
```

方法会被堆在一起

当你调用一个方法时，该方法会放在调用栈的栈顶。实际被堆上栈的是堆栈块，它带有方法的状态，包括执行到哪一行程序以及所有的局部变量的值。

栈顶上的方法是目前正在执行的方法（先假设只有一个，第14章有更多的说明）。方法会一直待在这里直到执行完毕，如果foo()方法调用bar()方法，则bar()方法会放在foo()方法的上面。

放了两个方法的栈



栈顶上方法是目前正在执行中的

```
public void doStuff() {
    boolean b = true;
    go(4);
}

public void go(int x) {
    int z = x + 24;
    crazy();
    // 假设还有很多程序代码
}

public void crazy() {
    char c = 'a';
}
```

stack 的情境

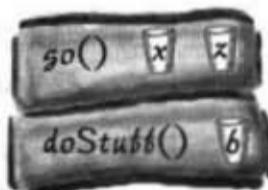
左边有3个方法，第一个方法在执行过程中会调用第二个方法，第二个会调用第三个。每个方法都在内容中声明一个局部变量，而go()方法还有声明一个参数（这代表go()方法有两个局部变量）。

① 某段程序代码调用了doStuff()使得doStuff()被放在stack最上方的栈块中。

② doStuff()调用go()。go()就被放在栈顶。

③ go()又调用crazy()使得crazy()现在处于栈顶。

④ 当crazy()执行完成后，它的堆栈块就被释放掉。执行就回到了go()。

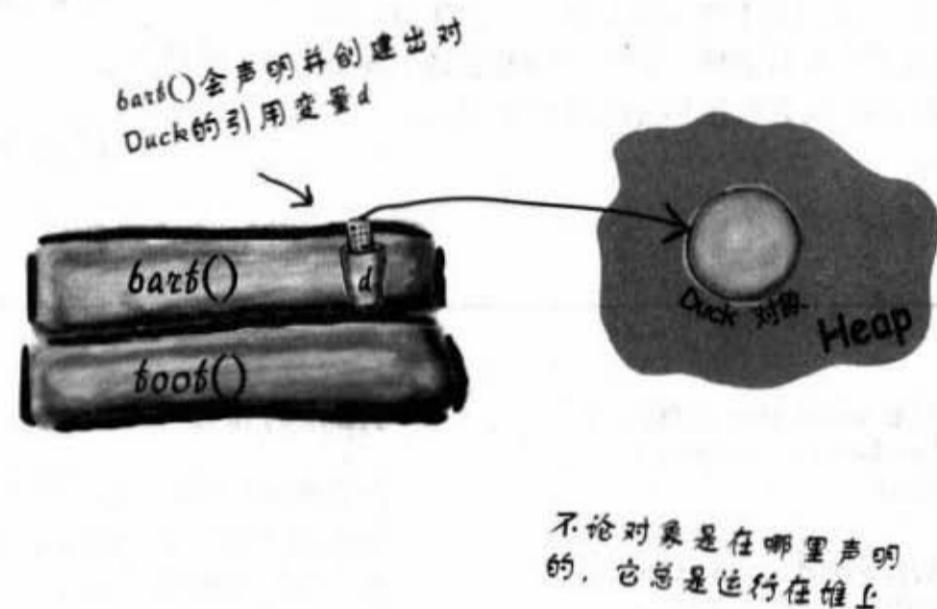


有关对象局部变量

要记得非primitive的变量只是保存对象的引用而已，而不是对象本身。你已经知道对象存在于何处——堆。不论对象是否声明或创建，如果局部变量是个对该对象的引用，只有变量本身会放在栈上。

对象本身只会存在于堆上。

```
public class StackRef {  
    public void foof() {  
        barf();  
    }  
  
    public void barf() {  
        Duck d = new Duck(24);  
    }  
}
```



there are no
Dumb Questions

问：到底为什么要学栈与堆的机制？
这真地跟我有关系吗？

答：如果想要了解变量的有效范围 (scope)、对象的建立、内存管理、线程 (thread) 和异常处理，则认识栈与堆是很重要的。后面的章节会讨论到有关线程与异常处理的部分，其余的都会在这一章讨论。你无需知道它们是如何实现的，只要能够理解这几页的内容就足够了。

要点

- 我们关心栈与堆这两种内存空间。
- 实例变量是声明在类中方法之外的地方。
- 局部变量声明在方法或方法的参数上。
- 所有局部变量都存在于栈上相对应的堆栈块中。
- 对象引用变量与primitive主数据类型变量都是放在栈上。
- 不管是实例变量或局部变量，对象本身都会在堆上。

如果局部变量生存在栈上，那么实例变量呢？

当你要新建一个CellPhone()时，Java必须在堆上帮CellPhone找一个位置。这会需要多少空间呢？足以存放该对象所有实例变量的空间。没错，实例变量存在于对象所属的堆空间上。

记住对象的实例变量的值是存放于该对象中。如果实例变量全都是primitive主数据类型的，则Java会依据primitive主数据类型的大小为该实例变量留下空间。int需要32位，long需要64位，依此类推。Java并不在乎私有变量的值，不管是32或32,000,000的int都会占用32位。

但若实例变量是个对象呢？如果CellPhone对象带有一个Antenna对象呢？也就是说CellPhone带有Antenna类型的引用变量呢？

当一个新建对象带有对象引用的变量时，此时真正的问题是：是否需要保留对象带有的所有对象的空间？不是这样的。无论如何，Java会留下空间给实例变量的值。但是引用变量的值并不是对象本身，所以若CellPhone带有Antenna，Java只会留下Antenna引用量而不是对象本身所用到的空间。

那么Antenna对象会取得在堆上的空间吗？我们得先知道Antenna对象是在何时创建的。这要看实例变量是如何声明的。如果有声明变量但没有给它赋值，则只会留下变量的空间：

```
private Antenna ant;
```

直到引用变量被赋值一个新的Antenna对象才会在堆上占有空间：

```
private Antenna ant = new Antenna();
```

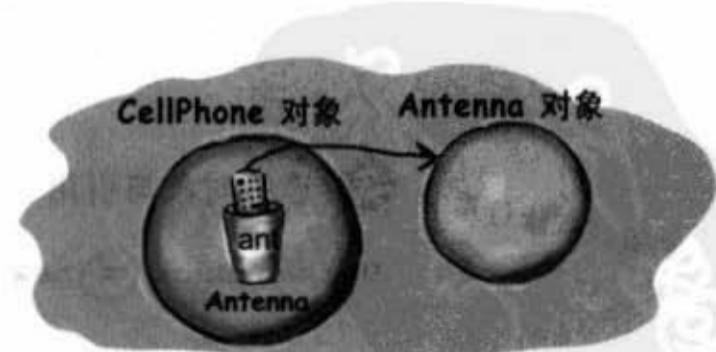


带有两个primitive主数据类型的实例变量的对象。变量所需的空间是在对象中。



对象带有引用到Antenna对象的变量，但是实际上没有初始Antenna对象的情形。

```
public class CellPhone {  
    private Antenna ant;  
}
```



对象带有一个新建出的Antenna对象

```
public class CellPhone {  
    private Antenna ant = new Antenna();  
}
```

创建对象的奇迹

现在你已经知道变量和对象的生存空间，我们可以开始更深入对象的创建。要记得声明对象和赋值有3个步骤：声明引用变量、创建对象、连接对象和引用。

但是第二个步骤还是个谜团——新对象的诞生。准备好接收新知识。

3个步骤的回顾：声明、创建、赋值

创造出新的引用变量
给该类型

1 声明引用变量

`Duck myDuck = new Duck();`



Duck引用

呈现奇迹

2 创建对象

`Duck myDuck = new Duck();`

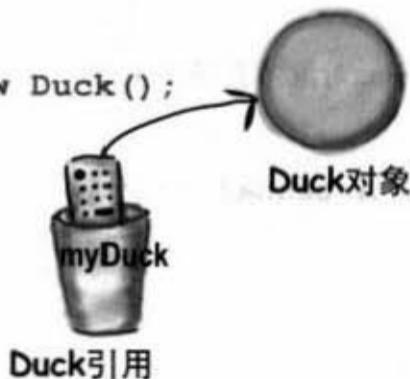


Duck对象

赋值对象给引用

3 连接对象与引用

`Duck myDuck = new Duck();`



Duck对象

看起来很像是在调用 Duck() 这个方法

```
Duck myDuck = new Duck();
```

看起来像是在调用
Duck()方法

并不是。

我们是在调用 Duck 的构造函数。

构造函数看起来像方法，感觉上也很像方法，但它并不是方法。它带有 new 的时候会执行的程序代码。换句话说，这段程序代码会在你初始化一个对象的时候执行。

唯一能够调用构造函数的办法就是新建一个类。（严格说起来，这是唯一在构造函数之外能够调用构造函数的方式，本章稍后会讨论这个部分）。

构造函数带有你在初始化对象时会执行的程序代码。也就是新建一个对象时就会被执行。

就算你没有自己写构造函数，编译器也会帮你写一个。

哪里来的构造函数？

我们没有写啊，难道说这本书有缺页？

你可以帮类编写构造函数，但如果你没有写，编译器会偷偷帮你写！

下面就是编译器写出来的

```
public Duck() {  
}
```

有没有发现少了什么？这跟方法有什么不同之处？

方法有返回类型，构造函数没有返回类型

```
public Duck() {  
    // 构造代码在此  
}
```

一定要与类的名称相同

构造Duck

构造函数的一项关键特征是它会在对象能够被赋值给引用之前就执行。这代表你可以有机会在对象被使用之前介入。也就是说，在任何人取得对象的遥控器前，对象有机会对构造过程给予协助。在Duck的构造函数中，我们没有作出什么有意义的事情，但还是有展示出事件的顺序。



```
public class Duck {  
  
    public Duck() {  
        System.out.println("Quack");  
    }  
}
```

列出一行语句

构造函数让你有机会可以介入new的过程

```
public class UseADuck {  
  
    public static void main (String[] args) {  
        Duck d = new Duck();  
    }  
}
```

这样会启动Duck的构造函数

Sharpen your pencil

构造函数让你可以在构造过程的步骤中参一脚。你能否想象这有什么用处吗？如果Car是赛车的类，对于右边几个情境你是否能想象出构造函数的用途？可以的话就打个勾。

- 记录已经构造出多少部赛车。
- 记录特定的状态。
- 给实例变量赋值。
- 留下创建对象的证据。
- 将对象加到ArrayList中。
- 创建HAS-A对象。
- _____ (自己想)

新建Duck状态的初始化

大部分的人都是使用构造函数来初始化对象的状态。也就是说设置和给对象的实例变量赋值。

```
public Duck() {
    size = 34;
}
```

这在开发者知道Duck类应该有多大时是没问题的。但如果是要由使用Duck的程序员来决定时应该怎么办？

你可以使用该类的setSize()来设定大小。但这会让Duck暂时处于没有大小数值的状态（实例变量没有默认值），且需要两行才能搞定。下面就是这么做的：

```
public class Duck {
    int size; ← 实例变量

    public Duck() {
        System.out.println("Quack"); ← 构造函数
    }

    public void setSize(int newSize) { ←
        size = newSize; ←
    } ←
}
```

```
public class UseADuck {
    public static void main (String[] args) {
        Duck d = new Duck();
        ←
        d.setSize(42); ←
    } ←
}
```

问题出在这里。Duck在此处已经建立。
但是却没有size值！你必须依赖Duck
的用户记得要设定大小。)

*there are no
Dumb Questions*

问：既然编译器会帮你写，那为何还要自己写构造函数？

答：如果你在创建对象时需要有程序代码帮忙初始化，那你就得自己编写构造函数。例如说你需要通过用户的输入来完成对象的创建。另外一个原因与父类的构造函数有关，稍后会讨论这个部分。

问：如何分辨构造函数和方法？

答：Java可以有与类同名的方法而不会变成构造函数。其中的差别在于是否有返回类型。构造函数不会有返回类型。

问：构造函数会被继承吗？

答：不会。我们稍后会讨论到这个部分。

使用构造函数来初始化 Duck 的状态

如果某种对象不应该在状态被初始化之前就使用，就别让任何人能够在没有初始化的情况下取得该种对象！让用户先构造出Duck对象再来设定大小是很危险的。如果用户不知道，或者忘记要执行setSize()怎么办？

最好的方法是把初始化的程序代码放在构造函数中，然后把构造函数设定成需要参数的。

今天的晚餐是……烤鸭！？
对不起，我是天鹅，我真地是
天鹅……

```
public class Duck {
    int size;

    public Duck(int duckSize) {
        System.out.println("Quack");
        size = duckSize;
    }

    System.out.println("size is " + size);
}
```

给构造函数加上参数
使用参数的值来设定size这个
实例变量

```
public class UseADuck {

    public static void main (String[] args) {
        Duck d = new Duck(42);
    }
}
```

传值给构造函数

只用一行就可以创
造出新的Duck并且设
定好大小

```
File Edit Window Help Honk
% java UseADuck
Quack
size is 42
```

Duck的简易饲养方法

一定要有不需参数的构造函数。

如果Duck的构造函数需要一项参数会怎样？想想看。上一页的Duck只有一个构造函数，且它需要一个int型的size参数。这也许不是个问题，但却让程序员感到更为困难，特别是在不知道Duck的大小时。如果有预设的大小让程序员在不知道适当大小时也可以创建出Duck不是更好吗？

想象一下你可以让用户在创建Duck时有两个选项：一个可以指定Duck的大小（通过构造函数的参数），另外一个使用默认值而无需指定大小。

你无法只依靠单一的构造函数就能够很清楚地达到这个目的。要记得，如果某个方法或构造函数有一项参数，你就必须在调用该方法或构造函数的时候传入适当的参数。你没有办法作出一种没给参数时就使用默认值的方法，因为在这个情况下没有给参数就无法通过编译程序。也许你可以用下面这种不太理想的方法取代：

```
public class Duck {
    int size;

    public Duck(int newSize) {
        if (newSize == 0) { ←
            size = 27;
        } else {
            size = newSize;
        }
    }
}
```

如果参数值为0就使用默认的大小

这代表程序员必须要知道传入0对于创建Duck的构造函数意味着要使用默认的大小而不是真正的0。万一程序员真的做出0大小的Duck怎么办？这样的问题在于传入0的意图无法确实的分辨。

你需要有两种方法来创建出新的Duck：

```
public class Duck2 {
    int size;

    public Duck2() {
        // 指定默认值
        size = 27;
    }

    public Duck2(int duckSize) {
        // 使用参数设定
        size = duckSize;
    }
}
```

知道大小时：

```
Duck2 d = new Duck2(15);
```

不知道大小时：

```
Duck2 d2 = new Duck2();
```

因此这会需要两个构造函数来分辨两种选项。一个需要参数，另外一个不需要参数。如果一个类有一个以上的构造函数，这代表它们也是重载的。

编译器一定会帮你写出没有参数的构造函数吗？No！

你可能会认为如果你只编写有参数的构造函数，编译器会看出你没有无参数的构造函数而帮你弄出一个来。别再相信没有事实根据的说法了，编译器只会在你完全没有设定构造函数时才会调用。

如果你已经写了一个有参数的构造函数，并且你需要一个没有参数的构造函数，则你必须自己动手写！

只要你有自己写的构造函数，不管是哪一种，这都会像是在跟编译器说：“老兄，我自己的构造函数不用你管”。

如果类有一个以上的构造函数，则参数一定要不一样。

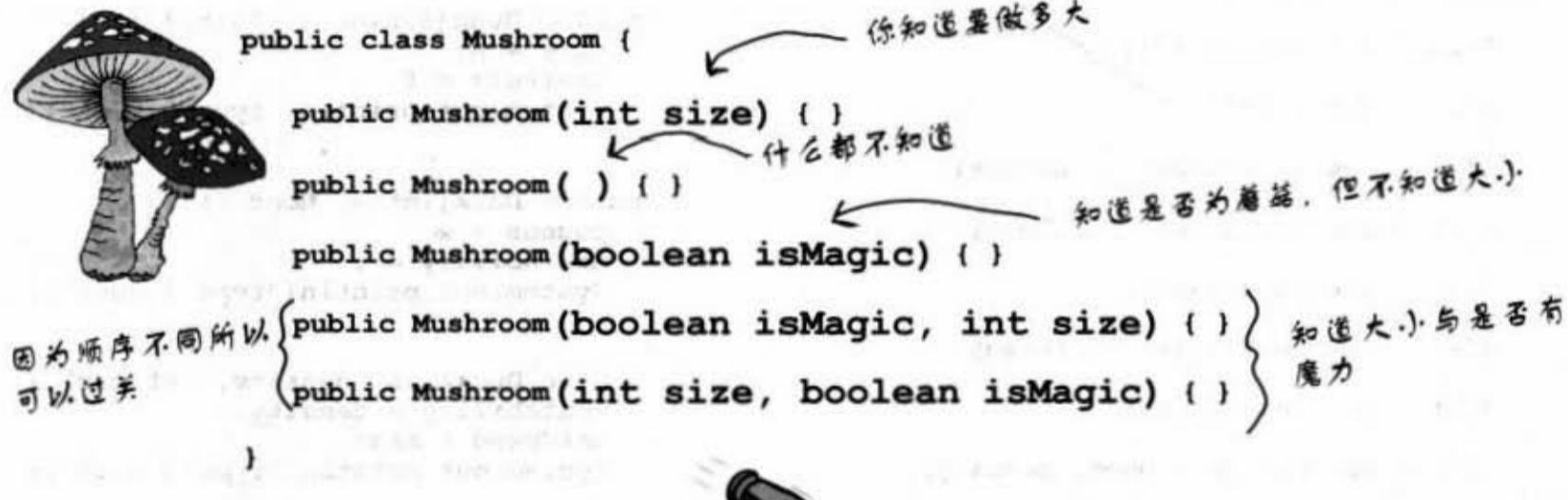
这包括了参数的顺序与类型，只要是不一样就可以。这就跟方法的重载是相同的，不过细节会留到其他的章节再讨论。



重载构造函数的意思代表你有一个以上的构造函数且参数都不相同

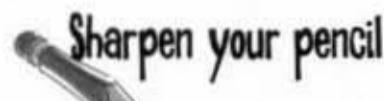
下面列出的构造函数都是合法的，因为参数都不相同。假设说有两个构造函数的参数都是只有一个int，则肯定无法通过编译程序。编译器看的是参数的类型和顺序而不是参数的名字。你可以做出相同类型但是顺序不同的参数。使用String以及int型的参数顺序与使用int以及String型的参数顺序是不同的。

4个不同的构造函数代表
有4种不同的创建方式



要点

- 实例变量保存在所属的对象中，位于堆上。
- 如果实例变量是个对对象的引用，则引用与对象都是在堆上。
- 构造函数是个会在新建对象的时候执行程序代码。
- 构造函数必须与类同名且没有返回类型。
- 你可以用构造函数来初始被创建对象的状态。
- 如果你没有写构造函数，编译器会帮你安排一个。
- 默认的构造函数是没有参数的。
- 如果你写了构造函数，则编译器就不会调用。
- 最好能有无参数的构造函数让人可以选择使用默认值。
- 重载的构造函数意思是超过一个以上的构造函数。
- 重载的构造函数必须有不同的参数。
- 两个构造函数的参数必须不同。
- 实例变量有默认值，原始的默认值是 0/0.0/false，引用的默认值是 null。



猜猜看哪个新建Duck()会用到哪个构造函数？我们已经帮你找到一个。

```
public class TestDuck {
    public static void main(String[] args) {
        int weight = 8;
        float density = 2.3F;
        String name = "Donald";
        long[] feathers = {1,2,3,4,5,6};
        boolean canFly = true;
        int airspeed = 22;

        Duck[] d = new Duck[7];
        d[0] = new Duck();
        d[1] = new Duck(density, weight);
        d[2] = new Duck(name, feathers);
        d[3] = new Duck(canFly);
        d[4] = new Duck(3.3F, airspeed);
        d[5] = new Duck(false);
        d[6] = new Duck(airspeed, density);
    }
}
```

```
class Duck {
    int pounds = 6;
    float floatability = 2.1F;
    String name = "Generic";
    long[] feathers = {1,2,3,4,5,6,7};
    boolean canFly = true;
    int maxSpeed = 25;

    public Duck() {
        System.out.println("type 1 duck");
    }

    public Duck(boolean fly) {
        canFly = fly;
        System.out.println("type 2 duck");
    }

    public Duck(String n, long[] f) {
        name = n;
        feathers = f;
        System.out.println("type 3 duck");
    }

    public Duck(int w, float f) {
        pounds = w;
        floatability = f;
        System.out.println("type 4 duck");
    }

    public Duck(float density, int max) {
        floatability = density;
        maxSpeed = max;
        System.out.println("type 5 duck");
    }
}
```

问：先前你说过最好要有没参数的构造函数以便让用户可以调用使用我们提供默认值的构造函数。但若不可能提供默认值的时候还应该要提供无参数的构造函数吗？

答：没错。有时候有默认值的无参数构造函数是不合理的。在Java API中有些类就没有无参数的构造函数，比如说Color这个类。它是用来设定字型或GUI按钮的颜色。当你制作Color的实例时，该实例会代表特定的颜色。如果你要用到Color，就必须以某种方式指定颜色：

```
Color c = new Color(3, 45, 200);
```

这是使用3个int来代表RGB三色的构造函数，后面讨论Swing的章节会有说明。如果没有给颜色，那Java API的设计人也许可以给你一个预设桃红色，想要吗？如果以这种方式创建Color对象：

```
Color c = new Color();
```

编译器会向你抱怨没有这样的构造函数：

```
File Edit Window Help StopBeingStupid
cannot resolve symbol
:constructor Color()
location: class java.awt.Color
Color c = new Color();
^
1 error
```

构造函数

迷你小回顾：

- 构造函数是在新建类时会执行的程序。

```
Duck d = new Duck();
```

- 构造函数必须与类的名字一样，且没有返回类型。

```
public Duck(int size) { }
```

- 如果你没有写构造函数，则编译器会帮你写一个没有参数的

```
public Duck() { }
```

- 一个类可以有很多个构造函数，但不能有相同的参数类型和顺序，这叫作重载过的构造函数。

```
public Duck() { }
public Duck(int size) { }
public Duck(String name) { }
public Duck(String name, int size) { }
```

研究显示挑战智力大考验最多可以提升神经的大小到 42% 以上。



想到父类吗？

在创建 Dog 的时候，Canine 的构造函数是否应该要执行？

如果父类是抽象的，那它可以有构造函数吗？

接下来会看到这些主题，你现在应该独立地思考一下构造函数与父类之间的关系。

there are no
Dumb Questions

问： 构造函数应该是公有的吗？

答： 不。构造函数可以是公有、私有或不指定的。第 16 章和附录 B 有关于不指定的预设存取权讨论。

问： 一个私有的构造函数有什么作用？没有人能够调用它，所以也就没有人能够创建该对象？

答： 不是这么说的。私有不是完全不能存取，它代表该类以外不能存取。这听起来很矛盾吧？下一章会讨论这个问题。

等一下……我们都还没有谈到父类以及继承与构造函数之间的关系

这样才有趣。还记得上一章我们说到Snowboard对象把Object部分包在自己的核心吗？那个讨论的重点在于每个对象不只是保存自行声明的变量，还有从父类来的所有东西（至少会带有Object，因为每个类都有继承过这个类）。

因此在创建某个对象时（完全没有其他方法能够创建对象，只能通过new来产生新对象），对象会取得所有实例变量所需的空间，这当然也包括一路继承下来的东西。想象一下，父类也许会有一个setter以包装私用的变量。但此变量必须有空间。概念上，你可以把整个情境用下面的图来表示，对象就像洋葱是有层次的（请见《史瑞克》），每一层都代表某一级的父类。



父类的构造函数在对象的生命中所扮演的角色

在创建新对象时，所有继承下来的构造函数都会执行。

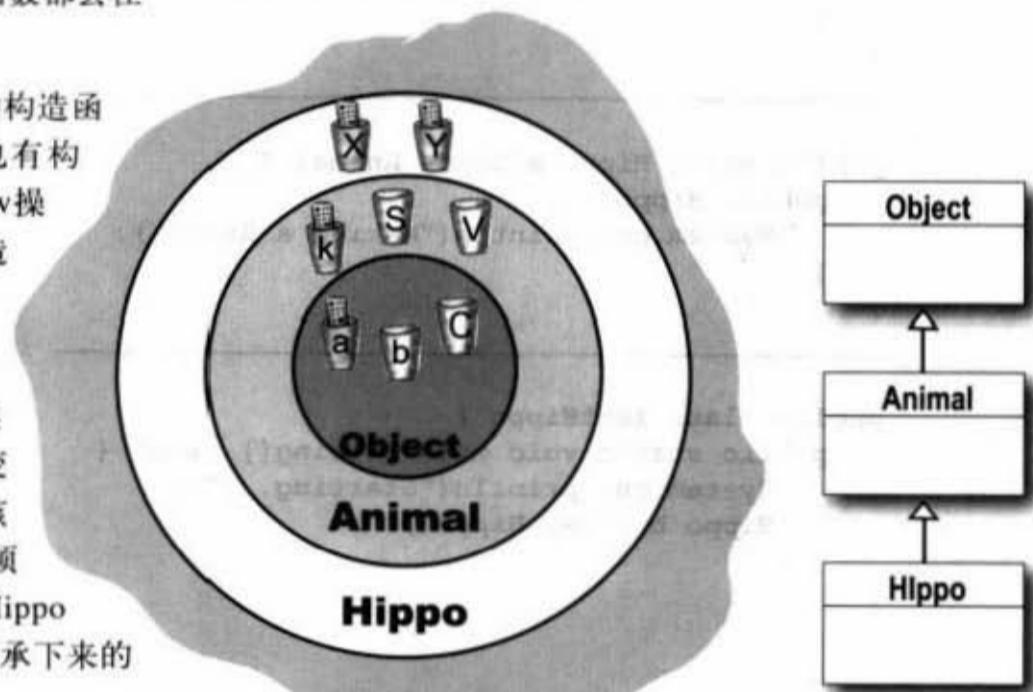
这代表每个父类都有一个构造函数（因为每个类至少都会有一个构造函数），且每个构造函数都会在子类对象创建时期执行。

执行new的指令是个重大事件，它会启动构造函数连锁反应。还有，就算是抽象的类也有构造函数。虽然你不能对抽象的类执行new操作，但抽象的类还是父类，因此它的构造函数会在具体子类创建出实例时执行。

在构造函数中用super调用父类的构造函数的部分。要记得子类可能会根据父类的状态来继承方法（也就是父类的实例变量）。完整的对象需要也是完整的父类核心，所以这就是为什么父类构造函数必须执行的原因。就算Animal上有些变量是Hippo不会用到的，但Hippo可能会用到某些继承下来的方法必须读取Animal的实例变量。

构造函数在执行的时候，第一件事是去执行它的父类的构造函数，这会连锁反应到Object这个类为止。

我们在接下来的几页会看到父类构造函数是如何被调用，以及你如何自行调用它们。你也会知道要如何调用有参数的父类构造函数。



在堆上的Hippo对象

Hippo对象IS-A Animal同时也IS-A Object。如果你要创建出Hippo，也得创建出Animal与Object的部分。

这样的过程被称为“构造函数链（Constructor Chaining）”

创建Hippo也代表创建Animal与Object

```
public class Animal {
    public Animal() {
        System.out.println("Making an Animal");
    }
}

public class Hippo extends Animal {
    public Hippo() {
        System.out.println("Making a Hippo");
    }
}

public class TestHippo {
    public static void main (String[] args) {
        System.out.println("Starting...");
        Hippo h = new Hippo();
    }
}
```

Sharpen your pencil

哪一个输出才是对的？执行左边的程式代码得到A还是B的结果？

答案就在下面。

A

```
File Edit Window Help Swear
% java TestHippo
Starting...
Making an Animal
Making a Hippo
```

B

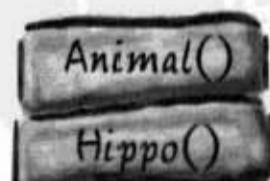
```
File Edit Window Help Swear
% java TestHippo
Starting...
Making a Hippo
Making an Animal
```

① 某个程序执行new Hippo()的动作，Hippo()的构造函数进入堆栈最上方的堆栈块

② Hippo()调用父类的构造函数导致Animal()的构造函数进入栈顶

③ Animal()调用父类的构造函数导致Object()的构造函数进入栈顶

④ Object()执行完毕，它的堆栈块被弹出，接着继续执行Animal()的



后一个执行完毕。

答案是A，Hippo()是最后调用的，但Object()是第一个调用的。

如何调用父类的构造函数？

以Duck的构造函数来说，你也许认为它会直接调用Animal()，但实际上不是这样的：

```
public class Duck extends Animal {
    int size;

    public Duck(int newSize) {
        → Animal(); ← 这不合法
        size = newSize;
    }
}
```

调用父类构造函数的唯一方法是调用super()。

它看起来会像下面这样：

```
public class Duck extends Animal {
    int size;

    public Duck(int newSize) {
        super(); ← 要这么调用
        size = newSize;
    }
}
```

在你的构造函数中调用super()会把父类的构造函数放在堆栈的最上方。你猜父类的构造函数会做什么？它会调用它的父类构造函数。这会一路上去直到Object的构造函数为止。然后再一路执行、弹出回到原来的构造函数。

如果我们没有调用super()会发生什么事？

你也许已经猜到了。

编译器会帮我们加上super()的调用。

所以编译器有两种涉及构造函数的方式：

● 如果你没有编写构造函数。

```
public ClassName() {
    super();
}
```

● 如果你有构造函数但没有调用super()。

编译器会帮你对每个重载版本的构造函数加上下面这种调用：

super();

编译器帮忙加的一定会是没有参数的版本，假使父类有多个重载版本，也只有无参数的这个版本会被调用到。

小孩能够在父母之前出生吗？

如果你把父类想象成子类的父母，那就可以看出来谁是先存在的。父类的部分必须在子类创建完成之前就必须完整地成型。记住，子类对象可能需要动用到从父类继承下来的东西，所以那些东西必须要先完成。父类的构造函数必须在子类的构造函数之前结束。

再看一下252页的堆栈，你会发现Hippo的构造函数是第一个被调用的（在堆栈上的第一个），却也是最后一个完成的！每个子类的构造函数会立即调用父类的构造函数，如此一路往上直到Object。等到Object完成后会回去执行Animal的，然后等Animal完成后又回去执行Hippo剩下的构造函数。这是因为：

对super()的调用必须是构造函数的第一个语句*。



类Boop的可能构造函数

```
R public Boop() {  
    super();  
}
```

```
R public Boop(int i) {  
    super();  
    size = i;  
}
```

```
R public Boop() {  
}
```

```
R public Boop(int i) {  
    size = i;  
}
```

```
🚫 public Boop(int i) {  
    size = i;  
    super();  
}
```

这样做也可以，因为编译器会自动把super()加到最前面

这就不行了，编译器不会让你这么做

*有异常情况，见256页

有参数的父类构造函数

如果父类的构造函数有参数该怎么办？你能够传值进去吗？如果不行的话，则没有无参数构造函数的类将不能被继承。想象这个情景：所有的动物都有名字。所以Animal这个类有个getName()可以返回name实例变量的值。此实例变量是被标记为私有的，但Hippo这个子类有把getName()继承下来。问题来了：

Hippo有getName()这个方法但是没有name实例变量。Hippo要靠Animal的部分来维持name实例变量，然后从getName()来返回这个值，但Animal要如何取得这个值呢？唯一的机会是通过super()来引用父类，所以要从这里把name的值传进去，让Animal把它存到私有的name实例变量中。

```
public abstract class Animal {
    private String name; ← 每个Animal都会有名字

    public String getName() { ← Hippo会继承这个getter
        return name;
    }

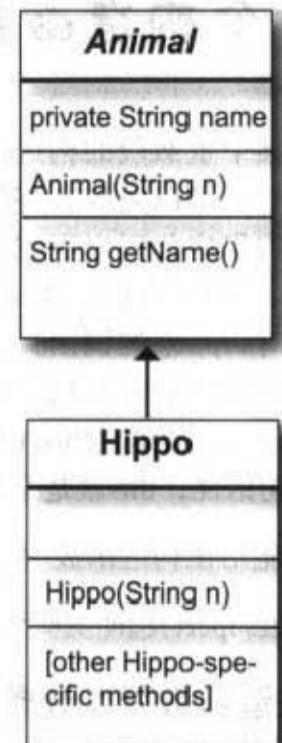
    public Animal(String theName) {
        name = theName; ← 有参数的构造函数，用来
    }
}
```

```
public class Hippo extends Animal {

    public Hippo(String name) {
        super(name); ← 这个构造函数会要求
    } ← 名称
}

传给Animal的构造函数
```

```
public class MakeHippo {
    public static void main(String[] args) {
        Hippo h = new Hippo("Buffy"); ← 创建Hippo，传入名字然后
        System.out.println(h.getName()); ← 再列出来看看
    }
}
```



人家的名字是珍妮弗啦！
所以应该是用这个名字给
super()才对



```
File Edit Window Help Hide
%java MakeHippo
Buffy
```

从某个构造函数调用重载版的另一个构造函数

如果有某个重载版的构造函数除了不能处理不同类型的参数之外，可以处理所有的工作，那要怎么办？你不想让相同的程序代码出现在每个构造函数中（维护起来很麻烦），所以你想把程序代码只摆在某个构造函数中（包括对super()的调用）。如此一来，所有的构造函数都会先调用该构造函数，让它来执行真正的构造函数。这很容易，只要调用this()或this(aString)或this(27, x)就行。换句话说，this就是个对对象本身的引用。

this()只能用在构造函数中，且它必须是第一行语句！

这样会跟super()起冲突吗？所以你必须选择：

每个构造函数可以选择调用super()或this()，但不能同时调用！

使用this()来从某个构造函数调用同一个类的另外一个构造函数。

this()只能用在构造函数中，且必须是第一行语句。

super()与this()不能兼得。

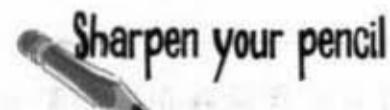
```
class Mini extends Car {  
  
    Color color;  
  
    public Mini() {  
        this(Color.Red); ←  
    }  
  
    public Mini(Color c) {  
        super("Mini"); ←  
        color = c;  
        // 初始化动作  
    }  
  
    public Mini(int size) {  
        this(Color.Red); ←  
        super(size);  
    }  
}
```

无参数的构造函数以默认的颜色调用真正的构造函数

这才是真正的构造函数

有问题！不能同时调用super()和this()，两者只能有一个会是第一行语句

```
File Edit Window Help Drive  
javac Mini.java  
Mini.java:16: call to super must  
be first statement in constructor  
        super();  
               ^
```



下面类SonOfBoo的构造函数中有某几个无法通过编译，试试看你能否找到不合法的是哪几个。在有问题的构造函数旁边划条线连到右边的错误信息上。

```
public class Boo {
    public Boo(int i) { }
    public Boo(String s) { }
    public Boo(String s, int i) { }
}
```

```
class SonOfBoo extends Boo {
    public SonOfBoo() {
        super("boo");
    }

    public SonOfBoo(int i) {
        super("Fred");
    }

    public SonOfBoo(String s) {
        super(42);
    }

    public SonOfBoo(int i, String s) {
    }

    public SonOfBoo(String a, String b, String c) {
        super(a,b);
    }

    public SonOfBoo(int i, int j) {
        super("man", j);
    }

    public SonOfBoo(int i, int x, int y) {
        super(i, "star");
    }
}
```



```
File Edit Window Help
%javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo
(java.lang.String,java.lang.String)
```

```
File Edit Window Help Yadayadayada
%javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo
(int,java.lang.String)
```

```
File Edit Window Help ImNotListening
%javac SonOfBoo.java
cannot resolve symbol
symbol:constructor Boo()
```

我们已经了解了对象的诞生过程，但对象会存活多久呢？

对象的生命周期完全要看引用到它的“引用”。如果引用还活着，则对象也会继续活在堆上。如果引用死了（稍后会解释），则对象就会跟着殉情……陪葬……送命……

如果对象生命周期要看引用变量的生命周期而定，那变量到底会活多久？

这又要看它是局部变量或实例变量而定。下面的程序展示出局部变量的生命周期。

```
public class TestLifeOne {  
    public void read() {  
        int s = 42; ← s的范围只限于read()里面。  
        sleep();  
    }  
  
    public void sleep() {  
        s = 7;  
    } ↑ 非法使用！  
}
```

sleep()
read() s

s 还活着，但范围仅限于read()。当sleep()执行完毕回到read()时，read()还是能使用s，但当read()执行完毕被弹出时，家属就得帮s举办追悼会。

① 局部变量只会存活在声明该变量的方法中

```
public void read() {  
    int s = 42;  
    // 's' 只能用在此方法中  
    // 当方法结束时  
    // s会完全消失
```

变量s只能用在read()方法中。换句话说，此变量的范围只会在所属方法的范围内。其余的程序代码完全见不到s。

② 实例变量的寿命与对象相同。如果对象还活着，则实例变量也会是活的。

```
public class Life {  
    int size;  
  
    public void setSize(int s) {  
        size = s;  
        // 's' 会在方法结束时  
        // 消失，但size在类中  
        // 到处都可用  
    }  
}
```

此时s变量（这次是方法的参数）的范围同样也只限制在所属的setSize()这个方法中。

“life”与“scope”的差别

Life

只要变量的堆栈块还存在于堆栈上，局部变量就算活着。也就是说，活到方法执行完毕为止。

Scope

局部变量的范围只限于声明它的方法之内。当此方法调用别的方法时，该变量还活着，但不在目前的范围内。执行其他方法完毕返回时，范围也就跟着回来。

```
public void doStuff() {
    boolean b = true;
    go(4);
}

public void go(int x) {
    int z = x + 24;
    crazy();
    // 这里有更多的代码
}

public void crazy() {
    char c = 'a';
}
```



① `doStuff()`运行在堆栈，变量`b`存活于scope中。

② 调用`go()`，`x`，`z`，`b`都活着，但只有`b`不在它的范围内。

③ 调用`crazy()`，只有`c`在它的范围内。

④ 完成`crazy()`，`c`既不在它的范围内，也没活下来，只有`x`和`z`在它的范围内。

当局部变量活着的时候，它的状态会被保存。只要`doStuff()`还在堆栈上，`b`变量就会保持它的值。但`b`变量只能在`doStuff()`待在栈顶时才能使用。也就是说局部变量只能在声明它的方法在执行中才能被使用。

那引用变量呢？

引用的规则与 primitive 主数据类型相同。引用变量只能在处于它的范围内才能被引用，也就是说除非引用变量是在它的范围内，不然就不能使用对象的遥控器。真正的问题是：

“变量的生命周期如何影响对象的生命周期？”

只要有活着的引用，对象也就会活着。如果某个对象的引用已经不在它的范围内，但此引用还是活着的，则此对象就会继续活在堆上。

如果对对象的唯一引用死了，对象就会从堆中被踢开。引用变量会跟堆栈块一起解散，因此被踢开的对象也就正式的声明出局。关键在于知道何时对象会变成可被垃圾收集器回收的。

一旦对象符合垃圾收集器 (GC) 的条件，你就无需担心回收内存的问题。如果程序内存不足，GC 就会去歼灭部分或全部的可回收对象。你可能还是会遇到内存不足的状况，但这要等到所有可回收的都被回收掉也还不够的时候才会发生。你要注意的是对象用完了就要抛弃，这样才能让垃圾收集器有东西可以回收。如果你把持着对象不放，垃圾收集器也帮不了什么忙。

除非有对对象的引用，否则该对象一点意义也没有。

如果你无法取得对象的引用，则此对象只是浪费空间罢了。

但若对象是无法取得的，GC 会知道该怎么做的。那种对象迟早会葬送在垃圾收集器的手上。



当最后一个引用消失时，对象就会变成可回收的。

有3种方法可以释放对象的引用：

① 引用永久性的离开它的范围。

```
void go() {  
    Life z = new Life();  
}
```

z会在方法结束时
消失

② 引用被赋值到其他的对象上。

```
Life z = new Life();  
z = new Life();
```

第一个对象会在z被赋值
到别处时挂掉

③ 直接将引用设定为null。

```
Life z = new Life();  
z = null;
```

第一个对象会在z被赋值
为null时击毙

对象杀手一号

引用永久性的
离开它的范围



```
public class StackRef {
    public void foof() {
        barf();
    }

    public void barf() {
        Duck d = new Duck();
    }
}
```

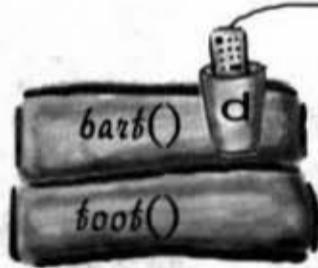
我真地很讨厌这个
例子。



- 1 foof()被推到堆栈上，
没有声明变量

foof()

- 2 barf()被推到堆栈上，
创建一个对象以及对它
的引用



新的Duck会放在堆上。
只要barf()还在运行，
d就还活着，所以鸭子也
就没事

- 3 barf()执行完毕，因此
d也就挂了



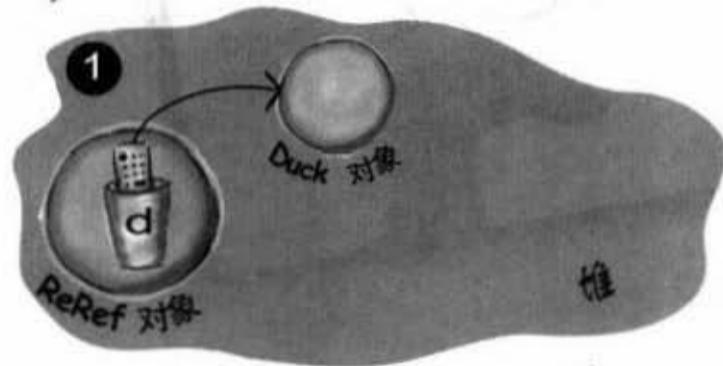
既然d已经不存在了，
Duck也就等着要
被垃圾收集器回收

对象杀手二号

引用被赋值到
其他的对象上

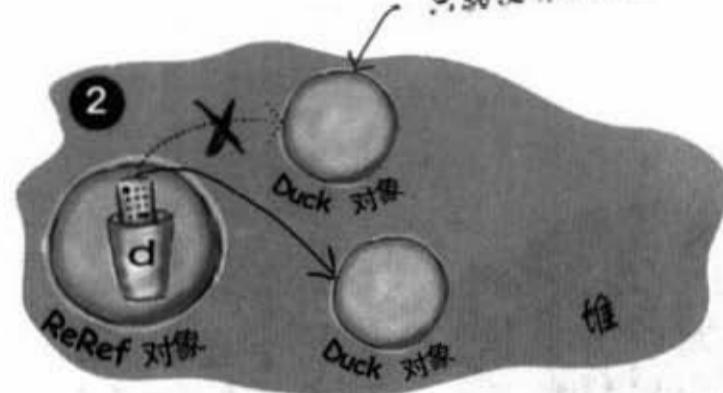


```
public class ReRef {  
    Duck d = new Duck();  
  
    public void go() {  
        d = new Duck();  
    }  
}
```



新的Duck会放在堆上，只要ReRef还活着，d就没事，除非……

考古学家发现四千年前的对象



既然d引用到其他的Duck，第一个Duck就跟死掉是一样的

哇！难道它们不知道
可以重置引用吗？也许古
人真的很不喜欢内存的管理
工作。



物件杀手三号

直接将引用

设定为null



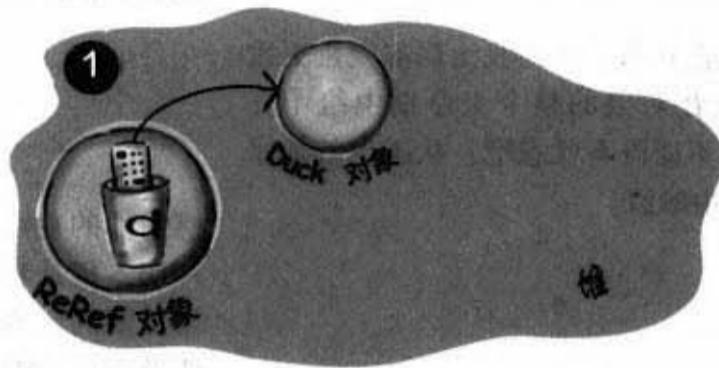
```
public class ReRef {
    Duck d = new Duck();
    public void go() {
        d = null;
    }
}
```

null的真相

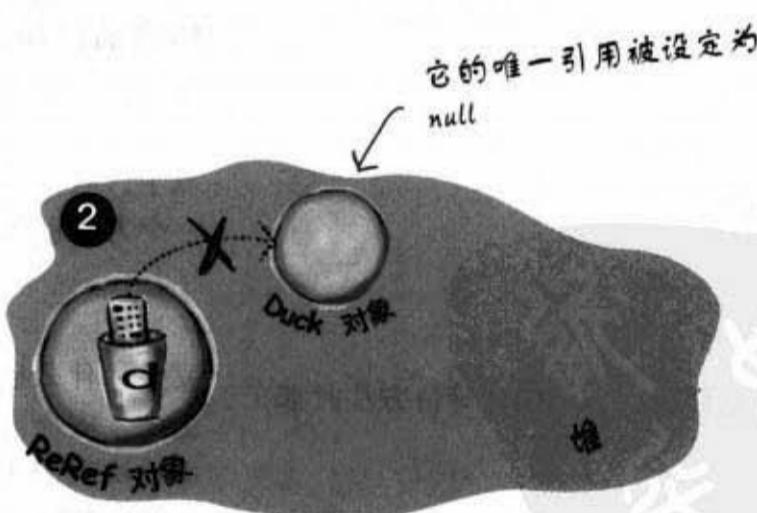
当你把引用设为null时，你就等于是抹除遥控器的功能。换句话说，你会拿到一个没有电视的遥控器。null是代表“空”的字节组合（实际上是什么只有Java虚拟机才会知道）。

如果你真地按下这种遥控器上的按钮，什么事情也不会发生。但在Java上，你是不能对null引用按钮的。因为Java虚拟机会知道（这是运行期，不是编译时的错误）你期待喵喵叫，但是却没有Cat可以执行！

对null引用使用圆点运算符会在执行期遇到NullPointerException这样的错误。后面会有讨论异常的章节。

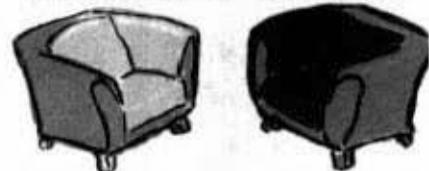


新的Duck会放在堆上，只要ReRef还活着，d就没事，除非……



它的唯一引用被设定为 null
d被设定成null，就好像没有操作对象的遥控器

Fireside Chats



实例变量

我想应该先从我开始，因为我比局部变量更重要。通常我会在对象的整个生命期中给予支持。毕竟对象不能没有状态吧？状态也就是保持在实例变量中的值。

误会可大了。我了解你在方法中的作用，只是你的命真的太短了。那也就是为何人们把你叫做“临时变量”的原因。

抱歉，我知道了。

我倒不知道这回事，那你们在等待方法时都在做什么？

今晚的话题： 实例变量与局部变量的生死对谈

局部变量

感谢你的观点，也感谢你对对象状态值所做的一切。但是我不想让大家误会。局部变量是相当重要的。容我引用你的说法：“毕竟对象不能没有行为吧？行为也就是保持在方法中的算法”。你也一定很清楚必须要有些局部变量才能让算法运作。

留点口德吧，“临时变量”在我们这边是很轻蔑的说法。我们比较喜欢自称“局部变量”、“栈变量”或者“变量作用域”。

算了。我们是不长命，但有时一个方法调用另一个方法也会让我们待在堆栈上很久。

发呆，什么也不做。但是我们所保存的值不会丢失，安全得很。要等到执行回到我们所处的堆栈块我们才会继续活动，不过这也代表我们又往生命终点迈进了一步。

实例变量

我看过你同伴出殡的新闻报导，看起来蛮惨的。我的意思是当方法执行完成后，堆栈块就直接被清除，那种死法应该很痛吧。

我跟对象一起生存在堆上，离西贡街与公众四方街口不远……嗯，其实是住在给我保存状态的对象里面才对。那附近地段很贵，物价也很高。

对啦，如果我是个猫对象上的实例变量所引用的跳蚤对象，当该变量被设定成null的时候，我就只好等着被收进垃圾堆，我的地方也会让出来给别人用。不过有人跟我说过，这猫不可能会洗澡的。

你不怕遇到警察吗？

局部变量

你还说呢。这叫做被弹出好吗？为什么不来说说你呢？我生存在堆栈上，你老兄生存哪啊？

但你也不一定会很长寿吧？例如说你是猫对象身上的跳蚤实例变量所引用的跳蚤对象。假使今天这猫执行了洗澡的行为，使得跳蚤属性被设定为null，那会发生什么事？

你就这么相信了？如果猫对象只有一个引用呢？假使这个唯一的引用是保存在局部变量中呢？如果声明此局部变量的方法执行完毕，你还不是得像我们一样。老实跟你说，我已经认命了，现在能够活一天就活一天，有机会喝喝酒、吃吃RAM，我就尽量吃喝。

我有一招可以逃过警察，你可以学学。如果你遇到警察的时候，就把……



我是垃圾收集器

假设批注部分实际上会是执行足够长时间的过程调用，下方右边有哪几行程序代码加到左边A位置会使得某一个额外的对象被认为是可以垃圾回收的？

```

public class GC {
    public static GC doStuff() {
        GC newGC = new GC();
        doStuff2(newGC);
        return newGC;
    }

    public static void main(String [] args) {
        GC gc1;
        GC gc2 = new GC();
        GC gc3 = new GC();
        GC gc4 = gc3;
        gc1 = doStuff();
        A
        // 调用更多的方法
    }

    public static void doStuff2(GC copyGC) {
        GC localGC
    }
}

```

1 copyGC = null;
2 gc2 = null;
3 newGC = gc3;
4 gc1 = null;
5 newGC = null;
6 gc4 = null;
7 gc3 = gc2;
8 gc1 = gc4;
9 gc3 = null;



最受欢迎 金对象奖

```

class Bees {
    Honey [] beeHA;
}

class Raccoon {
    Kit k;
    Honey rh;
}

class Kit {
    Honey kh;
}

class Bear {
    Honey hunny;
}

public class Honey {
    public static void main(String [] args) {
        Honey honeyPot = new Honey();
        Honey [] ha = {honeyPot, honeyPot, honeyPot, honeyPot};
        Bees b1 = new Bees();
        b1.beeHA = ha;
        Bear [] ba = new Bear[5];
        for (int x=0; x < 5; x++) {
            ba[x] = new Bear();
            ba[x].hunny = honeyPot;
        }
        Kit k = new Kit();
        k.kh = honeyPot;
        Raccoon r = new Raccoon(); ← 新的Raccoon对象
        r.rh = honeyPot; ← 引用它的变量!
        r.k = k;
        k = null;
    } // main函数结束
}

```

下面的程序代码新建出数个对象。你的任务是要找出“最受欢迎”的对象，也就是被最多变量所引用的对象。还有，标记出所有对象以及对它的引用。我们已经先帮你标记出一个对象。



华安 传奇实录



“我们已经测了4次，主要模块的温度还是持续下降”，秋香不耐烦地说着，“上礼拜就已经安装新的温度感应器，散热器（radiator）的读数应该也没有问题，所以我们把焦点放在恒温器（retention bot）上”。华安叹了一口气，想起一开始的时候还以为纳米技术可以帮他们超前进度。现在离发射只剩下5个礼拜，卫星的生存维护装置还是没有办法通过测试。

“你用什么比例来模拟？”，华安问到。

“我知道你要问什么”，秋香回答，“我们也想到了。如果不符合规格，任务控制中心不会验收的。我们必须以2:1的比例执行v3版和v2版的radiator测试单元”，秋香继续说，“整体来说，retention与radiator的比例应该是4:3”。

华安追问：“能源消耗呢？”。秋香想了一下：“那是另外一问题，能源消耗速度比预期的快。我们有另外一组人正在想办法。但是这些无线技术很难将radiator的能源消耗与retention的消耗分离”。

秋香停了一下又继续说：“设计上的整体能源消耗率应该是3:2，也就是radiator的能源消耗会比较快”。

“好吧，既然这样”，华安说，“那我们再看一下仿真程序初始化的程序代码。这个问题一定得要很快地解决！”

```

import java.util.*;
class V2Radiator {
    V2Radiator(ArrayList list) {
        for(int x=0; x<5; x++) {
            list.add(new SimUnit("V2Radiator"));
        }
    }
}

class V3Radiator extends V2Radiator {
    V3Radiator(ArrayList llist) {
        super(llist);
        for(int g=0; g<10; g++) {
            llist.add(new SimUnit("V3Radiator"));
        }
    }
}

class RetentionBot {
    RetentionBot(ArrayList rlist) {
        rlist.add(new SimUnit("Retention"));
    }
}

```

华安 传奇实录 (续)

```

public class TestLifeSupportSim {
    public static void main(String [] args) {
        ArrayList aList = new ArrayList();
        V2Radiator v2 = new V2Radiator(aList);
        V3Radiator v3 = new V3Radiator(aList);
        for(int z=0; z<20; z++) {
            RetentionBot ret = new RetentionBot(aList);
        }
    }
}

class SimUnit {
    String botType;
    SimUnit(String type) {
        botType = type;
    }
    int powerUse() {
        if ("Retention".equals(botType)) {
            return 2;
        } else {
            return 4;
        }
    }
}

```

把程序看了一遍之后，华安露出了得意的笑容。他说：“我想我知道那是怎么回事了。能源消耗这件事情其实也是有关的！”

到底华安看出哪里有问题？那要如何解决这个bug呢？

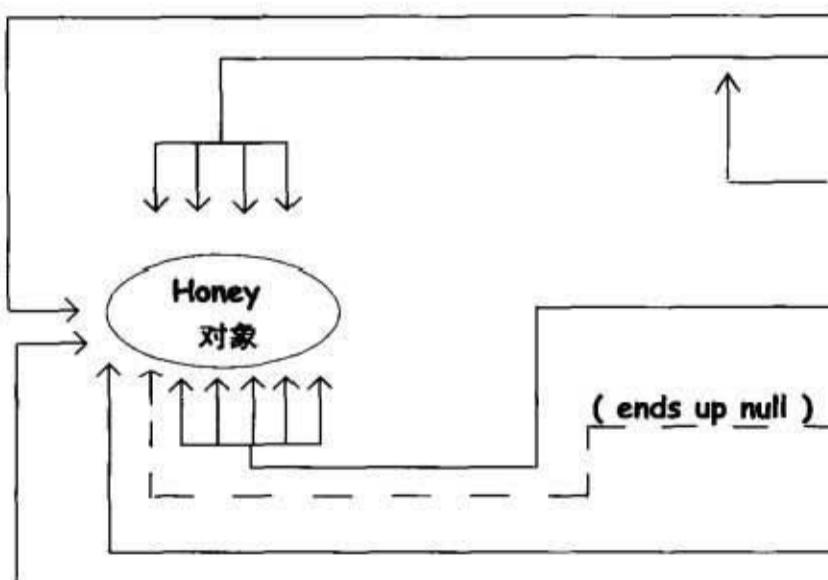


G.C.

- | | | |
|---|----------------|-------------------------------|
| 1 | copyGC = null; | 不行——这一行程序尝试要存取已经不在范围内
的变量。 |
| 2 | gc2 = null; | 可以—— gc2 是该对象唯一的引用。 |
| 3 | newGC = gc3; | 不行——也是超出范围。 |
| 4 | gc1 = null; | 可以—— gc1 是唯一的引用。 |
| 5 | newGC = null; | 不行—— newGC 已经超出范围。 |
| 6 | gc4 = null; | 不行——还有 gc3 引用该对象。 |
| 7 | gc3 = gc2; | 不行——还有 gc4 引用该对象。 |
| 8 | gc1 = gc4; | 可以——重新给对象引用赋值。 |
| 9 | gc3 = null; | 不行—— gc4 还在引用该对象。 |

最受欢迎 金对象奖

要指出Honey这个对象是这个类中最受欢迎的对象应该不难。但要看出这些变量都指向同一个对象其实是要动点脑筋的。



```

public class Honey {
    public static void main(String [] args) {
        Honey honeyPot = new Honey();
        Honey [] ha = {honeyPot, honeyPot,
                      honeyPot, honeyPot};
        Bees b1 = new Bees();
        b1.beeHA = ha;
        Bear [] ba = new Bear[5];
        for (int x=0; x < 5; x++) {
            ba[x] = new Bear();
            ba[x].hunny = honeyPot;
        }
        Kit k = new Kit();
        k.kh = honeyPot;
        Raccoon r = new Raccoon();
        r.rh = honeyPot;
        r.k = k;
        k = null;
    } } // main函数结束
  
```



华安 传奇实录

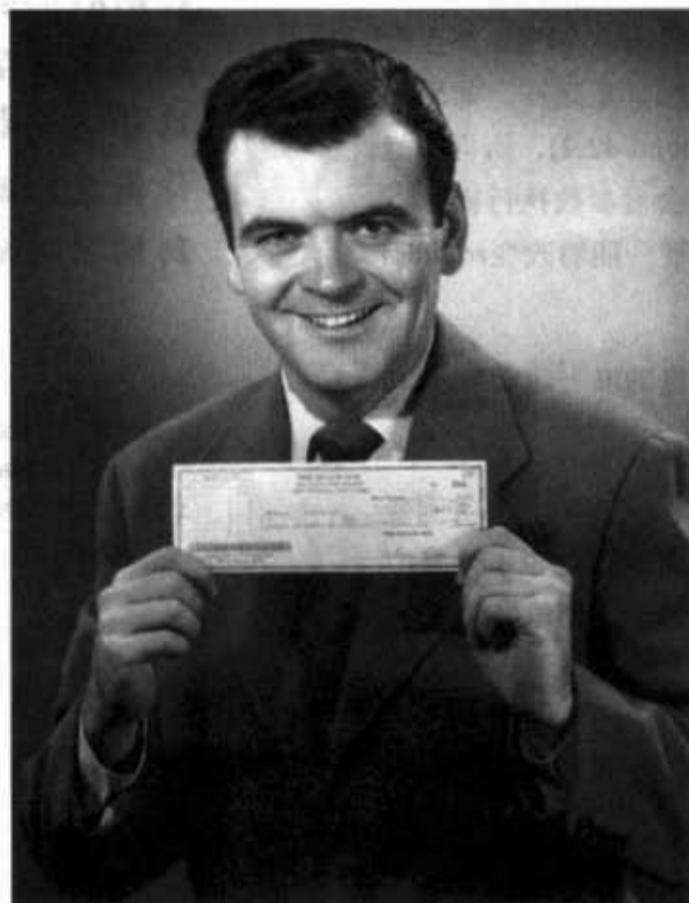
华安发现V2Radiator类的构造函数会取用一个ArrayList参数。这代表每次V3Radiator的构造函数被调用时，它会在对V2Radiator的super()调用中传入一个ArrayList。这样会额外多出5个V2Radiator的SimUnit。如此一来，总体能源消耗会是120而不是秋香预期的100。

因为每个Bot都会创建出SimUnit，所以在SimUnit的构造函数中加上一栏输出就能够很快的发现问题的来源！



java学习群：72030155，每天20:30-23:00都有大神视频教学，想学习的同学可以进群免费听课！

数字很重要



盘算一下吧。除了primitive主数据类型的运算之外，数字还有其他的工作。你可能需要对数字计算绝对值、取整。或许需要以小数点后两位的打印格式，或者每隔三位数加上逗点以方便阅读。还有日期也是，你可能需要用某种特定的格式来打印日期，或者对日期作运算，比如说“今天起的两个礼拜后”。此外，字符串要如何转换成数字呢？幸好Java API中有很多与数字有关的方法能够很方便地使用。但这些方法多为静态的，因此我们会先从静态的变量和方法开始说起——这也包括了静态的final变量这种Java常数。

Math方法：

最接近全局的方法

虽然在Java中没有东西是全局（global）的。但可以这么想：一种方法的行为不依靠实例变量值。例如Math这个类中的round()方法。它永远都执行相同的工作——取出浮点数（方法的参数）的最接近整数值。永远都是这样。如果你有10000个Math的实例，并且都执行round(42.2)，所得的值永远都会是42。换句话说，这个方法会对参数执行操作，但这操作不受实例变量状态的影响。唯一能够改变round()行为的只有所传入的参数。

你不觉得为了要执行round()而得在宝贵的堆上建立Math的实例是很浪费的事吗？

像round()、abs()、max()等数学运算方法其实不需要实例变量值。事实上也不会有Math的实例变量。因此也不会有空间被它的实例所占用。你猜怎么样？你不需创建Math的实例，实际上你也无法创建。

如果你硬要创建Math的实例：

```
Math mathObject = new Math();
```

会得到下面这样的错误信息：

```
File Edit Window Help IwasToldThereWouldBeNoMath
%javac TestMath
TestMath.java:3: Math() has private
access in java.lang.Math
    Math mathObject = new Math(); ^

1 error
```

错误信息指出Math的构造函数被标记为私有的！这代表你不能新建Math的对象

在**Math**这个类中的所有方法都不需要实例变量值。因为这些方法都是静态的，所以你无需**Math**的实例。你会用到的只有它的类本身。

```
int x = Math.round(42.2);
int y = Math.min(56,12);
int z = Math.abs(-343);
```

↑
这些方法无需实例变量，因此也不需要特定对象来辨别行为。

非静态方法与静态方法的差别

Java是面向对象的，但若处于某种特殊的情况下，通常是实用方法，则不需要类的实例。static这个关键词可以标记出不需类实例的方法。一个静态的方法代表说“一种不依靠实例变量也就不需要对象的行为”。

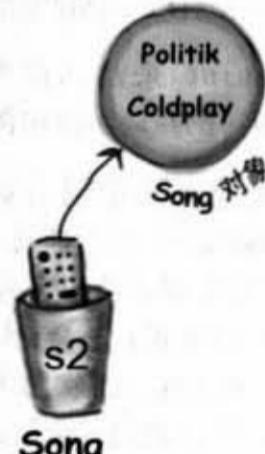
非静态方法

```
public class Song {
    String title; ← 实例变量的值会影响到 play()方法的行为
    public Song(String t) {
        title = t;
    }
    public void play() {
        SoundPlayer player = new SoundPlayer();
        player.playSound(title);
    }
}
```

Song
title
play()

有两个Song的实例

↑
title的值会决定play()方法所播放的内容



它会播放Politik
s2.play();

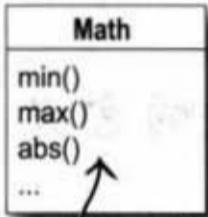
它会播放My Way (没听过魔克的歌吗？那别说你不知道什么叫做魔克)



s3.play();

静态方法

```
public static int min(int a, int b) {
    // 返回a与b中较小的值
}
```



没有实例变量

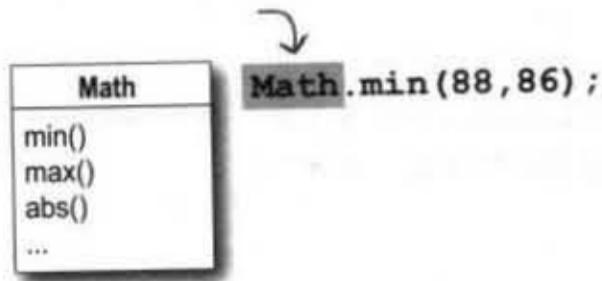
Math.min(42, 36);

直接用类的名字

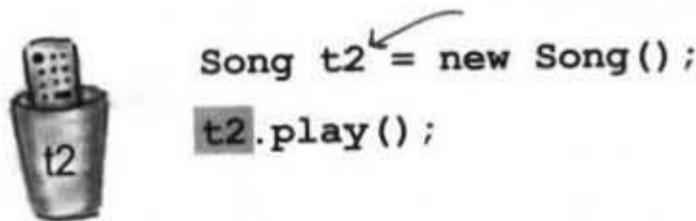


没有对象！绝对没有！

以类的名称调用静态的方法



以引用变量的名称调用非静态的方法



带有静态方法的含义

带有静态的方法的类通常（虽然不一定是这样）不打算要被初始化。在第8章我以已经讨论过抽象类，以及如何用abstract这个修饰字来标记类以让它不能被创建出实例。换句话说，抽象的类是不能被初始化的。

但你也可以用私有的构造函数来限制非抽象类被初始化。要记得，被标记为private的方法代表只能被同一类的程序所调用。构造函数也是同样意思，这也是Math如何防止被初始化的方法。它让构造函数标记为私有，所以你无法创建Math的实例。编译器会知道你不能存取这些私有的构造函数。

这并不是说有一个或多个静态的方法的类就不能被初始化。事实上，只要有main()的类都算有静态的方法！

通常你会写出main()来启动或测试其他的类。从main()中创建类的实例并调用新实例上的方法。

因此你可以任意地在类中组合静态与非静态的方法，然而任何非静态的方法都代表必须以某种实例来操作。取得新对象的方法只有通过new或者序列化(deserialization)以及我们不会讨论的Java Reflection API。除此之外，别无他法。实际上由谁来新建是一个很有意思的问题，稍后我们就会讨论这个部分。

静态的方法不能调用非静态的变量

静态的方法是在无关特定类的实例情况下执行的。如同你在上一页所看到的，甚至也不会有该类的实例出现。因为静态的方法是通过类的名称来调用，所以静态的方法无法引用到该类的任何实例变量。在此情况下，静态的方法也不会知道可以使用哪个实例变量值。

如果你尝试在静态的方法内使用实例变量，编译器会认为：“我不知道你说的是哪个实例的变量！”

静态的方法是不知道堆上有哪些实例的。

如果你要编译下面这段程序代码：

```
public class Duck {
    private int size;

    public static void main (String[] args) {
        System.out.println("Size of duck is " + size);
    }

    public void setSize(int s) {
        size = s;
    }

    public int getSize() {
        return size;
    }
}
```

哪一个Duck?



此时我们根本无法
得知堆上是否有
Duck

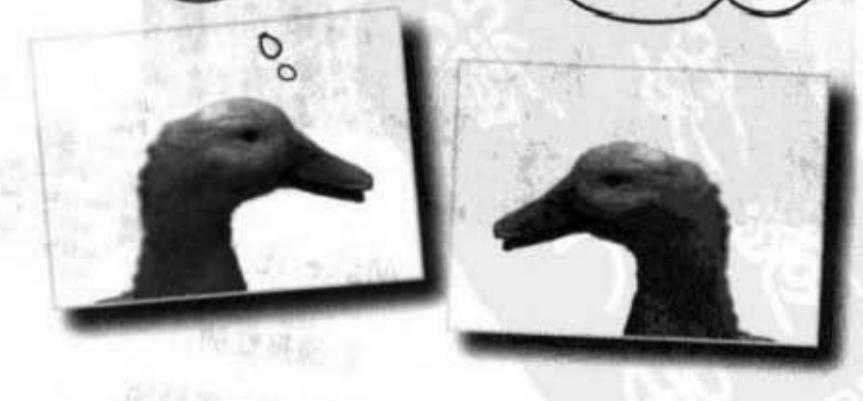
碧玉，我觉得
他们是在说我的
大小

不，述兰，他们
是在说我的大小

你会得到这样的错误：

```
File Edit Window Help Quack
% javac Duck.java
Duck.java:6: non-static variable
size cannot be referenced from a
static context
        System.out.println("Size
        of duck is " + size);
               ^

```



静态的方法也不能调用非静态的方法

非静态的方法做什么工作？它们通常是以实例变量的状态来影响该方法的行为。getName()方法会返回name变量的值。谁的名字？当然是被调用对象的。

这一段无法编译：

```
public class Duck {
    private int size;

    public static void main (String[] args) {
        System.out.println("Size is " + getSize());
    }

    public void setSize(int s) {
        size = s;
    }
    public int getSize() {
        return size;
    }
}
```

调用 getSize() 会需要用到 size 实例变量

老问题，到底谁的 size？

```
File Edit Window Help Jack-in
% javac Duck.java
Duck.java:6: non-static method
getSize() cannot be referenced
from a static context
    System.out.println("Size
of duck is " + getSize());
^
```



there are no
Dumb Questions

问： 如果从静态方法调用非静态方法，但此非静态方法没有用到实例变量，这样会通过编译吗？

答： 不会。编译器可以知道你有没有使用实例变量，你也知道。但如果现在可以通过，后来却把非静态变量改成会使用实例变量呢？又如果子类去覆盖这个方法成有用到实例变量的版本呢？

问： 我发誓曾经看过以引用变量代替类名称调用静态方法的程序代码，这样对吗？

答： 你确实可以这样做，但合法的事情并不一定都是好事。虽然可以使用类的实例来调用，但这样会产生容易误解的程序代码：

```
Duck d = new Duck();
String[] s = { };
d.main(s);
```

这段程序代码是合法的，但编译器还是会解析出原来的类。使用d来调用main()并不代表main会知道是哪个对象引用所做的调用。如此调用的方法也还是静态的！

静态变量： 它的值对所有的实例来说 都相同

假设你要在执行过程中计算有多少Duck的实例已经被建立出来。你要怎么做？或许可以在构造函数中递增某个实例变量的值？

```
class Duck {
    int duckCount = 0;
    public Duck() {
        duckCount++; 这会在创建Duck对象时
    } 执行递增
}
```

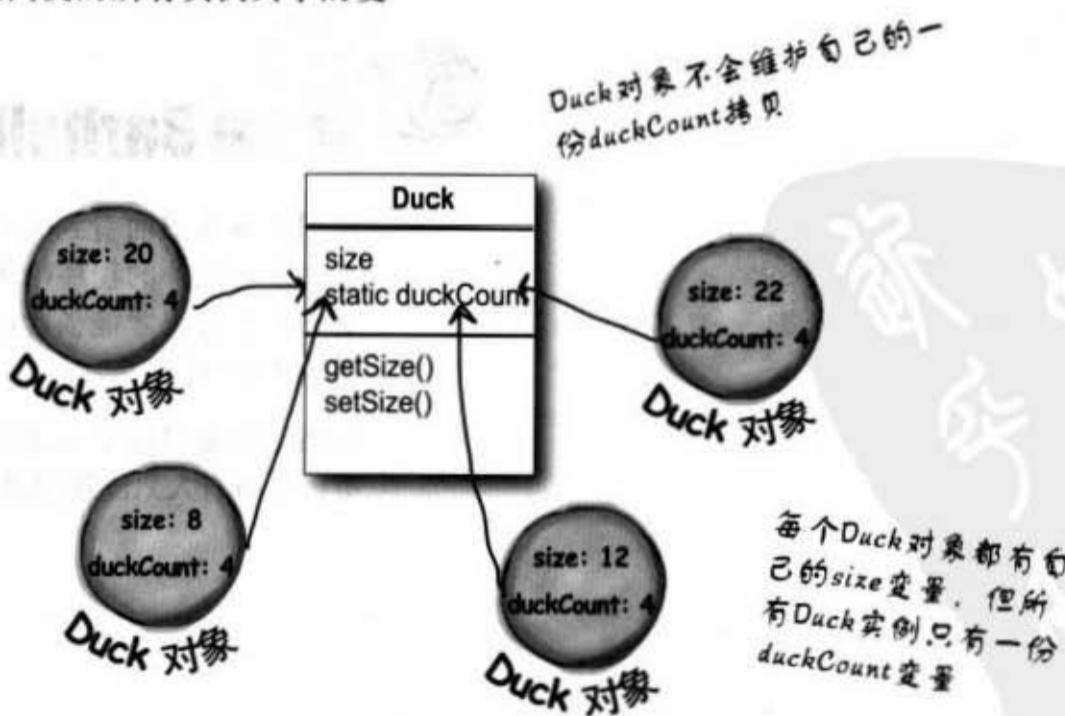
不行，因为duckCount是个实例变量，所以这样做不会成功。每个Duck在初始化的时候duckCount的值都是0。你也许可以调用别的类来计算，不过这样又不太优雅。你需要的是只会有一份拷贝的变量，且所有实例都会用到该拷贝。

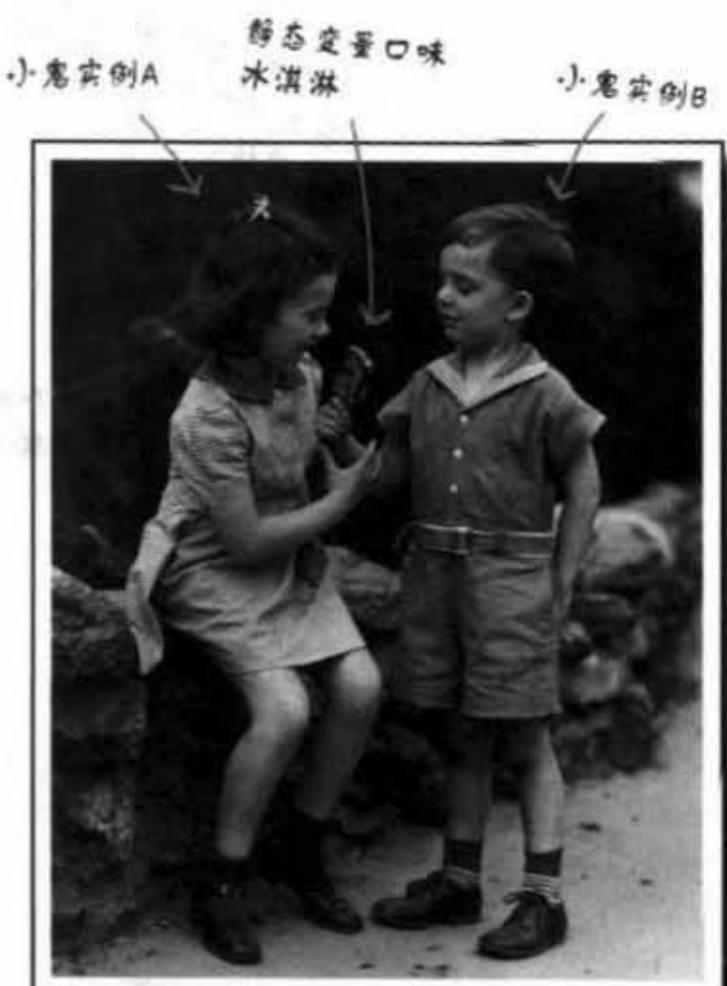
这就是静态变量的功用：被同类的所有实例共享的变量。

```
public class Duck {
    private int size;
    private static int duckCount = 0;
    public Duck() { 每当构造函数执行的时
        duckCount++; 候，此变量的值就会递
    }
}

public void setSize(int s) {
    size = s;
}
public int getSize() {
    return size;
}
```

此静态的duckCount变量只会在类第一次载入的时候被初始化。





静态变量是共享的。

同一类所有的实例共享
一份静态变量。

实例变量：每个实例一个。

静态变量：每个类一个。



Brain Barbell

我们在本章前面看到私有的构造函数代表这个类不能被本身以外的程序给实例化。也就是说，只有此类的程序代码才能创建出新的实例（到底是鸡生蛋还是蛋生鸡？）。

如果你想要以这个方法编写出只能创建一个实例的类，且所有人只能运用这一份实例，要该怎么做？

静态变量的起始动作

静态变量是在类被加载时初始化的。类会被加载是因为Java虚拟机认为它该被加载了。通常，Java虚拟机会加载某个类是因为第一次有人尝试要创建该类的新实例，或是使用该类的静态方法或变量。程序员其实也可以选择强制Java虚拟机去加载某个类，但你不太需要这么做。大部分的情况下还是让Java虚拟机来决定会比较好。

静态项目的初始化有两项保证：

静态变量会在该类的任何对象创建之前就完成初始化。

静态变量会在该类的任何静态方法执行之前就初始化。

class Player {
 static int playerCount = 0;
 private String name;
 public Player(String n) {
 name = n;
 playerCount++;
 }
}

public class PlayerTestDrive {
 public static void main(String[] args) {
 System.out.println(Player.playerCount);
 Player one = new Player("Tiger Woods");
 System.out.println(Player.playerCount);
 }
}

如果你没有给静态变量赋初值，它就会被设定默认值。int会被设定为0。静态变量的默认值会是该变量类型的默认值，就像实例变量所被赋予的默认值一样。

playerCount会在载入类的时候被初始化为0。

像是long或short等primitive主数据类型整数的默认值是0，primitive主数据类型的浮点数默认值是0.0，boolean是false，对象引用是null

静态变量也是通过类的名称来存取

File Edit Window Help What?
% java PlayerTestDrive
0 ← 在实例创建之前
1 ← 对象创建之后

静态的final变量是常数

一个被标记为final的变量代表它一旦被初始化之后就不会改动。也就是说类加载之后静态final变量就一直会维持原值。以Math.PI为例：

```
public static final double PI = 3.141592653589793;
```

此变量被标记为public，因此可供各方读取。

此变量被标记为static，所以你不需要Math的实例。

此变量被标记为final，因为圆周率是不变的。

此外没有别的方法可以识别变量为不变的常数（constant），但有命名惯例（naming convention）可以帮助你认出来。

常数变量的名称应该要都是大写字母！

静态初始化程序（static initializer）是一段在加载类时会执行的程序代码，它会在其他程序可以使用该类之前就执行，所以很适合放静态final变量的起始程序。

```
class Foo {
    final static int x;
    static {
        x = 42;
    }
}
```

静态final变量的初始化：

① 声明的时候：

```
public class Foo {
    public static final int FOO_X = 25;
}
```

注意这个命名惯例——
应该都是大写的，并以
下划线字符分隔

或

② 在静态初始化程序中：

```
public class Bar {
    public static final double BAR_SIGN;
    static {
        BAR_SIGN = (double) Math.random();
    }
}
```

这段程序会在类被加载时执行

如果你没有以这两种方式之一来给值的话：

```
public class Bar {
    public static final double BAR_SIGN;
}
```

没有初始化！

编译器会发现这个问题：

```
File Edit Window Help Jack-in
% javac Bar.java
Bar.java:1: variable BAR_SIGN
might not have been initialized
1 error
```

final 不只用在静态变量上……

你也可以用final关键字来修饰非静态的变量，这包括了实例变量、局部变量甚或是方法的参数。不管哪一种，这都代表它的值不能变动。但你也可以用final来防止方法的覆盖或创建子类。

非静态final变量

```
class Foof {
    final int size = 3; ← size将无法改变
    final int whuffle;

    Foof() {
        whuffle = 42; ← whuffle不能改变
    }

    void doStuff(final int x) {
        // 不能改变x
    }

    void doMore() {
        final int z = 7;
        // 不能改变z
    }
}
```

final 的 method

```
class Poof {
    final void calcWhuffle() {
        // 绝对不能被覆盖过
    }
}
```

final 的 class

```
final class MyMostPerfectClass {
    // 不能被继承过
}
```

final的变量代表你不能改变它的值。

final的method代表你不能覆盖掉该method。

final的类代表你不能继承该类（也就是创建它的子类）。

完了，一切都final了！做什么都没用了！



there are no
Dumb Questions

要点

问： 静态的方法不能存取非静态的变量，但非静态的方法可以读取静态的变量吗？

答： 当然可以。非静态方法不可以调用该类静态的方法或静态的变量。

问： 为何需要将类标记为final？这不会破坏面向对象的目的吗？

答： 会也不会。将类标记为final的主要目的是为了安全。例如String这个类，假使有人继承过，弄了一个行为很不一致的版本，就会对预期操作String的程序产生很多问题。

问： 如果类已经是final的，再标记final的方法是不是很多余？

答： 不只是多余，而且多了很多。如果一个类不能被子类化，则它的方法根本就无法被覆盖。如果只是想要限制部分的方法不能被覆盖过，那就单独地标记它们为final的就行。

- 静态的方法应该用类的名称来调用，而不是用对象引用变量。
- 静态的方法可以直接调用而不需要堆上的实例。
- 静态的方法是一个非常实用的方法，它不需要特别的实例变量值。
- 静态的方法不能存取非静态的方法。
- 如果类只有静态的方法，你可以将构造函数标记为private的以避免被初始化。
- 静态变量为该变量所属类的成员所共享。静态变量只会有一份，而不是每个实例都有自己的一份。
- 静态方法可以存取静态变量。
- 在Java中的常量是把变量同时标记为static和final的。
- final的静态变量值必须在声明或静态初始化程序中赋值：

```
static {  
    DOG_CODE = 420;  
}
```

- 常量的命名惯例是全部使用大写字母。
- final值一旦被赋值就不能更改。
- final的方法不能被覆盖。
- final的类不能被继承。



谁是合法的？

就你所学到的static和final知识来看，下面哪些程序可以通过编译？



- ```
public class Foo {
 static int x;

 public void go() {
 System.out.println(x);
 }
}
```
- ```
public class Foo2 {
    int x;

    public static void go() {
        System.out.println(x);
    }
}
```
- ```
public class Foo3 {
 final int x;

 public void go() {
 System.out.println(x);
 }
}
```
- ```
public class Foo4 {
    static final int x = 12;

    public void go() {
        System.out.println(x);
    }
}
```
- ```
public class Foo5 {
 static final int x = 12;

 public void go(final int x) {
 System.out.println(x);
 }
}
```
- ```
public class Foo6 {
    int x = 12;

    public static void go(final int x) {
        System.out.println(x);
    }
}
```

Math 的方法

现在我们已经知道 static 的方法是如何工作的，接着让我们来看一下 Math 的一些方法。这不是全部，API 文件还有像是 `sqrt()`、`tan()`、`ceil()` 等的说明。

Math.random()

返回介于 0.0 ~ 1.0 之间的双精度浮点数。

```
double r1 = Math.random();
int r2 = (int) (Math.random() * 5);
```

Math.abs()

返回双精度浮点数类型参数的绝对值。这个方法有覆盖的版本，因此传入整型会返回整型，传入双精度浮点数会返回双精度浮点数。

```
int x = Math.abs(-240); // 返回240
double d = Math.abs(240.45); // 返回240.45
```

Math.round()

根据参数是浮点型或双精度浮点数返回四舍五入之后的整型或长整型值。

```
int x = Math.round(-24.8f); // 返回-25
int y = Math.round(24.45f); // 返回24
```

↑ 文字直接表示的浮点数都会被当作双精度浮点数，除非后面有加上 f

Math.min()

返回两参数中较小的那个。这有 int、long、float 或 double 的覆盖版本。

```
int x = Math.min(24, 240); // 返回24
double y = Math.min(90876.5, 90876.49); // 返回90876.49
```

Math.max()

返回两参数中较大的那个。这有 int、long、float 或 double 的重载版本。

```
int x = Math.max(24, 240); // 返回240
double y = Math.max(90876.5, 90876.49); // 返回90876.5
```

primitive主数据类型的包装

有时你会想要把primitive主数据类型当作对象来处理。例如在5.0之前的Java版本上，你无法直接把primitive主数据类型放进ArrayList或HashMap中：

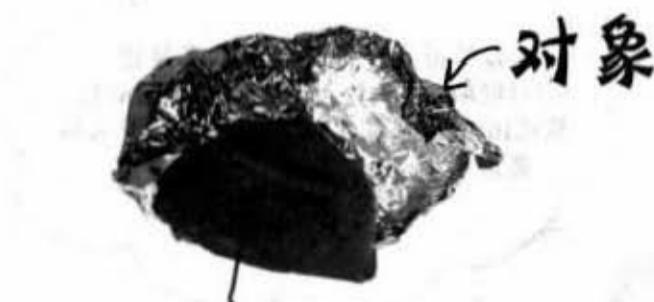
```
int x = 32;
ArrayList list = new ArrayList();
list.add(x);
```

除非是用Java 5.0或以上的版本，否则这个命令不会成功

每一个primitive主数据类型都有个包装用的类，且因为这些包装类都在java.lang这个包中，所以你无需去import它们。每个包装类都很好辨别，因为它的名称是照着所包装的类型所设定的，只是将第一个字母改为大写以符合命名惯例。

还有，为了某些没有人知道的理由，API的设计者决定让名称不是完全地符合primitive主数据类型的名字：

Boolean	
Character	←
Byte	
Short	← 注意！这个名称与primitive主数据类型不同，是完整拼出来的
Integer	←
Long	
Float	
Double	传入primitive主数据类型给包装类的构造函数
包装值	
int i = 288;	
Integer iWrap = new Integer(i);	
所有的包装类都有类似这样的方法	
int unWrapped = iWrap.intValue();	
解开包装	



primitive主数据类型

当你需要以对象方式来处理primitive主数据类型时，就把它包装起来。

Java 5.0之前的版本必须要这么做。



*上面的图听说是个叉烧包



这真的很蠢。你说我不能直接做
出int的ArrayList? 还得要把每个int包
装成Integer对象才行? 既浪费时间又容
易出错……

在Java 5.0以前你得这样做……

没错。在Java 5.0之前的版本上, primitive主数据类型就是原始类型, 而对象引用就是对象引用, 两者绝无交换使用的方法。要交互使用就得靠程序员进行包装与拆开包装的动作。没有办法能够以primitive主数据类型来直接传入期待对象引用的方法, 也没有办法能够把回传对象参考的method值赋值给primitive主数据变量。Integer与int两者间毫无关系可言, 只是Integer带有一个int类型的实例变量(以保存Integer所包装的primitive)。你得想办法自己进行这一类转换的工作。

primitive int的ArrayList

无autoboxing

```
public void doNumsOldWay() {
    ArrayList listOfNumbers = new ArrayList();
    listOfNumbers.add(new Integer(3));
    Integer one = (Integer) listOfNumbers.get(0);
    int intOne = one.intValue();
}
```

↑
最后再取出 primitive

创建出ArrayList对象
不能直接加入primitive的3, 得先
转换成Integer
返回Object类
型, 但你可以
将Object转换成
Integer

autoboxing：不必把primitive主数据类型与对象分得那么清楚

从5.0版开始加入的autoboxing功能能够自动地将primitive主数据类型转换成包装过的对象！

让我们看一下创建int的ArrayList时会发生什么事。

primitive int的ArrayList

有autoboxing

```
public void doNumsNewWay() {
    ArrayList<Integer> listOfNumbers = new ArrayList<Integer>();
    listOfNumbers.add(3);
    int num = listOfNumbers.get(0);
}
```

编译器也会自动地解开Integer对象的包装，因此可以直接赋值给int。

④ 为 Integer 类型的 ArrayList

虽然 ArrayList 没有 add(int) 这样的方法，但编译器会自动帮你包装。

问：为什么不直接声明 ArrayList<int>？

答：generic类型的规则是你只能指定类或接口类型。因此ArrayList<int>将无法通过编译。但你可以直接把该包装所对应的primitive主数据类型放进ArrayList中，比如说boolean类型的放入ArrayList<Boolean>中chars类型的放入ArrayList<Character>中。

静态的方法

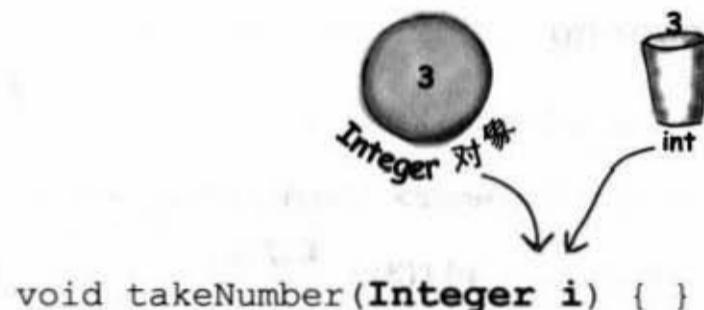
到处都用得到autoboxing

autoboxing不只是包装与解开primitive主数据类型给collection用而已，它还可以让你在各种地方交换地运用primitive主数据类型与它的包装类型。想想看有哪些地方会用到。

autoboxing乐趣多

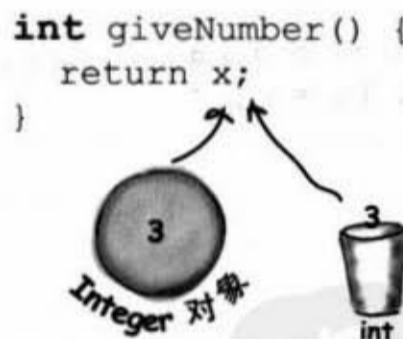
方法的参数

如果参数是某种包装类型，你可以传入相对应的primitive主数据类型，反之亦然。



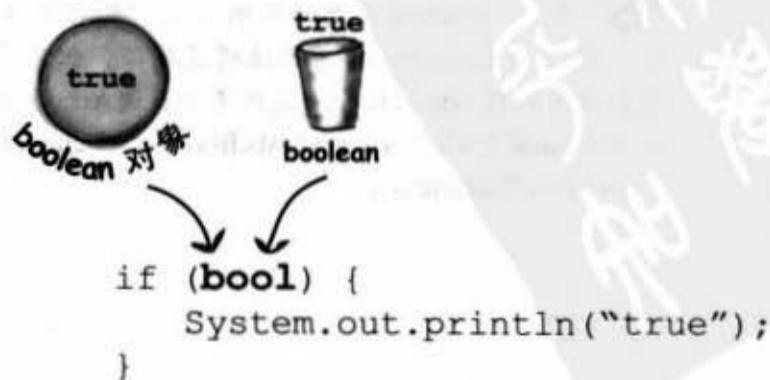
返回值

如果method声明为返回某种primitive主数据类型，你也可以返回兼容的primitive主数据类型或该primitive主数据类型的包装类型。



boolean 表达式

任何预期boolean值的位置都可以用求出boolean的表达式来代替，比如说`4>2`或是Boolean包装类型的引用。



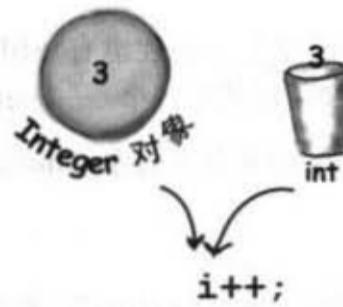
数值运算

这或许是最诡异的。你可以在使用 primitive 主数据类型作为运算子的操作中以包装类型来替换。这代表你可以对 Integer 的对象作递增运算！

```
Integer i = new Integer(42);
i++;
```

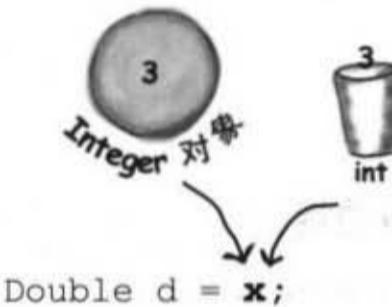
还可以这么做：

```
Integer j = new Integer(5);
Integer k = j + 3;
```



赋值

你可以将包装类型或 primitive 主数据类型赋给声明成相对应的包装或 primitive 主数据类型。



Sharpen your pencil

```
public class TestBox {
    Integer i;
    int j;

    public static void main (String[] args) {
        TestBox t = new TestBox();
        t.go();
    }

    public void go() {
        j=i;
        System.out.println(j);
        System.out.println(i);
    }
}
```

右边的程序代码能否通过编译？可以执行吗？如果可以，会有什么结果？

慢慢来，你有足够的时间去想，这会引起我们没讨论过的autoboxing问题。

你得要编译才会有答案，所以我们是在逼你上机操作。来吧！

包装类型的方法

等一下！还有呢！包装也有静态的实用性方法！

除了一般类的操作外，包装也有一组实用的静态方法。

我们在之前已经用过其中一个——Integer.parseInt()。

这个方法取用String并返回给你primitive主数据类型值。

将String转换成primitive主数据

类型值是很容易的：

```
String s = "2";
int x = Integer.parseInt(s);
double d = Double.parseDouble("420.24");

boolean b = new Boolean("true").booleanValue();
```

将“2”解析成2

你可能会以为有Boolean.parseBoolean()吧？其实没有。但是Boolean的构造函数可以取用String来创建对象

但若你这么做的话：

```
String t = "two";
int y = Integer.parseInt(t);
```

啊！可以通过编译，但执行时就会出状况

就会在运行期间遇到异常：

```
File Edit Window Help Clue
% java Wrappers
Exception in thread "main"
java.lang.NumberFormatException: two
at java.lang.Integer.parseInt(Integer.java:409)
at java.lang.Integer.parseInt(Integer.java:458)
at Wrappers.main(Wrappers.java:9)
```

解析String的方法或构造函数会抛出NumberFormatException异常。这是运行期间的异常，你应该会处理这种异常。

(下一章会讨论异常)

反过来将 primitive 主数据类型 值转换成 String

有好几种方法可以将数值转换成 String。最简单的方法是将数字接上现有的 String。

```
double d = 42.5;
String doubleString = "" + d;  " + " 这个操作数是 Java 中唯一有  
重载过的运算符
```

```
double d = 42.5;
String doubleString = Double.toString(d);
```

Double 这个类的静态方法

如果我要以某种特定的格式
列出数字怎么办？比如说
\$56.87 这样……

有没有像 C 语言中
的 printf 这种格式化功
能呢？



数字的格式化

在Java中，数字与日期的格式化功能并没有结合在输出/输入功能上。通常对用户显示数字是通过GUI来进行的。你会把String放在可滚动的文字区域块或表格中。如果格式化功能只有绑在文字模式输出的命令上，那就没有办法把字符串以比较漂亮的格式输出到GUI上。在Java 5.0之前的格式化功能是通过java.text这个包来处理，但本书已经不屑去提它了。

从Java 5.0起，更多更好更有扩展性的功能是通过java.util中的Formatter这个类来提供的。但你无需自己创建与调用这个class上的方法，因为Java 5.0已经把便利性的功能加到部分的输出/输入类与String上。因此只要调用静态的String.format()并传入值与格式设定就好。

当然，你还是得知道如何提供格式设定，本章会有一些基本的说明，我们会从基本的范例开始，并观察它们是如何运行的（在讨论输出/输入的章节中还会再看过一次）。

将数字以带逗号的形式格式化

```
public class TestFormats {
    public static void main (String[] args) {
        String s = String.format("%, d", 1000000000);
        System.out.println(s);
    }
}
```

1,000,000,000

↑
有逗号的数字格式

要格式化的数字

格式设定，用来指示应该用哪种形式来输出：这里的逗号是表示数字要以逗号来分隔，并不是说这里有%与d两项参数，千万别弄错了！

解析格式化结构.....

基本上来说，格式化由两个主要部分组成（不只是这样，但我们可以先从简单开始）：

● 格式指令

描述要输出的特殊格式。

要格式化的值

不是所有东西都能被格式化。例如，如果你的格式指令适用于浮点数，则你就不能传入 Dog 或看起来很像浮点数的 String。

如果你本来就已经熟悉C/C++的printf()，那这几页说的全部都是废话！

以这种格式…… 来表现这个值

```
format("%, d", 1000000000);
```



用这个格式将这个参数格式化

这个指令代表什么？

将此方法的第二个参数以第一个参数所表示带有逗号的整数 (decimal) 方式表示。

它会怎么做？

下一页会对“%， d”作更详细的说明，我们在这里先简略地说明一下。在格式化指令中的%代表一项变量，此变量就是跟在格式化指令后面的参数，其余的字符各有所代表的意义。

format()方法

%符号代表把参数放在这里

format()方法的第一个参数被称为“格式化串”，它可以带有实际上就是要这么输出而不用转译的字符。当你看到%符号时，要把它想做是会被方法其余参数替换掉的位置。

这会原汁原味地输出

参数，也就是后面那个数字

这也会原汁原味地输出

要被处理的值

format("I have %.2f bugs to fix.", 476578.09876);

输出 I have 476578.10 bugs to fix.

注意：数值的精确度有损失！你看出来.2f的意义吗？

This diagram shows the code `format("I have %.2f bugs to fix.", 476578.09876);`. Annotations explain the format string: "%." is labeled '参数，也就是后面那个数字' (parameter, which is the number after), '%.2f' is labeled '要被处理的值' (value to be processed), and 'I have' and 'bugs to fix.' are labeled '这会原汁原味地输出' (will be output as is). The output is shown as a box: 'I have 476578.10 bugs to fix.'. A note at the bottom says: '注意：数值的精确度有损失！你看出来.2f的意义吗？' (Note: The precision of the number is lost! Can you figure out the meaning of .2f?).

此例的%符号是第二个参数会放置的位置，“.2f”代表该参数要使用的格式，其他的字都会以原来的方式输出。

加上逗号

```
format("I have %,.2f bugs to fix.", 476578.09876);
```

I have 476,578.10 bugs to fix.

↑
%符号后面的指令被加上逗号之后，
输出也有了变化



格式化语句有自己的一套语法

很明显，%符号后面不可以随便填上任意的字符。%的语法有非常特殊的规则，是用来描述此处所用的格式。

你已经看过两个例子：

`%,d`：这代表以十进制整数带有逗号的方式来表示。

`%.2f`：这代表以小数点后两位的方式来格式化此浮点数。

`%,.2f`：代表整数部分以有逗号的形式表示，小数部分以两位来格式化。

所以问题在于你怎么知道什么字符代表什么意义，以及这些指令字符的使用顺序和时机。

想想看下面这个语句会做出什么样的输出？答案在下一页：

```
String.format("I have %.2f, bugs to fix.", 476578.09876);
```

“格式化说明”的格式

跟在百分号后而包括类型指示（像是d或f）的每个东西都是格式化指令。除非遇到新的百分号，在类型指示之后的一组字符，格式化程序会假设都是直接输出的字符串。这可能吗？要被格式化的参数可以超过一个以上吗？先别管这个，稍后再讨论。现在先来看格式化说明的语法——跟在%后面的那些指令。

格式化说明最多会有5个部分（不包括%符号）。下面的[]符号里面都是选择性的项目，因此只有%与type是必要的。格式化说明的顺序是有规定的，必须要以这个顺序来指定。

%[argument number] [flags] [width] [.precision] **type**

如果要格式化的参数超过一个以上，可以在这些里指定是哪一个，我们稍后会讨论这部分

指定类型的特定选项，例如数字要加逗号或正负号

最小的字符数，指出可以超过此宽度，若不足则会主动补零

精度，注意前面这不是总数，输出可以超过此宽度，若不足则会主动补零

一定要指定的类型标识

%[argument number] [flags] [width] [.precision] **type**

format("%,6.1f", 42.000);

除了没有argument number之外，其他的项目都用到

唯一的必填项目是类型

虽然类型是唯一必填的项目，但若指定别的项目，则类型必须是最后一项！类型修饰符有十几种（这不包括日期和时间的，它们有自己的一组），但大部分时间你会使用到%d或%f。且通常你会对%f加上精确度指示来设定所需要的小数长度。

The TYPE is mandatory, everything else is optional.

%d decimal
format("%d", 42);
42

如果给的是42.45就不行

参数必须能够与 int 相容。

%f floating point
format("%.3f", 42.000000);
42.000

3配上会强制输出3位的小数

参数必须是浮点数类型。

%x hexadecimal
format("%x", 42);

2a

参数必须是byte、short、int、long、BigInteger。

%c character
format("%c", 42);
* ASCII的42代表“*”号

参数同上，但不包括BigInteger。

在格式化指令中一定要给的类型，如果还要指定其他项目的话，要把类型摆在最后。

超过一项以上的参数时呢？

如果你要输出像下面这样的字符串：

“The rank is 20,456,654 out of 100,567,890.24.”

但这两个数字来自于不同的变量，该怎么做？把新的参数加到后面，因此你会以3个参数来调用format()而不是两个。并且在第一个参数中，也就是格式化串中，会有两个不同的格式化设定，也就是两个%开头的字符组合。第二个参数会应用在第一个%号上面，第三个参数会用在第二组%组合上。也就是说参数会依照顺序应用在%上面。

```
int one = 20456654;
double two = 100567890.248907;
String s = String.format("The rank is %,d out of %,.2f", one, two);
```

The rank is 20,456,654 out of 100,567,890.25

一项以上的参数会依序
对应到格式化设定

两项都有加逗号，且后者的
小数被限制在两位

当我们讨论到日期的格式化时，你就会看到以同一个参数应用在不同的格式化设定上。这可能有点难以理解，除非你直接看到日期格式化的例子。接下来我们就会讨论如何明确地指定哪个格式化设定要用在哪个参数上。

问：有些地方真的怪怪的，到底可以传多少个参数进去？我是说format()到底有多少个重载的版本？如果有10个参数要用在格式化上面会怎样？

答：问得好！没错，这里有些不一样的东西，且实际上是没有一大堆的重载版format()来取用不同数目排列组合的参数。为了要应付格式化的API，Java语言需要一种新的功能——称为可变参数列表（variable argument list，简称为vararg）。这个部分在附录中有说明，通常你很少会用到这样的功能。

都与数字有关，那日期呢？

如果你要输出这样的字符串：Sunday, Nov 28 2004

看起来好像没什么特别。但是如果说是要把Date类型的变量日期用这样的格式输出呢？Date类型是Java上表示时间用的，而现在你得处理这个类型。

数值与日期时间格式化的主要差别在于日期格式的类型是用“t”开头的两个字符来表示，下面有几个范例：

完整的日期与时间：%tc

```
String.format("%tc", new Date());
Sun Nov 28 14:52:41 MST 2004
```

只有时间：%tr

```
String.format("%tr", new Date());
03:01:47 PM
```

周、月、日：%tA %tB %td

因为没有刚好符合我们要求的输出格式，所以得组合3种形式来产生出所需要的格式：

```
Date today = new Date();
String.format("%tA, %tB %td", today, today, today);
```

这样就得把Date对象传
进去3次

这里的逗号是直接输出的

```
Sunday, November 28
```

同上，但不用重复给参数

```
Date today = new Date();
String.format("%tA, %<tB %<td", today);
```

“<”这个符号是个特殊的指示，用来告诉格式化程序重复利用之前用过的参数



上上个月是18号来的，上
个月是15号来的……糟了，
那这个月不就……

操作日期

除了取得今天的日期之外，你还会遇到很多日期上的操作。你会需要调整日期、计算花费时间、排定优先级、找出下一周期的开始时间等。所以你会要用到日期的高级操作功能。

你可以自己编写日期程序（别忘记处理闰年）。嗯，这实际做起来还蛮复杂的。所以最好还是使用Java API中已经写好、功能丰富的类来帮你处理日期吧。但是有时你会发现这些类的功能也太丰富了吧……

在时光中前后移动

假如说中国厨艺学院的课程表是从周一到周五。你被十八铜人指定要查出今年每个月最后的上课日……

看来java.util.Date没什么用处

之前我们使用java.util.Date来查询今天的日期，因此从这个类开始寻找适用的功能是很合理的，但当你检查API文件时，你会发现有很多Date的功能被停用了！

但这个类还是很适合用来取得目前的时间。

好消息是API建议到java.util.Calendar上面去寻找其余的功能，因此我们要说：

就用java.util.Calendar来操作日期吧！

Calendar这个API的设计者打算要做全球化的思考。基本的想法是当你要操作日期时，你会要求一个Calendar（通过下一页会讨论的一个静态方法），然后Java虚拟机会赏你一个Calendar的子类实例（实际上Calendar是个抽象的类，所以你能用到的是它的具体子类）。

有意思的是，你所取得的Calendar是符合所在地区（locale）特性的。通常大部分的地区适用公历，但你也有可能处于使用农历或其他特殊格式的情况，此时你可以让Java函数库来处理这一类的日期。

标准的Java API带有java.util.GregorianCalendar，因此我们在书上使用这个历法。通常你也不需要担心你是在使用哪一种Calendar的sub-class，只要专注于Calendar的方法就可以了。

要取得当前的日期时间就用Date，其余功能可以从Calendar上面找。

上一页的老兄是在看信用卡的账单日期……

取得继承过Calendar的对象

你要如何取得抽象类的“实例”呢？当然不行，下面这个例子根本无法运行。

这个不行：

```
Calendar cal = new Calendar();
```

要用这个静态的方法：

```
Calendar cal = Calendar.getInstance();
```

无法通过编译！

这个语法看起来很熟悉——是个对静态方法的调用

等一下！如果不能创建
Calendar的实例，那Calendar
的引用到底引用了谁？



你无法取得Calendar的实例，但是你可以取得它的具体子类的实例

很明显，你不能取得Calendar的实例，因为Calendar是抽象的。但你还是能够不受限制地调用Calendar的静态method，这是因为静态的方法是在类上而不是在某个特定的实例上。所以你对Calendar调用getInstance()会返回给你具体子类的实例。那是某种继承过Calendar（也就是Calendar的多态变化版本）并且会依据合约来响应Calendar应有的方法。

大部分的Java版本都会默认返回一个java.util.GregorianCalendar的实例。

运用Calendar对象

要运用Calendar对象你得先了解几个关键的概念：

- 字段会保存状态——Calendar对象使用许多字段来表示某些事物的最终状态，也就是日期和时间。比如说你可以读取和设定它的year或month字段。
- 日期和时间可以运算——Calendar的方法能够让你对不同的字段作加法或减法的运算，比如说对month字段加一个月或对year减去三年。
- 日期与时间可以用millisecond来表示——Calendar可以让你将日期转换成微秒的表示法，或将微秒转换成日期。（更精确的说法是相对于1970年1月1日的微秒数），因此你可以执行精确的相对时间计算。

运用Calendar对象的范例：

```

Calendar c = Calendar.getInstance();
c.set(2004,1,7,15,40);           ← 将时间设定为2004年1月7日
long day1 = c.getTimeInMillis();  ← 15:40，注意月份是零基的
day1 += 1000 * 60 * 60;
c.setTimeInMillis(day1);         ← 将目前时间转换为以
                                 millisecond表示
                                 ← 将c的时间加上一个半小时
System.out.println("new hour " + c.get(c.HOUR_OF_DAY));
c.add(c.DATE, 35);              ← 加上35天，所以c已经
                                 到了2月
System.out.println("add 35 days " + c.getTime());
c.roll(c.DATE, 35);             ← 滚动35天，注意：只有日期字
                                 段会动，月份不会动
System.out.println("roll 35 days " + c.getTime());
c.set(c.DATE, 1);               ← 直接设定DATE的值
System.out.println("set to 1 " + c.getTime());

```

```

File Edit Window Help Time-Files
new hour 16
add 35 days Wed Feb 11 16:40:41 MST 2004
roll 35 days Tue Feb 17 16:40:41 MST 2004
set to 1 Sun Feb 01 16:40:41 MST 2004

```

输出结果

Calendar API的精华

这里只有列出Calendar部分的字段和方法。它是相当大的API，因此这里只能列出常用的部分，一旦你熟悉它的操作之后，其余的部分也会很容易使用。

重要的方法

add(int field, int amount)

加减时间值

get(int field)

取出指定字段的值

getInstance()

返回Calendar，可指定地区

getTimeInMillis()

以毫秒返回时间

roll(int field, boolean up)

加减时间值，不进位

set(int field, int value)

设定指定字段的值

set(year, month, day, hour, minute)

设定完整的时间

setTimeInMillis(long millis)

以毫秒指定时间

// 不只这些……

关键字段

DATE / DAY_OF_MONTH

每月的几号

HOUR / HOUR_OF_DAY

小时

MILLISECOND

毫秒

MINUTE

分钟

MONTH

月份

YEAR

年份

ZONE_OFFSET

时区位移

// 还有更多……

静到最高点！静态的import

这是Java 5.0的新功能：一把双刃剑。有些人很喜欢这个主意，有些人恨死它了。如果你讨厌多打几个字，那你会喜欢这项功能。但它的缺点是会让程序比较难阅读。

基本上，这功能是让你import静态的类、变量或enum（稍后介绍）时能够少打几个字。

警告：这会让你的
程序更易混淆

旧式的写法：

```
import java.lang.Math;
class NoStatic {
    public static void main(String [] args) {
        System.out.println("sqrt" + Math.sqrt(2.0));
        System.out.println("tan" + Math.tan(60));
    }
}
```

静态import的语法

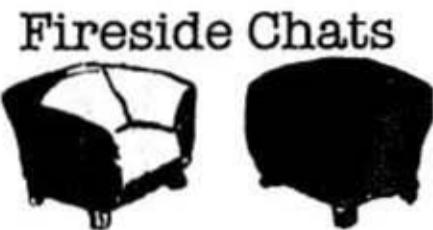
使用static import的写法：

```
import static java.lang.System.out;
import static java.lang.Math.*;
class WithStatic {
    public static void main(String [] args) {
        out.println("sqrt" + sqrt(2.0));
        out.println("tan" + tan(60));
    }
}
```

只不过是少打一些字

一 时机与要领

- 如果只会用到一两次，不如不用静态的import，这样程序会比较好阅读。
- 如果会用到很多次，或许用static的import会让程序看起来比较清爽。
- 在静态import的声明中也可以使用.*这样的通用字符。
- 最重要的问题是很容易产生名称的冲突。例如add()到底是要调用哪个的方法？



今晚的话题： 实例变量对静态变量的卑劣指控

实例变量

我不明白为什么要谈这些事情。每个人都知道静态变量只是用来当常数而已。又没有多少常数要用，我猜整个API大约只有四五个吧？真的用过的人也没几个吧。

对啦，全部都是。不过每个人都知道Swing只是个特殊的例子。

是呀，除了少数几个GUI之外，你还能举出任何一个真的大家都在用的静态变量吗？

呃……这也算特殊的例子呀。何况大家只是用它来除错而已。

静态变量

老兄，没有常识也应该要多看电视。你一定很少看API吧？里面有一堆的静态变量呢。甚至有一整个类是专门用来放常数值的。比如说 SwingConstants就全部都是。

或许它是个特例，但也是个很重要的特例啊！还有Color这个类，如果你记住所有颜色的RGB值那会有多恐怖？所以它已经定义好了红橙黄绿等颜色，用起来很方便的。

例如System.out，它是System这个类的一个静态变量。你不必自己创建出System的实例，只要从该class调用out变量就行。

怎样？除错不重要吗？你的豆腐脑一定不知道静态变量比较有效率，共享的类会省下很多内存的！

实例变量

嗯……你该不会忘记了吧？

静态变量不是面向对象的！我们干脆回到古代用算盘来执行计算的工作好了。

就像全局变量一样，许多潇洒帅气的程序员都知道那通常是很负面的事情。

这样就不叫面向对象程序设计了，这叫笨蛋。
你真的是老古董啊。

对啦，偶尔用一下静态变量还算合理，但是我要告诉你，滥用静态变量和方法就是不成熟面向对象程序员的招牌。程序员应该想的是对象的状态而不是类的状态。

这代表程序员还在用程序化的思维想事情，而不是依据对象的状态来进行处理。

是是是，就你自己需要嘛！

静态变量

什么？

别乱扣帽子，不是面向对象怎么了？那是什么？

对不起，我并不是全局变量。根本没有全局变量，我是在面向对象化的类中的！我是对象的自然状态，唯一的差别是我被很有效率地共享。

够了！闭嘴！这不是真的。某些静态变量对系统是非常关键的，不然至少也能够提供便利性。

你说什么？这又跟静态的方法扯上什么关系？

当然，我同意面向对象设计应该要专注于对象，有些问题只是因为菜鸟的因素……当你真的有需要的时候，没有东西能够代替静态。



```

class StaticSuper{
    static {
        System.out.println("super static block");
    }
}

StaticSuper{
    System.out.println(
        "super constructor");
}

public class StaticTests extends StaticSuper {
    static int rand;

    static {
        rand = (int) (Math.random() * 6);
        System.out.println("static block " + rand);
    }

    StaticTests() {
        System.out.println("constructor");
    }

    public static void main(String [] args) {
        System.out.println("in main");
        StaticTests st = new StaticTests();
    }
}

```

我是编译器

这一页的Java程序代码代表一份完整的程序。你的任务是要扮演编译器角色并判断程序输出会是哪一个？



哪个才是它的输出？

可能输出：

```

File Edit Window Help Cling
%java StaticTests
static block 4
in main
super static block
super constructor
constructor

```

可能输出：

```

File Edit Window Help Electricity
%java StaticTests
super static block
static block 3
in main
super constructor
constructor

```



这一章讨论 Java 的静态世界。你的任务是辨别下面的陈述哪些是对的、哪些是错的？

是非题

- (1) 使用Math类的第一个步骤是创建出它的实例。
- (2) 构造函数可以标记为静态的。
- (3) 静态的方法不能存取“this”所引用的对象。
- (4) 最好通过引用变量来调用静态的方法。
- (5) 静态变量可以用来计算类的实例数量。
- (6) 构造函数是在静态变量的初始化之前执行的。
- (7) MAX_SIZE是个合法的静态final变量名称。
- (8) 静态初始化程序会在构造函数之前执行。
- (9) 如果类被标记为final，则它的方法也必须标记为final。
- (10) final的方法只能在它的类被继承时覆盖。
- (11) boolean类型没有包装用的类。
- (12) wrapper是用来把primitive主数据类型包装成对象。
- (13) parseXxx方法都会返回String。
- (14) 格式化的类都在java.format中包。



排排看

下面是被打散的Java程序片段，程序的目的是要以29.52的周期计算满月：上一次的满月是在2004年1月7日。你是否能够将它们重新排列以成为可以编译并执行并产生如同下方的输出结果？注意到有些括号已经遗失，所以你可以在认为有需要时自行补上。

```

long day1 = c.getTimeInMillis();
c.set(2004,1,7,15,40);

import static java.lang.System.out;
static int DAY_IM = 60 * 60 * 24;
("full moon on %tc", c));
(c.format
Calendar c = new Calendar();
class FullMoons {

public static void main(String [] args) {
    day1 += (DAY_IM * 29.52);

    for (int x = 0; x < 60; x++) {
        static int DAY_IM = 1000 * 60 * 60 * 24;
        println
        import java.io.*;
        ("full moon on %t", c));
        import java.util.*;

        static import java.lang.System.out;
        c.set(2004,0,7,15,40);
        out.println
        c.setTimeInMillis(day1);
        (String.format
Calendar c = Calendar.getInstance();

```

```

File Edit Window Help Howl
% java FullMoons
full moon on Fri Feb 06 04:09:35 MST 2004
full moon on Sat Mar 06 16:38:23 MST 2004
full moon on Mon Apr 05 06:07:11 MDT 2004

```

练习解答

是非题

我是编译器

```
StaticSuper() {
    System.out.println(
        "super constructor");
}
```

构造函数必须要有括号。

可能输出

```
File Edit Window Help Cling
%java StaticTests
super static block
static block 3
in main
super constructor
constructor
```

- (1) 错
- (2) 错
- (3) 对
- (4) 错
- (5) 对
- (6) 错
- (7) 对
- (8) 对
- (9) 错
- (10) 错
- (11) 错
- (12) 对
- (13) 错
- (14) 错

练习解答



```
import java.util.*;  
  
import static java.lang.System.out;  
  
class FullMoons {  
  
    static int DAY_IM = 1000 * 60 * 60 * 24;  
  
    public static void main(String [] args) {  
  
        Calendar c = Calendar.getInstance();  
  
        c.set(2004,0,15,40);  
  
        long day1 = c.getTimeInMillis();  
  
        for (int x = 0; x < 60; x++) {  
  
            day1 += (DAY_IM * 29.52)  
  
            c.setTimeInMillis(day1);  
  
            out.println(String.format("full moon on %tc", c));  
        }  
    }  
}
```

注意：这个解决方案有些取巧的做法，如果要做到完美就又太过复杂了。

```
File Edit Window Help Howl  
* java FullMoons  
full moon on Fri Feb 06 04:09:35 MST 2004  
full moon on Sat Mar 06 16:38:23 MST 2004  
full moon on Mon Apr 05 06:07:11 MDT 2004
```

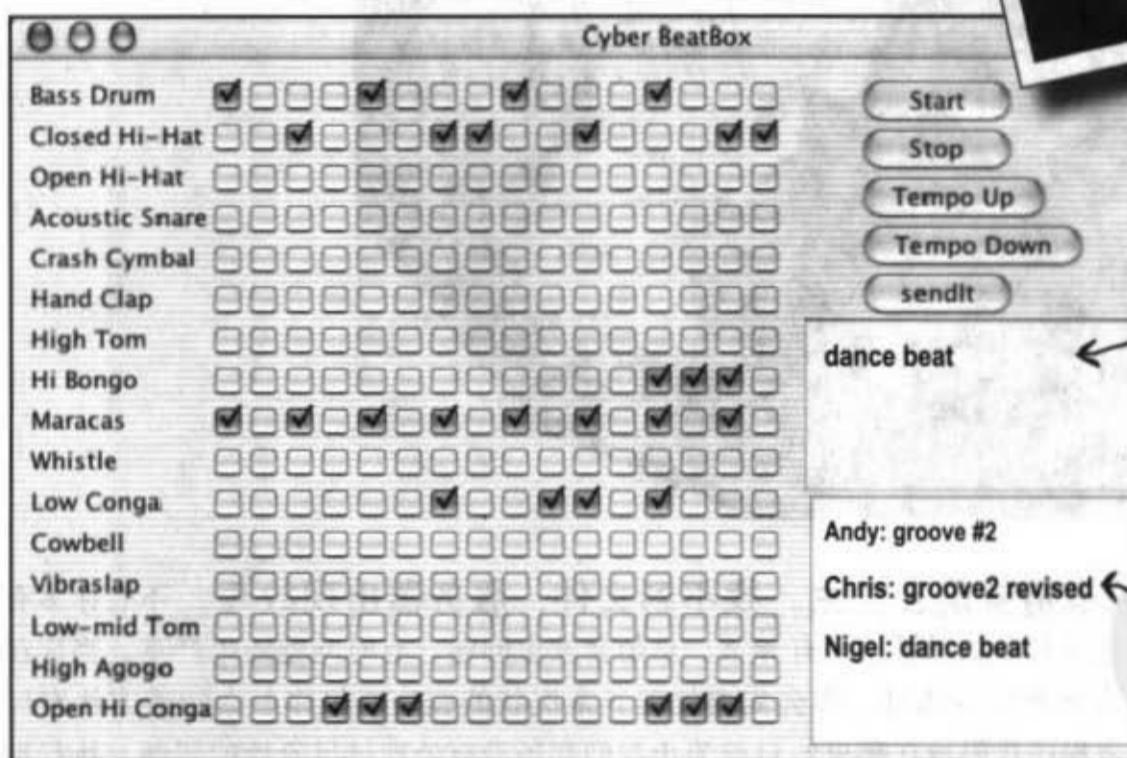


倒霉的事情就是会发生。找不到文件、服务器出现故障。不管你多有天份，你也没有办法保证不会有异常。总是会发生问题，有时问题还很严重。当你在编写可能有异常的方法时，你会需要处理异常状况的程序。但你怎么知道方法有风险呢，异常程序代码放在哪里？目前为止我们都还没有处理过风险性的问题。执行期肯定会有问题，但大多数都来自于程序的错误，而这些错误应该在开发阶段解决掉。不过我们所说的问题是你无法保证在执行期不会出现的。例如预期某些文件会正确地待在某个特定的目录中，但实际执行时文件却又失踪了。这一章我们会使用具有风险的JavaSound API来创建一个MIDI音乐播放程序。

创建MIDI音乐播放器

我们会在接下来的三章创建一个不太一样的声效应用程序，其中包括了BeatBox Drum播放机。在本书结束之前，我们会有个可以把鼓声送给另外一个播放器的版本，这有点类似讨论版。你可以全部自己动手写，也可以下载已经写好的GUI部分。虽然这个程序没什么商业价值，但是可以通过这个过程学习Java。这样的过程只是学习Java的一种比较有趣的方式。

对checkbox打勾来设定是否要发出声音



你可以在上面的16个拍子上打勾。例如在第一拍Bass Drum与Maracas会发出声音，而第二拍不会有声音，Maracas与Closed Hi-Hat会在第三拍发声……这样懂了吧？按下Start会持续循环地演奏，按Stop就会停下来。任何时候你都可以送出目前的组合到BeatBox服务器上，让其他人也能够聆听你的经典大作。你也可以点选别人送来的信息以加载节拍组合。

从基本谈起

很明显的，我们得要先学习某些事情才能完成这个程序。这包括了如何创建Swing GUI、如何通过网络连接到其他计算机，以及让我们可以把数据送出的输入/输出设备。

当然还有JavaSound这个API。这一章会先谈这个部分，你可以先暂时忘掉GUI与网络联机等部分，现在只要专注于由MIDI来发出声音就好。这一章会说明MIDI以及如何读取与播放声音。你是否已经开始想象上台领金曲奖的模样？

JavaSound API

这是在Java 1.3之后所加入的一组类和接口。它们是放在J2SE 的类函数库中。JavaSound被分为两个部分：MIDI和取样（sampled）。这本书只会讨论MIDI，它代表Musical Instrument Digital Interface，也是不同电子发声装置沟通的标准协议。但在我们的程序中，你可以把MIDI想象成某种乐谱，它可以输入到某种“高级多功能电子魔音琴”中。换句话说，MIDI本身不带有声音，它带的是有MIDI播放功能装置的指令。

MIDI数据表示执行的动作（播放中央C，以及音量大小和长度等），但没有实际的声音。它并不知道怎样产生钢琴、吉它、鼓的音效，实际的声音是靠装置发出的。这样的装置很像是个编制完整的交响乐团，它可能是个高级键盘乐器，也有可能是计算机中的纯软件音源。

对BeatBox来说，只会使用内建、纯软件的乐器音效。这通常被称为synthesizer（有些人把它叫做software synth）。



看起来很简单

首先我们需要一个Sequencer

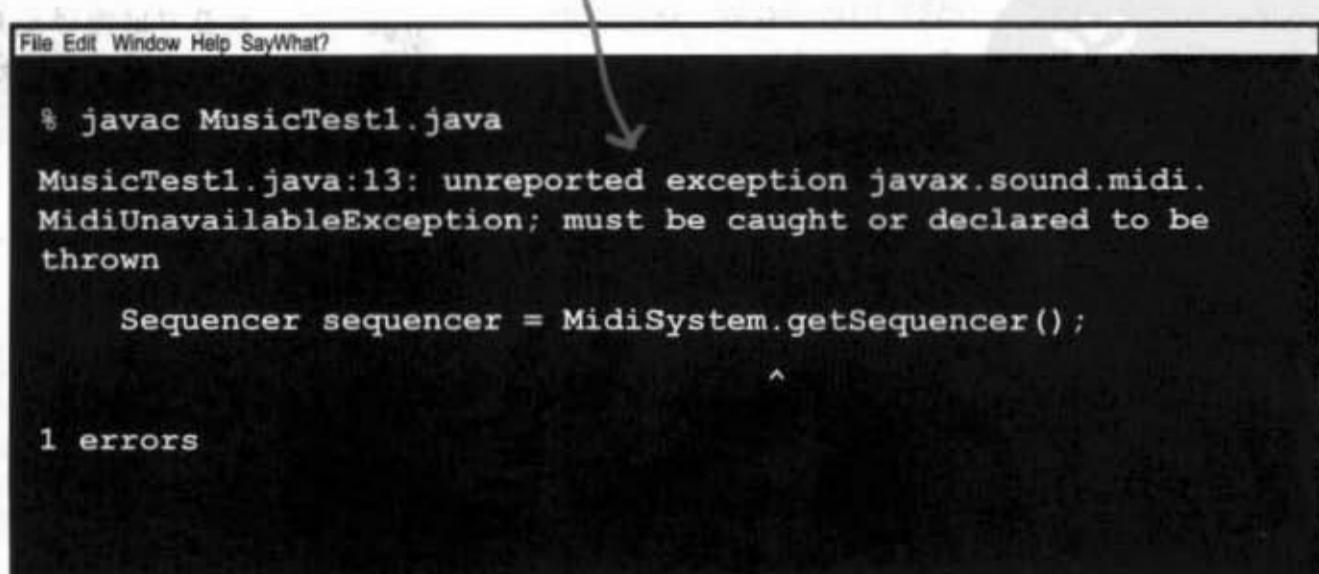
在我们要能够发出任何声音之前，必须先要取得Sequencer对象。此对象会将所有的MIDI数据送到正确的装置上，由装置来产生音乐。sequencer实际上可以做很多事情，但在我们的章节中所指的是播放的装置，就好像是你家中音响的CD唱盘。Sequencer这个类位于javax.sound.midi这个包中（Java 1.3版之后的标准Java函数库中），所以我们先从确认可以取得Sequencer对象开始。

```
import javax.sound.midi.*; ← 引用该包
public class MusicTest1 {
    public void play() {
        Sequencer sequencer = MidiSystem.getSequencer();
        System.out.println("We got a sequencer") ↗
    } // 关闭播放
    public static void main(String[] args) {
        MusicTest1 mt = new MusicTest1();
        mt.play(); } // 关闭 main
} // 关闭类
```

这个对象的作用是将MIDI信息组合成“乐曲”。

有问题！

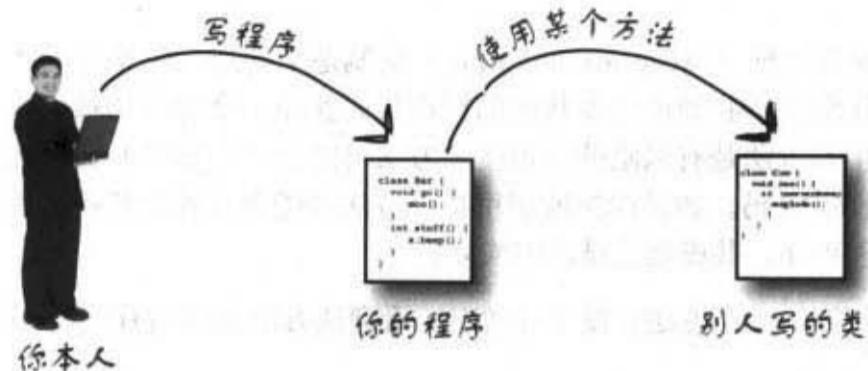
无法通过编译！编译器表示有异常状况必须要处理



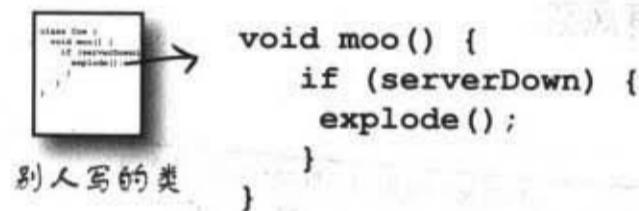
```
File Edit Window Help SayWhat?
* javac MusicTest1.java
MusicTest1.java:13: unreported exception javax.sound.midi.
MidiUnavailableException; must be caught or declared to be
thrown
    Sequencer sequencer = MidiSystem.getSequencer();
^
1 errors
```

调用有风险的方法（或许不是你自己写的）时会发生什么事？

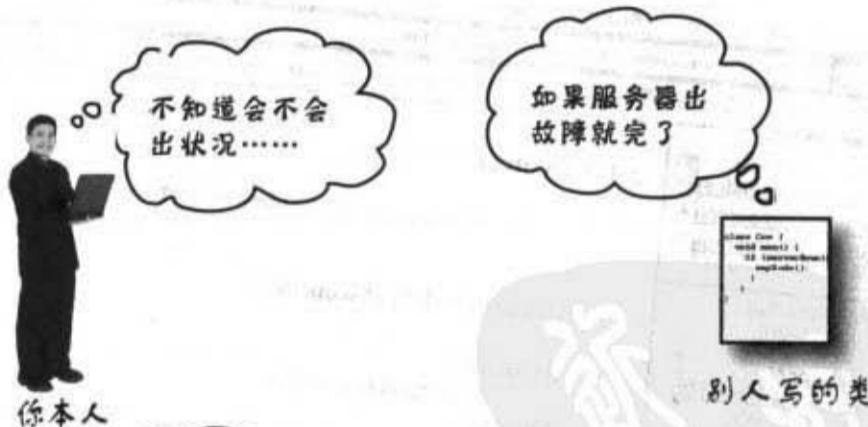
- ① 假设你调用了一个不是自己写的方法。



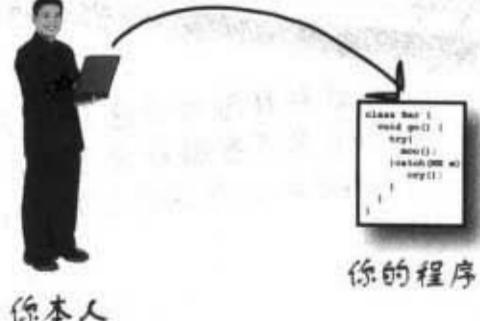
- ② 该方法执行某些有风险的任务，可能会在运行期间出状况。



- ③ 你必须认识到该方法是有风险的。



- ④ 你得写出可以在发生状况时加以处理的程序代码，未雨绸缪！



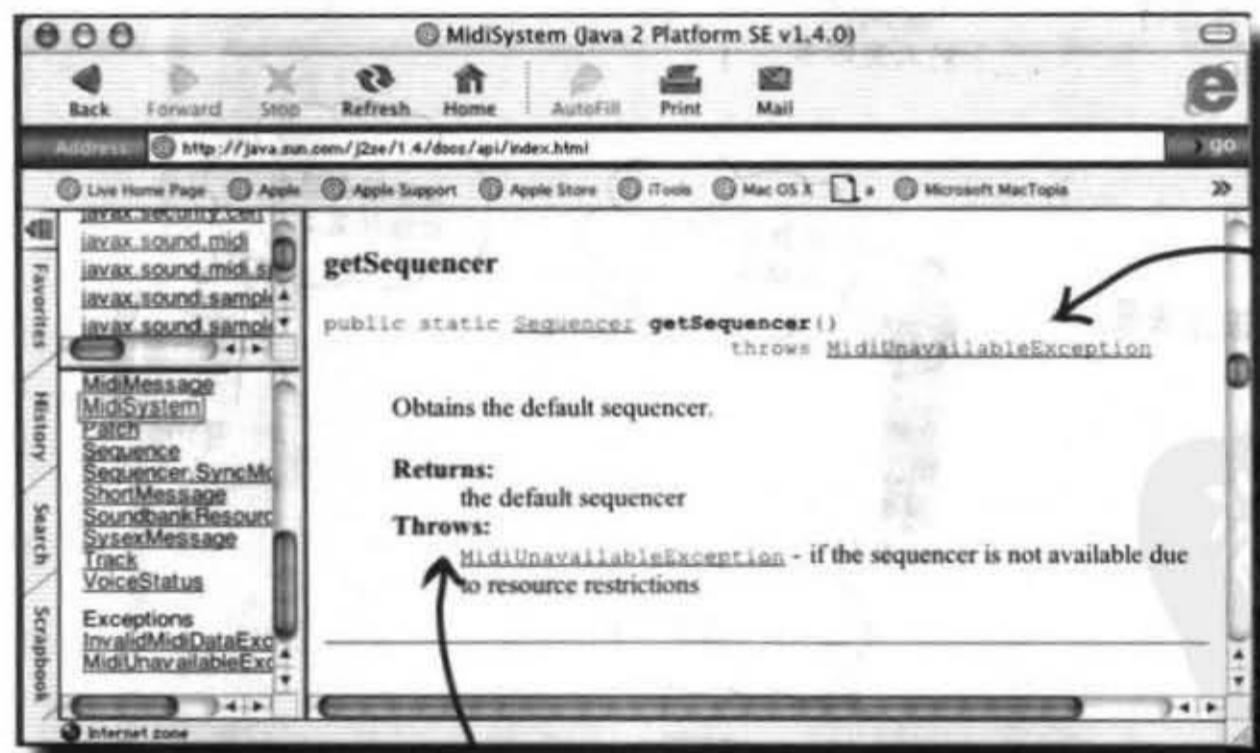
有可能出错的时候

Java 使用异常来告诉调用方法的程序代码：“有问题！我不行了！”

Java的异常处理(exception-handling)机制是个简捷、轻量化的执行期间例外状况处理方式，它让你能够将处理错误状况的程序代码摆在一个容易阅读的位置。这要依赖你已经知道所调用的方法是有风险的（也就是方法可能会产生异常），因此你可以编写出处理此可能的程序代码。如果你知道调用某个方法可能会有异常状况，你就可以预先准备好对问题的处理程序，甚或是从错误中恢复。

你要如何得知某个方法会抛出异常呢？看到该方法的声明有throws语句就知道了。

`getSequencer()`这个方法可能会在执行期间出问题，所以必须声明出调用它可能会有风险。



API文件说明
`getSequencer()`
可能会抛出
`MidiUnavailableException`
异常

此处解释何时会遇到异常，在
此情况下是因为资源受限，比
如sequencer已经被占用了

编译器要确定你了解所调用的方法是有风险的

如果你把有风险的程序代码包含在try/catch块中，那么编译器会放心许多。

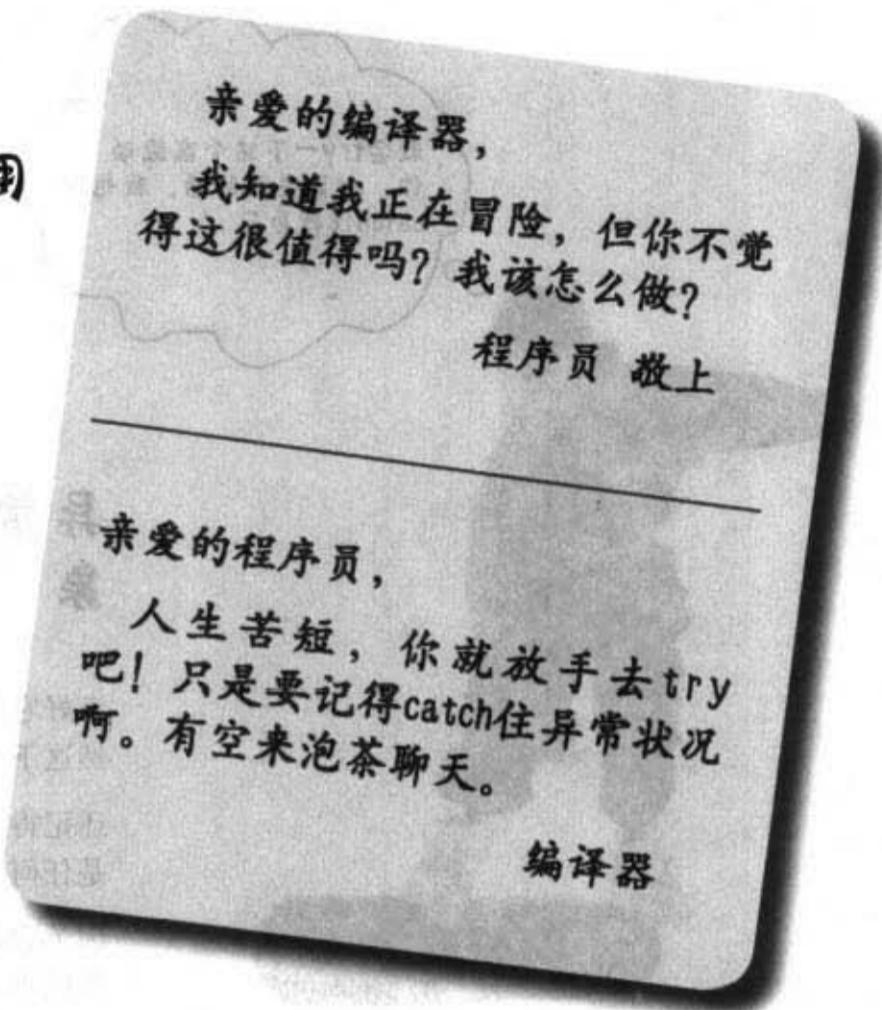
try/catch块会告诉编译器你确实知道所调用的方法会有风险，并且也已经准备好要处理它，它只会注意你有没有表示你会注意到异常。

```
import javax.sound.midi.*;

public class MusicTest1 {
    public void play() {

        try {
            Sequencer sequencer = MidiSystem.getSequencer(); ← 把有风险的程序
            System.out.println("Successfully got a sequencer");
        } catch (MidiUnavailableException ex) {
            System.out.println("Bummer");
        }
    } // 关闭播放

    public static void main(String[] args) {
        MusicTest1 mt = new MusicTest1();
        mt.play();
    } // close main
} // 关闭类
```



编译器



用catch块摆放异常状况的处理程序

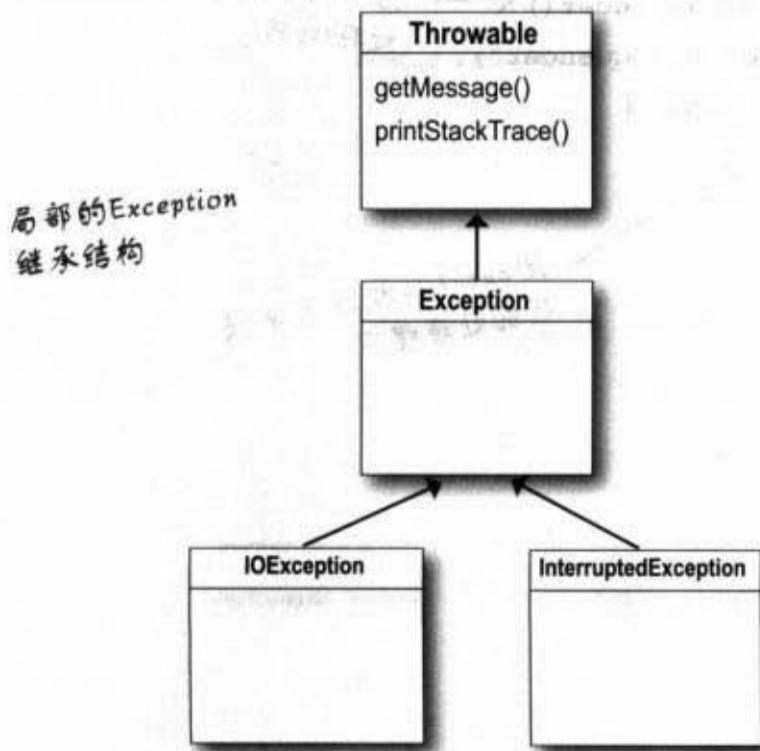


异常是一种Exception类型的对象

幸好它的类型名称不是Asabulu543AlubaE04Aligado，不然这下可难记了。

还记得关于多态的那一章提到Exception类型的对象可以是任何它的子类的实例吗？

因为它是对象，所以你catch住的也是对象。下面的程序代码中catch的参数是Exception类型的ex引用变量：



```
try {
    // 危险动作
} catch (Exception ex) {
    // 尝试恢复
}
```

就跟方法的参数声明一样

这一段口会在有抛出异常时执行

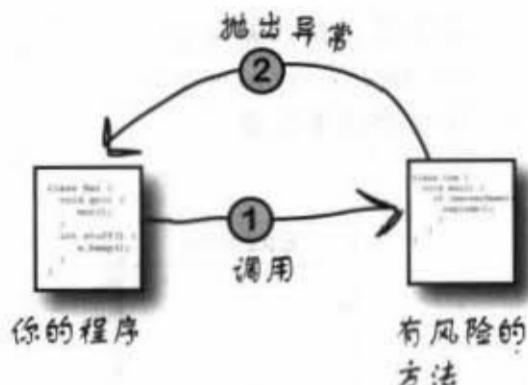
你写在catch块中的程序必定与所抛出的异常有关。例如，如果遇到服务器出故障的异常，你可能会在这里写寻找替代服务器的方法。

如果你的程序代码会抓住异常， 那是谁把它抛出来的？

你花在处理异常的程序设计时间会比花在自己创建与抛出异常的时间还多很多。现在只要知道当你的程序代码调用有风险的方法时，也就是声明有异常的方法，就是该方法把异常丢给你的。

实际上，两者可能都是你自己写的。由谁写的程序其实并不重要，重点在于哪个方法抛出异常与哪个方法抓到它。

在编写可能会抛出异常的方法时，它们都必须声明有异常。



① 有风险、会抛出异常的程序代码：

```

public void takeRisk() throws BadException {
    if (abandonAllHope) {
        throw new BadException();
    }
}
  
```

必须声明它会抛出
BadException

↑ 创建Exception对象并抛出

② 调用该方法的程序代码：

```

public void crossFingers() {
    try {
        anObject.takeRisk();
    } catch (BadException ex) {
        System.out.println("Aaargh!");
        ex.printStackTrace();
    }
}
  
```

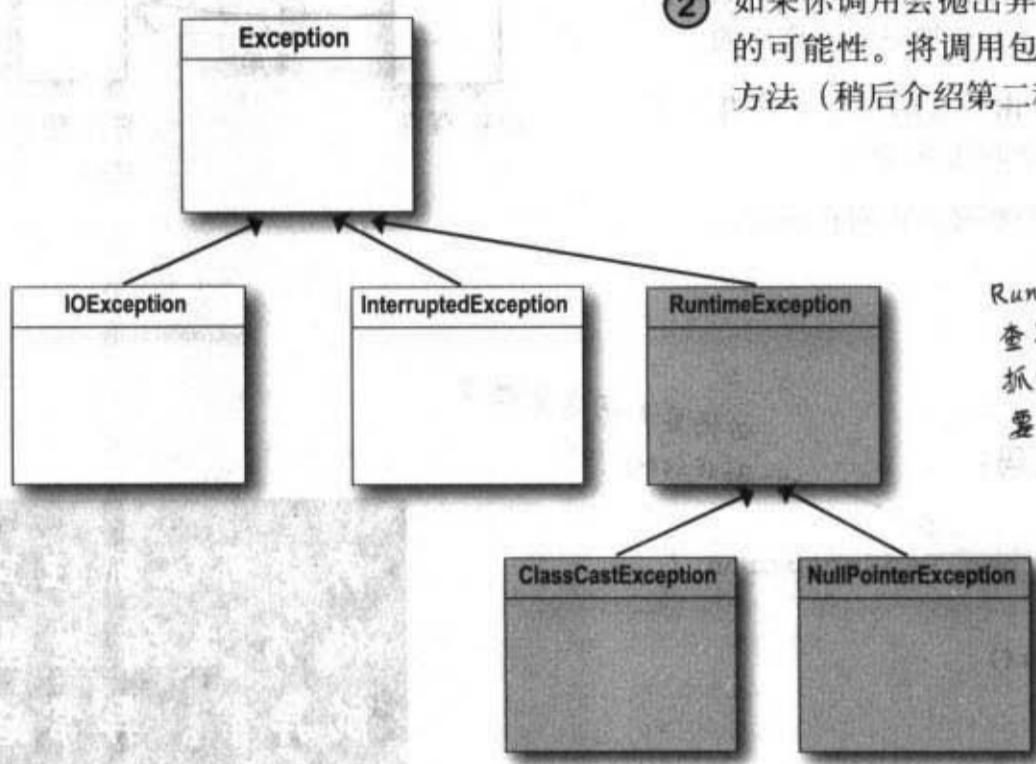
方法可以抓住其他方法所抛出的异常。异常总是会丢回给调用方。

会抛出异常的方法必须要声明它有可能会这么做。

如果无法从异常中恢复，至少也要使用
printStackTrace()来列出有用的信息

编译器会核对每件事，除了
Runtime Exceptions之外。编译器保证：

- ① 如果你有抛出异常，则你一定要使用throw来声明这件事。
- ② 如果你调用会抛出异常的方法，你必须得确认你知道异常的可能性。将调用包在try/catch块中是一种满足编译器的方法（稍后介绍第二种方法）。



there are no
Dumb Questions

问：等一下！这怎么会是我们第一次遇到必须处理的异常？我已经遇过`NullPointerException`和`DivideByZeroException`，甚至`Integer.parseInt()`也关照我`NumberFormatException`，为什么这些都不用处理？

答：除了`RuntimeException`这种特例之外，编译器会关照`Exception`所有的子类。任何继承过`RuntimeException`的类都不会受编译器关于是否声明它会抛出`RuntimeException`的检查，同样的，也不会管调用方是否认识到可能在运行期间遇到异常。

问：为什么编译器不管那些运行期间的异常？它们不也是会让整个程序跟着死掉吗？

答：大部分的`RuntimeException`都是因为程序逻辑的问题，而不是以你所无法预测或防止的方法出现的执行期失败状况，你无法保证文件一直都在。你无法保证服务器不会死机。但是你可以确保程序不会运行不合理的逻辑，例如对只有5项元素的数组取第八个元素的值。

你会需要在开发与测试期间发生`RuntimeException`，以便能够在不把程序代码放进try/catch块的情况下抓一开始就不应该出现的问题。

try/catch是用来处理真正的异常，而不是你程序的逻辑错误，该块要做的是恢复的尝试，或者至少会优雅的列出错误信息。

要点

- 方法可在运行期间遇到问题时抛出异常。
 - 异常是Exception类型的对象。
 - 编译器不会注意RuntimeException类型的异常。RuntimeException不需要声明或被包在try/catch的块中（然而你还是可以这么做）。
 - 编译器所关心的是称为检查异常的异常。程序必须要认识有异常可能的存在。
 - 方法可以用throw关键词抛出异常对象：
- ```
throw new FileIsTooSmallException();
```
- 可能会抛出异常的方法必须声明成throws Exception。
  - 如果程序调用了有声明会抛出异常的方法，就得要告诉编译器已经注意到这件事。
  - 如果要处理异常状况，就把调用包在try/catch块中，并将异常处理/恢复程序放在catch块中。
  - 如果不打算处理异常，还是可以正式地将异常给ducking来通过编译，稍后会解释ducking。

### metacognitive tip

如果想要学习某些事物，就把它当作是睡前最后学习的东西。当你放下书本时，就不要再做其他需要动脑的事情。大脑需要时间处理学习内容。这可能要几个钟头，如果此时有还有别的事情要想，之前所看过的东西可能会被遗忘。

当然啦，这跟技巧性的活动无关，或许平日的自由搏击或摔跤练习并不会影响到你对Java的学习。

最好的方法是在就寝前看这本书（就算只是看看图片也无所谓）。



### Sharpen your pencil

右列有哪些事情是你认为编译器会在乎的异常？我们只要找出无法在程序中控制的事情。第一项已经帮你写好了（因为它是最简单的）。

#### 要执行的工作

- ✓ 连接远程服务器
- 存取数组
- 显示Window
- 从数据库取出数据
- 判断文件是否存在
- 打开文件
- 从命令列读取字符

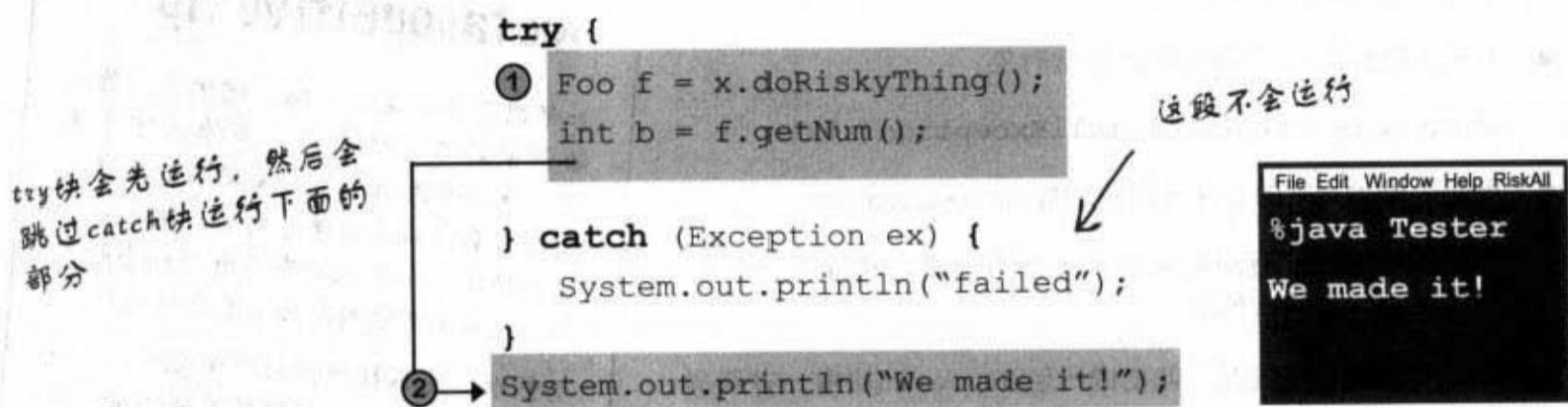
#### 会有什么问题？

- 服务器死掉

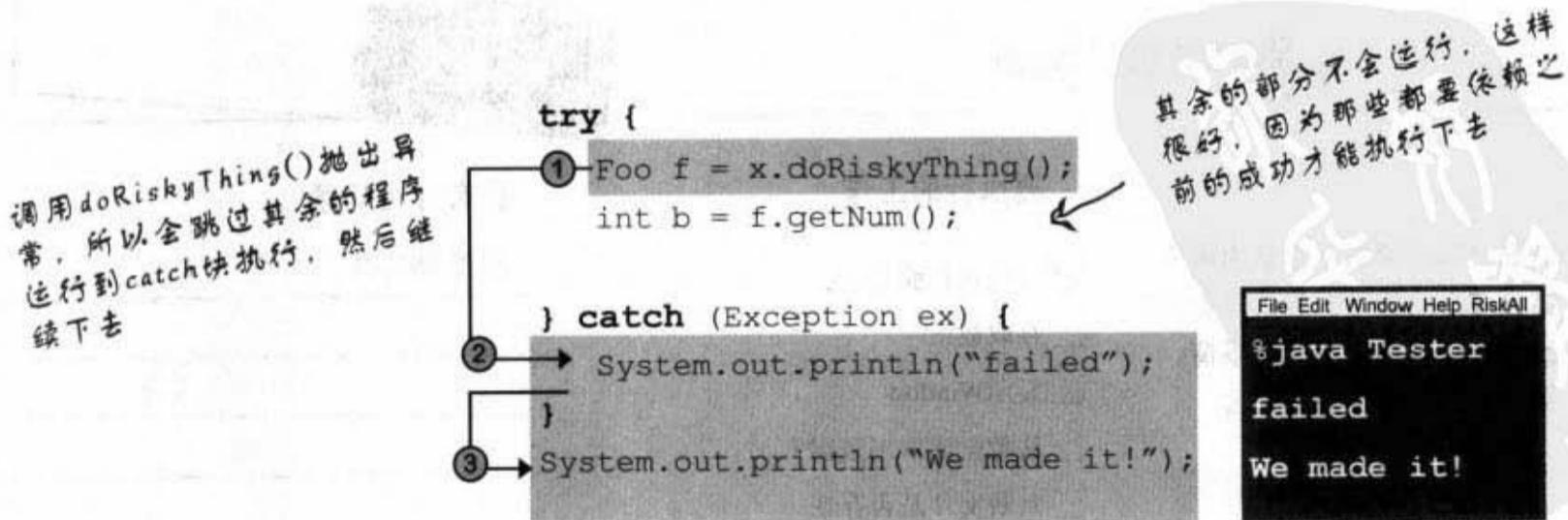
## try/catch块的流程控制

当你要调用有风险的方法时，有一两件事情可能会发生。  
该方法若不是成功地把try块完成的话，不然就是会把异常  
丢回调用方的方法。

如果成功的话：



如果失败：



## finally：无论如何都要执行的部分

如果你要煮东西吃，得先把炉子打开。

如果你的烹饪过程失败了，必须把炉子关掉。

如果你成功了，必须把炉子关掉。

不管怎样，你终究得关掉炉子。

finally块是用来存放不管有没有异常都得执行的程序。

```
try {
 turnOvenOn();
 x.bake();
} catch (BakingException ex) {
 ex.printStackTrace();
} finally {
 turnOvenOff();
}
```

如果没有finally，你得同时把turnOvenOff()摆在try与catch两处。finally块可以让你把所有重要的清理程序代码集中在一处，而不需要复制两份成下面这样：

```
try {
 turnOvenOn();
 x.bake();
 turnOvenOff();
} catch (BakingException ex) {
 ex.printStackTrace();
 turnOvenOff();
}
```



如果try块失败了，抛出异常，流程会马上转移到catch块。当catch块完成时，会执行finally块。当finally完成时，就会继续执行其余的部分。

如果try块成功，流程会跳过catch块并移动到finally块，当finally完成时，就会继续执行其余的部分。

如果try或catch块有return指令，finally还是会执行！流程会跳到finally然后再回到return指令。

 Sharpen your pencil

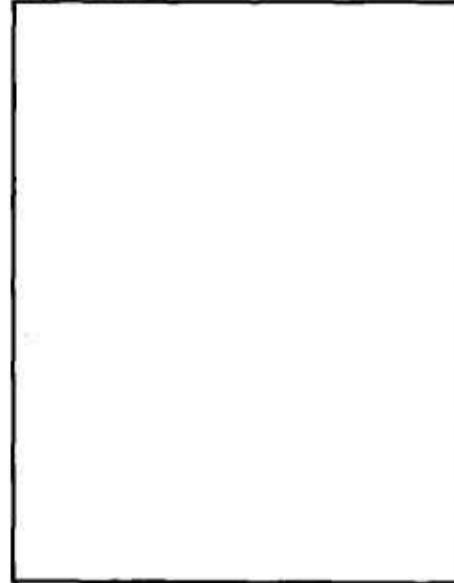
## 流程控制

假设ScaryException继承Exception，观察左方的程序代码。你认为这个程序会有怎样的输出？如果程序的第三行改成：String test = "yes"，会怎样？

```
public class TestExceptions {
 public static void main(String [] args) {
 String test = "no";
 try {
 System.out.println("start try");
 doRisky(test);
 System.out.println("end try");
 } catch (ScaryException se) {
 System.out.println("scary exception");
 } finally {
 System.out.println("finally");
 }
 System.out.println("end of main");
 }

 static void doRisky(String test) throws ScaryException {
 System.out.println("start risky");
 if ("yes".equals(test)) {
 throw new ScaryException();
 }
 System.out.println("end risky");
 return;
 }
}
```

输出当test = "no"



输出当test = "yes"



When test = "yes": start try - start risky - scary exception - finally - end of main  
 When test = "no": start try - start risky - end risky - end try - finally - end of main

## 我们讨论过方法可以抛出一个以上的异常吗？

如果有必要的话，方法可以抛出多个异常。但该方法的声明必须要有含有全部可能的检查异常（若两个或两个以上的异常有共同的父类时，可以只声明该父类就行）。

### 处理多重异常

编译器会检查你是否处理所有可能的异常。将个别的catch块逐个放在try块下。某些情况下catch出现的先后顺序会有影响，但这部分我们稍后再加以说明。

```
public class Laundry {
 public void doLaundry() throws PantsException, LingerieException {
 // 有可能抛出两个异常的程序代码
 }
}
```



声明两个可能的异常类型

```
public class Foo {
 public void go() {
 Laundry laundry = new Laundry();
 try {
 laundry.doLaundry();
 } catch(PantsException pex) {
 // 恢复程序代码
 } catch(LingerieException lex) {
 // 恢复程序代码
 }
 }
}
```



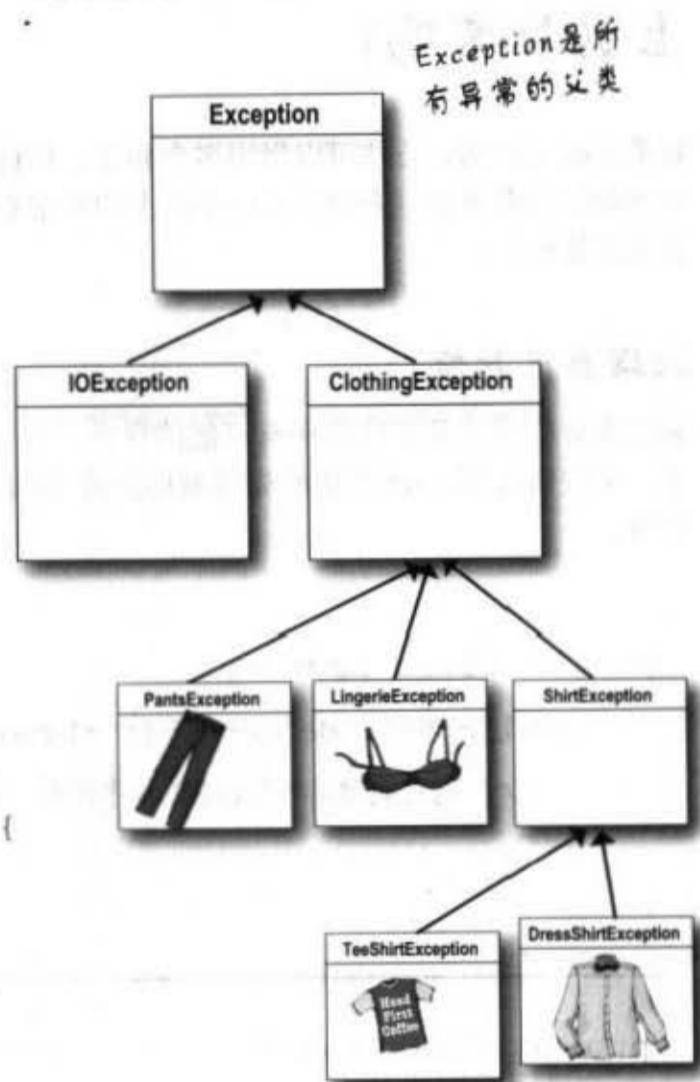
如果抛出的是PantsException，它就会运行到这个块



如果抛出的是LingerieException，则跳到这个段

## 异常也是多态的

别忘记异常是对象。除了可以被抛出之外，并没有什么特别的。因此如同所有的对象，异常也能够以多态的方式来引用。举例来说，`LingerieException`对象能被赋值给`ClothingException`的引用。`PantsException`也能够被赋值给`Exception`的引用。这样的好处是方法可以不必明确地声明每个可能抛出的异常，可以只声明父类就行。对于`catch`块来说，也可以不用对每个可能的异常作处理，只要有一个或少数几个`catch`可以处理所有的异常就够了。



### ① 以异常的父型来声明会抛出的异常

```
public void doLaundry() throws ClothingException {
```

声明成 `ClothingException` 可让你抛出  
任何 `ClothingException` 的子类。这代  
表此方法可以抛出 `PantsException`、  
`LingerieException` 等异常而不用个别的声  
明

### ② 以所抛出的异常父型来 `catch` 异常

```
try {
 laundry.doLaundry();
} catch(ClothingException cex) {
 // 解决方案
}
```

可 `catch` 任何  
`ClothingException` 的子  
类

```
try {
 laundry.doLaundry();
} catch(ShirtException sex) {
 // 解决方案
}
```

只能 `catch` 其两种子  
类

## 可以用super来处理所有异常并不代表就应该这么做

你可以把异常处理程序代码写成只有一个catch块以父型的Exception来捕获，因此就可以抓到任何可能被抛出的异常：

```
try {
 laundry.doLaundry();
} catch (Exception ex) {
 // 解决方案……
}
```

恢复什么？这会捕获所有的异常，因此你会搞不清楚哪里出错

## 为每个需要单独处理的异常编写不同的catch块

举例来说，如果你的程序代码处理TeeShirtException的方法与LingerieException的方法不同，则要个别地写出catch块。但如果ClothingException都是以同样的方式处理，则可以使用ClothingException的catch来处理。

```
try {
 laundry.doLaundry();
} catch (TeeShirtException tex) {
 // 恢复此问题
} catch (LingerieException lex) {
 // 恢复此问题
} catch (ClothingException cex) {
 // 恢复其他问题
}
```

两者的处理不同，所以使用不同的块

同样处理方法的都在这边

多个catch的顺序

## 有多个catch块时要从小排到大



catch(TeeShirtException tex)

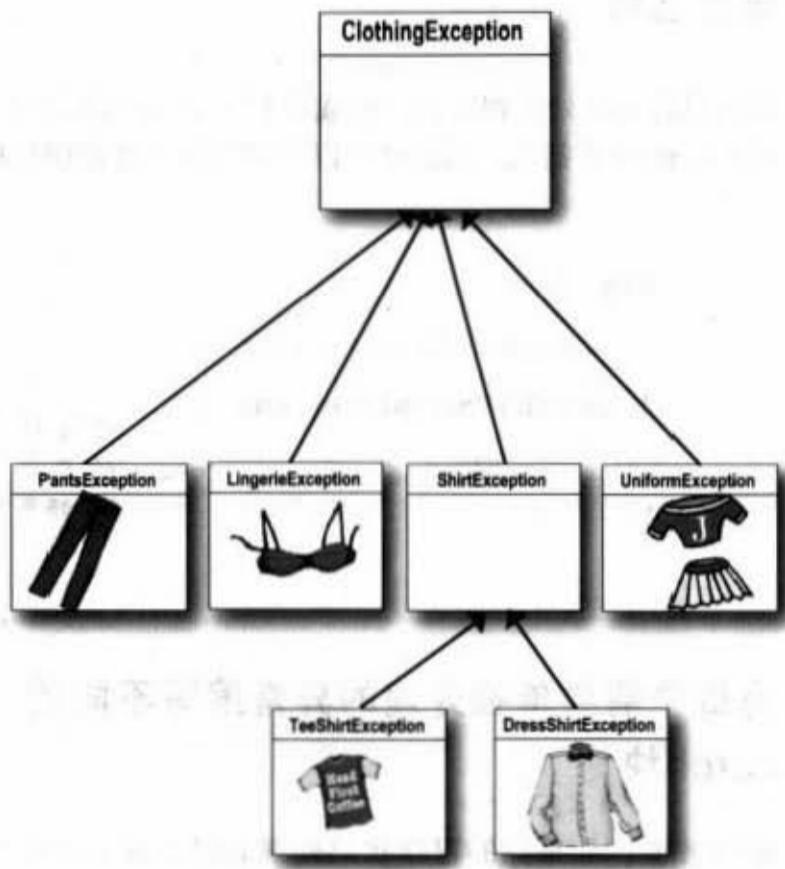
容得下此类型的子类



catch(ShirtException sex)



catch(ClothingException cex)



在继承树上层次越高，则“篮子”就越大。若你从上往下沿着继承层次走，异常类就会越来越有特定的取向，且catch的篮子也会越来越小。这是多态的常态现象。

ShirtException足以容下TeeShirtException或DressShirtException。而ClothingException甚至更大（能够引用的范围更多），但真的要说到大，Exception类型无疑是头号的霸王，它可以catch所有的异常，还包括了运行期间（unchecked）的异常，因此你或许不会把它用在测试以外的环境中。

## 不能把大篮子放在小篮子上面

嗯，你硬要这么做也可以，但是会无法通过编译。catch块不像重载的方法会被挑出最符合的项目。使用catch块时，Java虚拟机只会从头开始往下找到第一个符合范围的异常处理块。如果第一个catch就是catch(Exception ex)，则编译器会知道其余的都没有用处——绝对不会被用到。

亲爱的，大小很重要噢！小的要先上，不然就根本不会有使用机会

别这样！

```
try {
 laundry.doLaundry();
} catch(ClothingException cex) {
 // 恢复此问题
}
} catch(LingerieException lex) {
 // 恢复此问题
}
} catch(ShirtException sex) {
 // 恢复此问题
}
```



姐妹可以一起上……次序不重要，因为两者不会吞下对方的异常

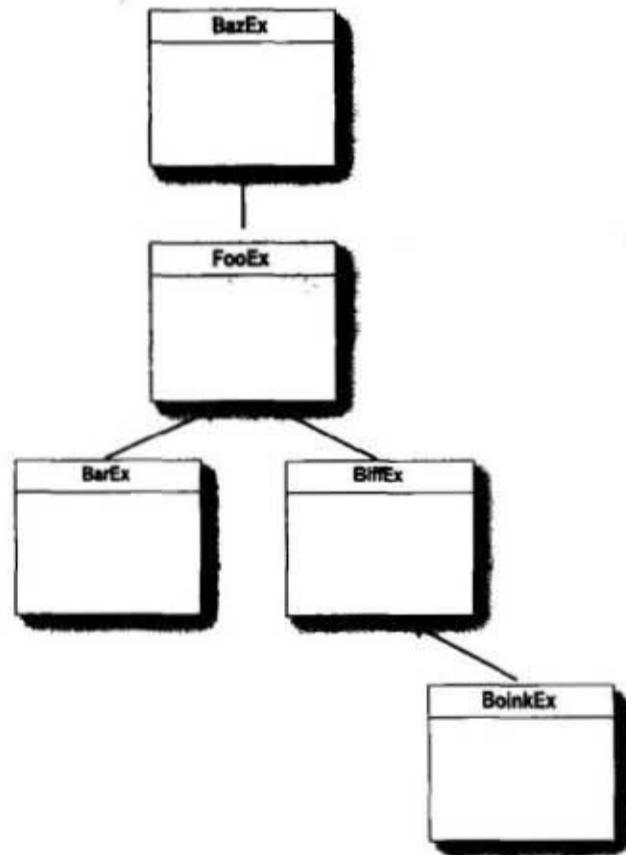
把ShirtException放在LingerieException上面不会有问题，因为ShirtException不会catch到LingerieException这个异常。



假设左方的块是合法的程序。你的任务是画两个不同的类图来精确地反映出Exception。换句话说，哪种类继承结构能够使这段程序代码为合法的？

```
try {
 x.doRisky();
} catch(AlphaEx a) {
 // 恢复此问题
} catch(BetaEx b) {
 // 恢复此问题
} catch(GammaEx c) {
 // 恢复此问题
} catch(DeltaEx d) {
 // 恢复此问题
}
```

你的工作是创建出两个不同的try/catch块结构（类似左上），以精确地展现左边的类图。假设所有的异常都会被方法中的try块所抛出。



不想处理异常时……

just duck it

如果不想处理异常，你可以把它 duck 掉来避开。

当你调用有危险的方法时，编译器需要你对这件事情有所表示。大部分情况下这代表说得把此调用包在try/catch块中。但也可以实行不同的方案：把它duck掉以让调用你的方法的程序来catch该异常。

这很容易，你只要表示出你会再throw此异常就好。技术上，其实它也不是你抛出的，不过这不重要。你只是让异常有出路而已，所以发生异常状况时会怎样呢？

方法抛出异常时，方法会从栈上立即被取出，而异常会再度丢给栈上的方法，也就是调用方，如果调用方是个ducker，则此ducker也会从栈被取出，异常再度抛给此时栈上方的方法，如此一路下去。何时会终止？稍后分晓。

```
public void foo() throws ReallyBadException {
 // 调用有风险的方法
 laundry.doLaundry();
}
```

够了！我不管了！谁调用我，谁就得自己处理异常状况



并没有try/catch块来处理有风险的方法，因此这个方法本身就是有风险的

## ducking只是在踢皮球

早晚还是得有人来处理这件事。但若连main()也duck掉异常呢？

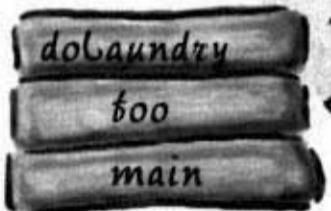
```
public class Washer {
 Laundry laundry = new Laundry();

 public void foo() throws ClothingException {
 laundry.doLaundry();
 }

 public static void main (String[] args) throws ClothingException {
 Washer a = new Washer();
 a.foo();
 }
}
```

两者都躲避异常，因此没人来处理，但有duck掉，所以可以通过编译

1 抛出  
ClothingException。



main()调用foo()  
foo()调用doLaundry()  
doLaundry()抛出  
ClothingException。

2 foo()已经duck掉  
异常。



doLaundry()从stack上被  
取走。  
异常抛给 foo()。

3 连main()也duck掉  
异常。



foo()也被取走……最后  
只剩下Java虚拟机，你  
知道这家伙对异常是没  
有什么责任感的。

4 Java虚拟机  
只好死给你  
看。

 我们使用T-Shirt来展示衣服的异常。其实我们  
知道你最喜欢的是比基尼。

## 处理或声明，做个堂堂正正的程序员

我们已经看过两种满足编译器的有风险方法调用方式。

### ● 处理。

把有风险的调用包在try/catch块中

```
try {
 laundry.doLaundry();
} catch (ClothingException cex) {
 // 恢复程序代码
}
```

这最好能处理掉所有  
doLaundry()可能抛出的异常，不然编译器还是会跟着句句念

### ● 声明 (duck 掉)。

把method声明成跟有风险的调用一样会抛出相同的异常

```
void foo() throws ClothingException {
 laundry.doLaundry();
}
```

有声明会抛出异常，但没有  
try/catch块，所以就全duck 掉  
异常留给调用方

这代表调用foo()的程序必须要处理或也跟着声明异常

```
public class Washer {
 Laundry laundry = new Laundry();

 public void foo() throws ClothingException {
 laundry.doLaundry();
 }

 public static void main (String[] args) {
 Washer a = new Washer();
 a.foo();
 }
}
```

要不就用try/catch块包起来，  
不然就duck掉

有问题！  
无法通过编译，且  
会有“unreported  
exception”错误信息

## 回到音乐播放程序……

你应该已经快要忘记这一章一开始要讲的是JavaSound程序代码。我们把sequencer对象创建出来，但因为Midi.getSequencer()声明了检查异常（MidiUnavailableException）使得程序无法通过编译。现在我们就可以通过包装在try/catch块的调用来解决这个问题。

```
public void play() {
 try {
 Sequencer sequencer = MidiSystem.getSequencer();
 System.out.println("Successfully got a sequencer");

 } catch(MidiUnavailableException ex) {
 System.out.println("Bummer");
 }
} // 关闭播放
```

包在try/catch块中  
就可以通过编译

catch的参数必须要是正确的异常，你一定要捕获有可能被抛出的异常

## 异常处理规则

- catch与finally不能没有try。

```
void go() {
 Foo f = new Foo();
 f.foo();
 catch(FooException ex) {}
}
```

缺了try!

- try一定要有catch或finally。

```
try {
 x.doStuff();
} finally {
 // 清理
}
```

这是合法的，但还要注意第四项

- try与catch之间不能有程序。

```
try {
 x.doStuff();
}
int y = 43;
} catch(Exception ex) {}
```

不能在这里放程序

- 只带有finally的try必须要声明异常。

```
void go() throws FooException {
 try {
 x.doStuff();
 } finally {}
}
```

## 程序料理



你不一定亲自下厨，但自己动手的乐趣还是比较多的。

本章接下来的部分是选择性的材料，你也可以下载已经写好的程序代码。

如果想知道更多的细节，就继续读下去吧！

## 实际发出声音

记得在本章开始时我们已经看过MIDI数据是如何保存应该演奏哪些音乐的指令，并且也了解到MIDI数据其实并没有夹带实际发出的声音。对于真正要给音箱发出的声音而言，MIDI的数据还需送到某种MIDI装置上，并将数据转换成声音。本书只使用软件装置来发声，以下是JavaSound的工作原理：

### 4项必备的条件：

- ① 发声的装置
- ② 要演奏的乐曲
- ③ 带有乐曲的信息记录
- ④ 乐曲的音符等信息



另外还需5个步骤：

① 取得Sequencer并将它打开。

```
Sequencer player = MidiSystem.getSequencer();
player.open();
```

② 创建新的Sequence。

```
Sequence seq = new Sequence(timing, 4);
```

③ 从Sequence中创建新的Track。

```
Track t = seq.createTrack();
```

④ 填入MidiEvent并让Sequencer播放。

```
t.add(myMidiEvent1);
player.setSequence(seq);
```



啊！天然的真好

## 史上第一个声音播放程序

把它输入然后运行看看。你会听到某物发出单一钢琴音。这是你个人的一小步，人类史上无关痛痒的一大步。

import javax.sound.midi.\*; ← 别忘了要import进midi的包

public class MiniMiniMusicApp {

    public static void main(String[] args) {  
        MiniMiniMusicApp mini = new MiniMiniMusicApp();  
        mini.play();  
    } // 关闭main

    public void play() {

        try {

            ① Sequencer player = MidiSystem.getSequencer();  
            player.open();

取得Sequencer并将其打开

            ② Sequence seq = new Sequence(Sequence.PPQ, 4);

先不用管参数的意义

            ③ Track track = seq.createTrack();

要求取得Track

            ④ ShortMessage a = new ShortMessage();  
            a.setMessage(144, 1, 44, 100);  
            MidiEvent noteOn = new MidiEvent(a, 1);  
            track.add(noteOn);  
  
            ShortMessage b = new ShortMessage();  
            b.setMessage(128, 1, 44, 100);  
            MidiEvent noteOff = new MidiEvent(b, 16);  
            track.add(noteOff);

对Track加入几个MidiEvent。要注意的是setMessage()的参数，以及MidiEvent的constructor，下一页会讨论

            player.setSequence(seq); ← 将Sequence送到Sequencer上

            player.start(); ← 开始播放

    } catch (Exception ex) {  
        ex.printStackTrace();  
    }  
} // 关闭播放  
} // 关闭类

## 制作 MidiEvent (乐曲信息)

MidiEvent是组合乐曲的指令。一连串的MidiEvent就好像是乐谱一样。我们会在乎的MidiEvent大部分都与描述要做的事情以及时机有关。因为MidiEvent是非常琐碎的描述，所以你必须指定何时开始播放某个音符(NOTE ON事件)以及何时停止(NOTE OFF事件)，因此你可以想象在“开始发出G音”之前发出“停止播放G音”是没有作用的。

MIDI指令实际上会放在Message对象中，MidiEvent是由Message加上发音时机所组成的。也就是说Message会带有“开始播放C”指令，并伴随着“于第四拍执行指令”的信息。

因此我们会同时需要MidiEvent与Message。

Message描述做什么，而MidiEvent指定何时做。

MidiEvent用来指示在何时执行什么操作。

每个指令都必须包括该指令的执行时机。

也就是说，乐声应该在哪一拍发响。

### ① 创建Message。

```
ShortMessage a = new ShortMessage();
```

### ② 置入指令。

```
a.setMessage(144, 1, 44, 100);
```

这代表发出44音符，其余的参数下一页会说明

### ③ 用Message创建MidiEvent。

```
MidiEvent noteOn = new MidiEvent(a, 1);
```

在第一拍启动a这个Message

### ④ 将MidiEvent加到Track中

```
track.add(noteOn);
```

Track带有全部的MidiEvent对象，Sequence会根据事件时间组织它们，然后Sequencer会根据此顺序来播放，同一时间可以执行多个操作，例如和弦声音或不同乐器的声音

## MIDI的Message是MidiEvent的关键

MIDI的Message带有事件中要执行什么操作的部分，也就是要sequencer实际执行的指令。指令的第一个参数是信息的类型，后面的3个参数要看信息的类型来决定它们的意义。例如144类型的信息代表“NOTE ON”。为了要带出NOTE ON指令，sequencer还需要知道几件事。你可以想象sequencer会问到：“好啊，我会发这个音，但是要用哪个频道？是鼓声的还是钢琴的频道？音量要多大？”

要创建MIDI的Message，用ShortMessage的实例调用setMessage()，传入该信息的4个参数。但要记住，你还需要把信息加上执行时机装入事件中。

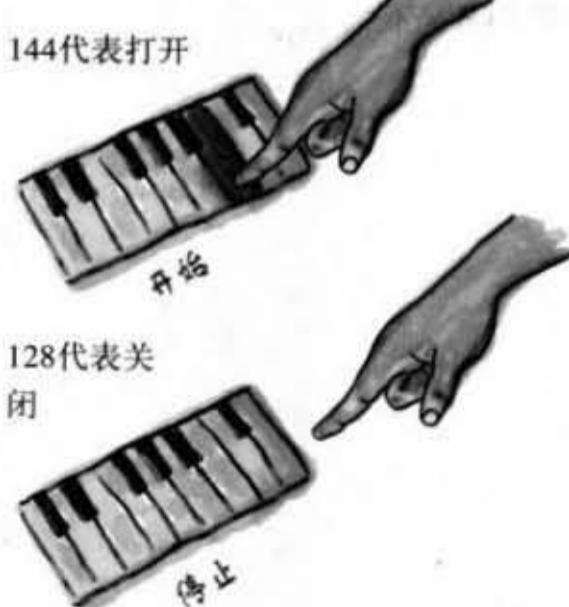
### 信息的格式

第一个参数是信息类型，其余3个要看信息类型而定。

Message是执行的内容，  
MidiEvent是执行的时机

```
a.setMessage(144, 1, 44, 100);
 ↑ ↑ ↑ ↑
 音高 频道 音名 音量
这是一个NOTE ON信息，因此其余三项参数用来描述所要发出的声响
```

#### ① 信息类型。



#### ② 频道。

每个频道代表不同的演奏者。例如一号频道是吉他、二号是Bass；或者可以像Iron Maiden用3把不同音色的吉他编制。

#### ③ 要发出的音符。

从0~127代表不同音高。



#### ④ 音量。

用多大的音量按下？0几乎听不到，100算是差不多。

## 改变信息

现在你已经知道Midi信息的内容，可以开始进行实验。你可以改变播放的音符、音长、加入更多音符、甚或是改变乐器。

### ① 改变音符。

试试看用0 ~ 127之间的数字来改变。

```
a.setMessage(144, 1, 20, 100);
```



### ② 改变音长。

对NOTE OFF的事件作些音长的变化。

```
b.setMessage(128, 1, 44, 100);
MidiEvent noteOff = new MidiEvent(b, 3);
```



### ③ 改变乐器。

在播放信息之前加入新的信息，换个不一样的乐器，从0 ~ 127之间找，看看还有什么音色。

```
first.setMessage(192, 1, 102, 0);
```

改变乐器的信息  
用第一个频道  
换成102的乐器



## 第二版：使用命令列参数

这个版本还是播放单一的音符而已，但你可以使用命令列参数来改变乐器和音符。试试看传入两个介于0~127之间的整数参数。第一个参数设定乐器，第二个整数设定音符。

```

import javax.sound.midi.*;
public class MiniMusicCmdLine { // 这是第一个
 public static void main(String[] args) {
 MiniMusicCmdLine mini = new MiniMusicCmdLine();
 if (args.length < 2) {
 System.out.println("Don't forget the instrument and note args");
 } else {
 int instrument = Integer.parseInt(args[0]);
 int note = Integer.parseInt(args[1]);
 mini.play(instrument, note);
 }
 } // 关闭main
 public void play(int instrument, int note) {
 try {
 Sequencer player = MidiSystem.getSequencer();
 player.open();
 Sequence seq = new Sequence(Sequence.PPQ, 4);
 Track track = seq.createTrack();
 MidiEvent event = null;
 ShortMessage first = new ShortMessage();
 first.setMessage(192, 1, instrument, 0);
 MidiEvent changeInstrument = new MidiEvent(first, 1);
 track.add(changeInstrument);
 ShortMessage a = new ShortMessage();
 a.setMessage(144, 1, note, 100);
 MidiEvent noteOn = new MidiEvent(a, 1);
 track.add(noteOn);
 ShortMessage b = new ShortMessage();
 b.setMessage(128, 1, note, 100);
 MidiEvent noteOff = new MidiEvent(b, 16);
 track.add(noteOff);
 player.setSequence(seq);
 player.start();
 } catch (Exception ex) {ex.printStackTrace();}
 } // 关闭播放
} // 关闭类

```

以两个介于0~127的整数参数来运行

```

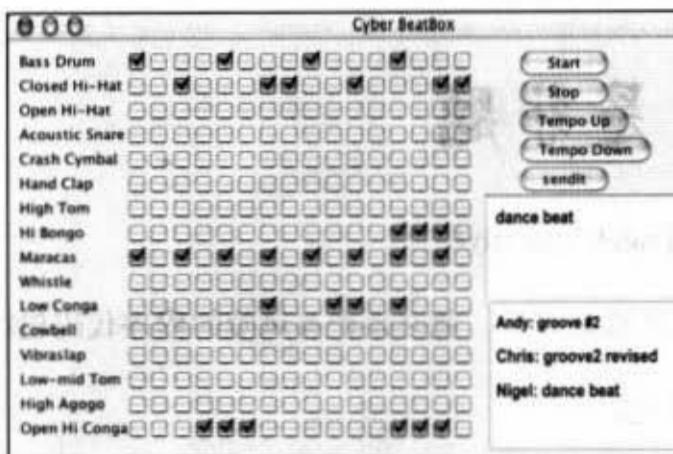
File Edit Window Help Attenuate
%java MiniMusicCmdLine 102 30
%java MiniMusicCmdLine 80 20
%java MiniMusicCmdLine 40 70

```

## 接下来的章节会继续料理程序

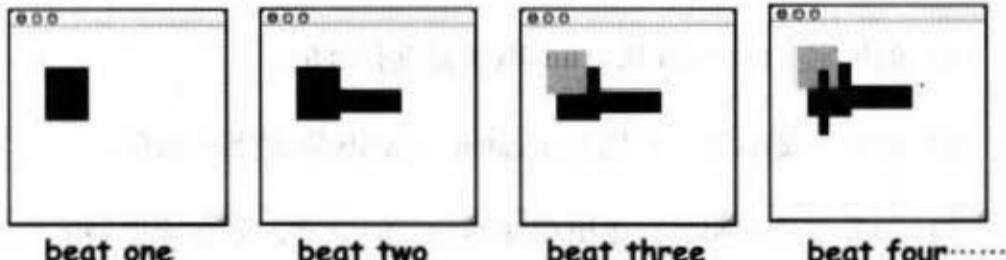
### 第15章：目的地

完成之后我们就会有一个BeatBox程序，它同时也是Drum Chat的客户端程序。我们需要学习GUI、I/O、网络和线程等。接下来的3章就包括这些内容。



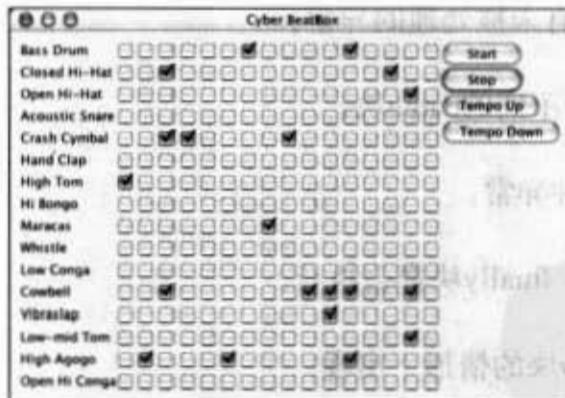
### 第12章：MIDI事件

这一章会创建出一个小小的“MTV”，它会根据MIDI音乐的节奏来画出随机的字符。这一章的内容能让我们学习到MIDI事件的处理。



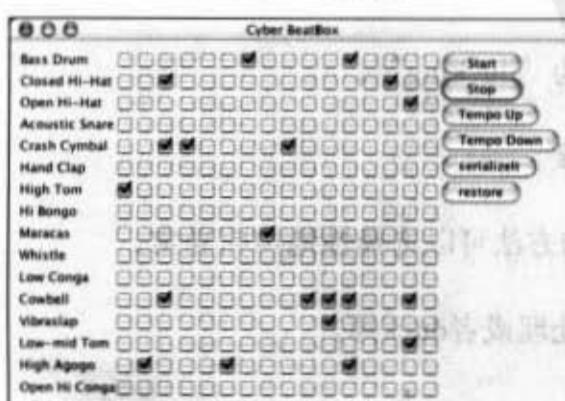
### 第13章：独立的BeatBox

我们确实创建出BeatBox、GUI等功能。但这还是很受限的版本。没有存储和还原的功能，也无法通过网络通信。



### 第14章：存储和还原

现在可以存盘与加载还原以再度播放了。这让我们可以准备好完成最终版本，把存盘发到网络另一端的chat服务器。





这一章讨论Java的异常。你的任务是辨别下面的陈述哪些是对的？哪些是错的？

## 是非题

- (1) try块必须要跟着catch与finally块后。
- (2) 如果遇到编译器的检查异常，就必须把有风险的程序代码包在try/catch块中。
- (3) catch块可以多态化。
- (4) 只有编译器的检查异常才会被捕获。
- (5) 如果定义try/catch块，finally块是选择性的。
- (6) 如果定义try块，可以加上catch、finally块或两者都有。
- (7) 如果方法声明可以抛出编译器检查的异常，则必须把抛出异常的程序代码包在try/catch块中。
- (8) main()方法必须处理所有未被处理的异常。
- (9) 单一的try块可以有多个不同的catch块。
- (10) 一个方法只能抛出一种异常。
- (11) 不管有没有抛出异常，finally块都会执行。
- (12) finally块可以在没有try块的情形下出现。
- (13) try块可以在没有catch或finally块的情形下单独出现。
- (14) 异常的处理有时被称为“ducking”。
- (15) catch块的顺序并不重要。
- (16) 带有try块和finally块的方法可以选择性地声明异常。
- (17) 运行期的异常必须要处理或者duck掉。



## 排排看

下面是被打散的Java程序片段。你是否能够将它们重新排列以成为可以编译与执行并产生如同下方的输出结果？注意到有些括号已经遗失，所以你可以在认为有需要时自行补上。

```
System.out.print(r);
try {
 System.out.print(t);
 doRisky(test);
}
System.out.println(s);
} finally {
 System.out.print(o);
}
```

```
class MyEx extends Exception { }

public class ExTestDrive {
```

```
System.out.print(w);
if (yes .equals(t)) {
 System.out.print(a);
 throw new MyEx();
} catch (MyEx e) {
```

```
static void doRisky(String t) throws MyEx {
 System.out.print(h);
```

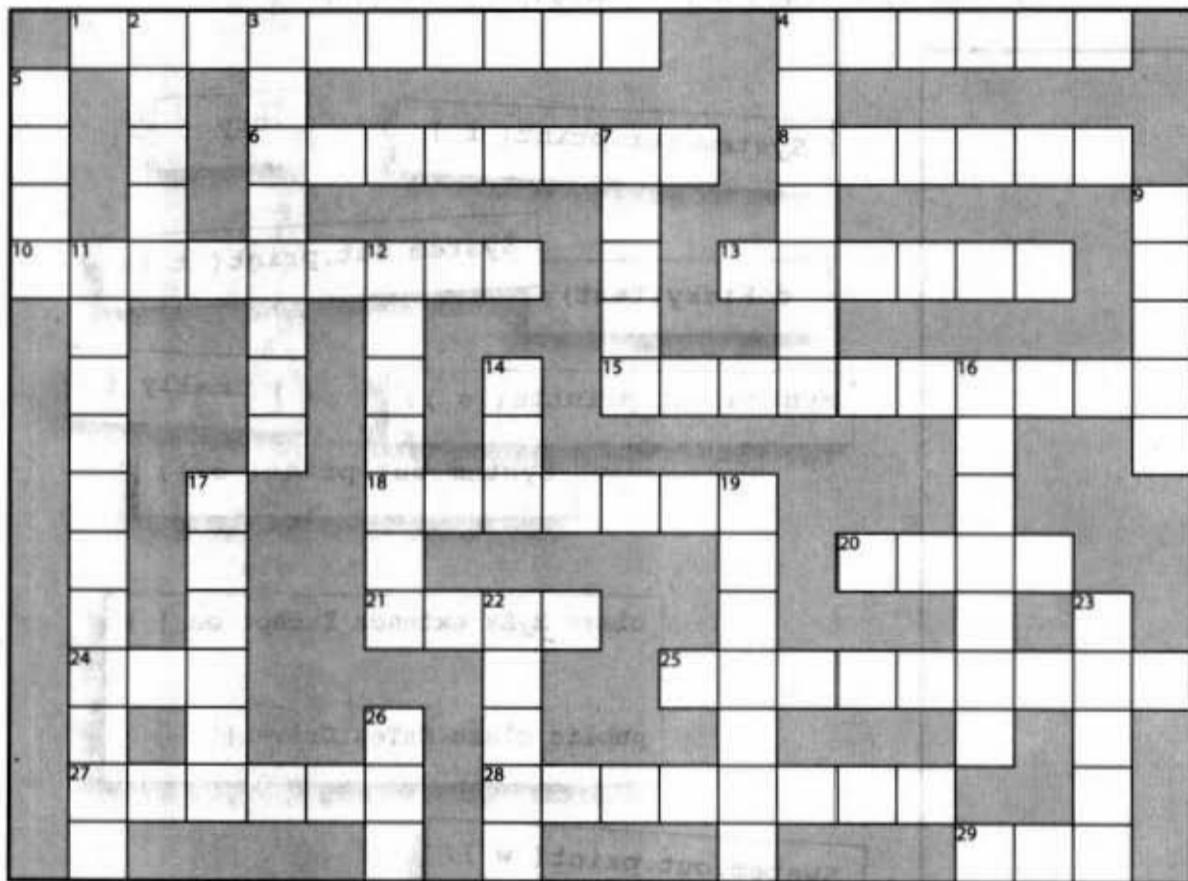
```
public static void main(String [] args) {
 String test = args[0];
```

```
File Edit Window Help ThrowUp
% java ExTestDrive yes
throws

% java ExTestDrive no
throws
```



# JavaCross 7.0



你知道该怎么做了吧？

(当然不是直接偷看答案)

## 横排提示：

- 1. To give value
- 4. Flew off the top
- 6. All this and more!
- 8. Start
- 10. The family tree
- 13. No ducking
- 15. Problem objects
- 18. One of Java's '49'
- 20. Class hierarchy
- 21. Too hot to handle
- 24. Common primitive
- 25. Code recipe
- 27. Unruly method action
- 28. No Picasso here
- 29. Start a chain of events

## 竖排提示：

- 12. Javac saw it coming
- 14. Attempt risk
- 16. Automatic acquisition
- 17. Changing method
- 19. Announce a duck
- 22. Deal with it
- 23. Create bad news
- 26. One of my roles
- 20. Roll another one off the line
- 21. Starts a problem
- 23. For \_\_\_\_\_ (not example)
- 25. the family fortune
- 27. Starts a problem
- 28. Not Abstract
- 3. For \_\_\_\_\_ (not example)
- 5. Numbers ...
- 7. Not a 'getter'
- 9. Only public or default
- 11. Starts a method
- 13. Instead of declare
- 15. Starts a method
- 16. Starts a problem
- 17. Starts a problem
- 19. Starts a problem
- 21. Starts a problem
- 23. Starts a problem
- 25. Starts a problem
- 27. Starts a problem
- 28. Starts a problem

## 补充提示：

(字谜的问题部分保留原汁原味的英文, 请自己动手查字典! )



## 练习解答

## 排排看

### 是非题

- (1) 错, 任一或两者皆可。
- (2) 错, 也可以声明异常。
- (3) 对。
- (4) 错, 运行期间异常也会被捕获到。
- (5) 对。
- (6) 对。
- (7) 错, 只要声明就够了。
- (8) 错, 但如果连它也不管, 那Java虚拟机只好中断。
- (9) 对。
- (10) 错。
- (11) 对。
- (12) 错。
- (13) 错。
- (14) 错, duck在这里有声明的意思。
- (15) 错, 越广泛的异常越要在后面。
- (16) 错, 如果没有catch就要duck。
- (17) 错。

```

class MyEx extends Exception { }

public class ExTestDrive {
 public static void main(String [] args) {
 String test = args[0];
 try {
 System.out.print("t");
 doRisky(test);
 System.out.print("o");
 } catch (MyEx e) {
 System.out.print("a");
 } finally {
 System.out.print("w");
 }
 System.out.println("s");
 }

 static void doRisky(String t) throws MyEx {
 System.out.print("h");
 if ("yes".equals(t)) {
 throw new MyEx();
 }
 System.out.print("r");
 }
}

```

```

File Edit Window Help Chill
% java ExTestDrive yes
throws

% java ExTestDrive no
throws

```



## JavaCross答案

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|
|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |   |   |   |   |
|   | A | S | S | I | G | N | M | E | N | H  | E  | R  | A  | R  | C  | H  | Y  | K  | E  | Y  | W  | O  | R  | D  | A  | L  | G  | O  | R  | I | T | H | M |
| M | C |   | N |   | S | U | B | C | L | A  | S  |    |    |    |    |    |    |    |    |    |    |    |    | R  |    |    |    |    |    |   |   |   |   |
| A | O |   |   |   | S | U | B | C | L | A  | S  |    |    |    |    |    |    |    |    |    |    |    |    | I  | N  | V  | O  | K  | E  |   |   |   |   |
| T | P |   | T |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | T  | V  |    |    |    |   |   |   |   |
| H | I | E | R | A | R | C | H | Y |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    | A  | H  | A  | N  | D  | L  | E |   |   |   |
| N |   |   |   |   | N | H |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    | T  | T  |    |    |    |    |   |   |   |   |
| S |   |   |   |   | C | E |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    | E  | X  | C  | E  | P  | T  | I | O | N | S |
| T |   |   | E |   | C | R |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    | N  |    |    |    |    |    |   |   |   |   |
| A |   | S |   |   | E |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    | H  |    |    |    |    |    |   |   |   |   |
| N | E |   |   |   | E |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    | T  | R  | E  | E  |    |    |   |   |   |   |
| T | T |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    | C  |    | R  | T  |    |    |   |   |   |   |
| I | N | T |   |   |   |   |   | A |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    | E  |    | T  | R  | E  | E  |   |   |   |   |
| A | E |   |   |   |   | I |   | T |   |    |    |    |    |    |    |    |    |    |    |    |    |    | A  |    |    |    |    |    |    |   |   |   |   |
| T | H | R | O | W | S |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    | E  |    |    |    |    |    |    |   |   |   |   |
| E |   |   |   |   | A |   |   | H |   |    |    |    |    |    |    |    |    |    |    |    |    |    | E  |    |    |    |    |    |    |   |   |   |   |

### 看图说故事



面对现实吧，你得做出图形用户接口。如果你要创建别人会使用的应用程序，就必须用到图形接口。如果你是写给自己用，那应该也会想要有个图形接口吧。即使你只会写服务器端的应用程序，客户端的用户接口是由Web页面组成的，早晚你还是得写图形化的工具接口。当然啦，命令行的程序一样可以使用，但逃避并不是个好办法。命令行又差又没适应性，而且也不好用。我们会用两章的篇幅来讨论GUI，并且同时学习包括事件处理与内部类别等Java语言的关键功能。在这一章我们会制作按钮、绘制屏幕画面、显示jpeg图文件，另外还做动画。

## 一切都从window开始

JFrame是个代表屏幕上window的对象。你可以把button、checkbox、text字段等接口放在window上面。标准的menu也可以加到上面，并且能够带最小化、最大化、关闭等图标。

JFrame的长相会依据所处的平台而有所不同。下图是JFrame在Mac OS X上的样子：



## 将组件加到window上

一旦创建出JFrame之后，你就可以把组件(widget)加到上面。有很多的Swing组件可以使用，它们是在javax.swing这个包中。最常使用的组件包括：JButton、JRadioButton、JCheckBox、JLabel、JList、JScrollPane、JSlider、JTextArea、JTextField和JTable等。大部分都很容易使用，但像JTable这些是有点复杂的。



“如果再让我看到一个命令行的程序，你就得走人”

创建GUI真简单：

① 创建frame。

```
JFrame frame = new JFrame();
```

② 创建widget。

```
JButton button = new JButton("click me");
```

③ 把widget加到frame上。

```
frame.getContentPane().add(button);
```

组件不会直接加到frame上，你可以把frame想象成window的框，组件是加到window的pane上面

④ 显示出来。

```
frame.setSize(300,300);
frame.setVisible(true);
```

# 你的第一个GUI

```

import javax.swing.*; ← 别忘了import进这个包

public class SimpleGuil {
 public static void main (String[] args) {
 JFrame frame = new JFrame(); ← ① 建frame和button
 JButton button = new JButton("click me");

 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 ← 这一行程序会在window关闭时把程序结束掉

 frame.getContentPane().add(button);
 ← 把button加到frame的pane上

 frame.setSize(300,300); ← 设定frame的大小
 ←
 frame.setVisible(true);
 }
}

← 最后把frame显示出来!

```

执行时会是怎样的情景：

%java SimpleGuil



啊！

好大一个按钮啊！

这个按钮会填满整个frame。稍后我们会讨论如何控制按钮的大小和位置。

## 但是按钮没有功能……

其实并不是这样。当你按下按钮时，它会显示出“被按下”或“被推倒”的外观（实际的样子要看平台，但一定会显示出让你能够分辨的形状）。

所以这个问题应该要这样问：如何让按钮在用户按下时执行特定的工作？

### 需要这两项：

- ① 被按下时要执行的方法（也就是按钮的任务）。
- ② 检测按钮被按下的方法，换句话说，就是按钮的感应装置。



我们得想办法知道鼠标是否被按下了。

所以说，我们很在意“用户按下按钮”这个事件。

\* pushed的按钮应该要翻译成维持在按下状态的按钮，或者你可以把它想象成开关的状态，翻译成被推倒是因为译者个人的兴趣……

*there are no  
Dumb Questions*

**问：** 按钮在Windows上面运行的时候会不会就长得跟其他Windows程序的按钮一样？

**答：** 想要的话也可以。你能够使用一些核心函数库的“look and feel”类型，该类来控制界面的外观和操作感受。大部分情况下都有两种可以选：称为Metal的Java标准和平台原始界面两种。本书展示的是Mac OS X的Aqua外观或Metal外观。

**问：** 能不能在Windows上使用Aqua外观？

**答：** 你做梦。不是所有外观和操作感受都能够在不同平台上发现的。要取得跨平台的相同外观就得使用Metal，不然就不要指定，让外观使用平台的默认值。

**问：** 听说Swing很慢并且根本没人用？

**答：** 以前是这样，现在可就不同了。如果机器很烂，那你可能会用得很痛苦，但对于稍微像样一点的机器来说，你根本感受不到速度有差别。很多应用程序都用到Swing。

## 取得用户的事件

假设你想要把按钮上面的文字在用户按下按钮时从“click me”变成“I've been clicked”。首先你得编写改变按钮文字的方法（查一下API你就会知道怎么改）：

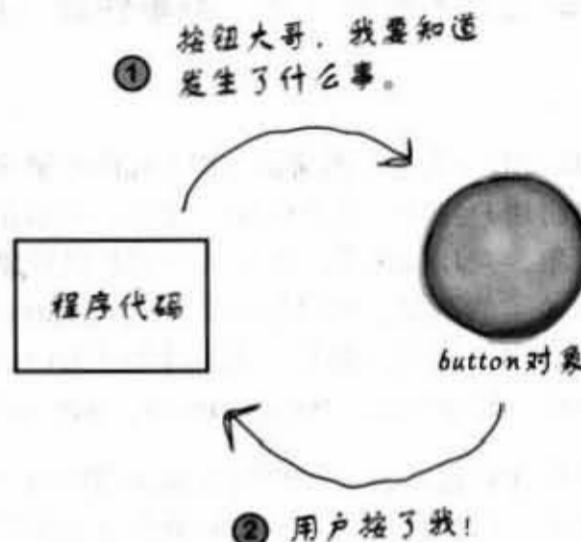
```
public void changeIt() {
 button.setText("I've been clicked!");
}
```

然后呢？这个方法应该在什么时候执行？我们怎么知道按钮被按下去呢？

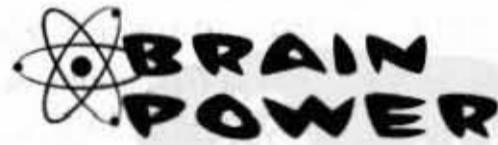
在Java上，取得与处理用户操作事件的过程称为even-handling。Java有许多不同的事件类型，大多数都与GUI上的用户操作有关。如果用户按下了按钮，就会产生事件。这是一个关于用户想要采取启动按钮功能的事件。如果这是个“放慢节奏”的按钮，就表示说用户想要让节奏慢一点。所以最直截了当的事件就是这种表明要执行某种操作的事件。

对这样的按钮来说，你不在乎像是按钮正被按住或按钮已经放开这类的过渡性事件。你只想知道用户是不是想要采取某种行动。也就是说你不管鼠标是否一直按着不放之类的事情，只管用户真正的意图！

首先，按钮要知道它的作用。



其次，按钮要在按键事件发生时调用执行功能的方法。



(1) 你要怎样告诉按钮对象你在乎它的事件？如何表白你对它的关心？

(2) 假设你不能把方法的名字告诉按钮，按钮要怎么通知你？我们怎样确保当某特定事件发生时会调用指定的方法？

## 如果想要知道按钮的事件，就会监听事件的接口

监听接口是介于监听（你）与事件源（按钮）间的桥梁

Swing的GUI组件是事件的来源。以Java的术语来说，事件来源是个可以将用户操作（点击鼠标、按键、关闭窗口等）转换成事件的对象。对Java而言，事件几乎都是以对象来表示。它会是某种事件类的对象。如果你查询API中的java.awt.event这个包，就会看到一组事件的类（名称中都有Event）。你会看到MouseEvent、KeyEvent、WindowEvent、ActionEvent等等。

事件源（例如按钮）会在用户做出相关动作时（按下按钮）产生事件对象。你的程序在大多数的情况下是事件的接受方而不是创建方。也就是说，你会花较多的时间当监听者而不是事件来源。

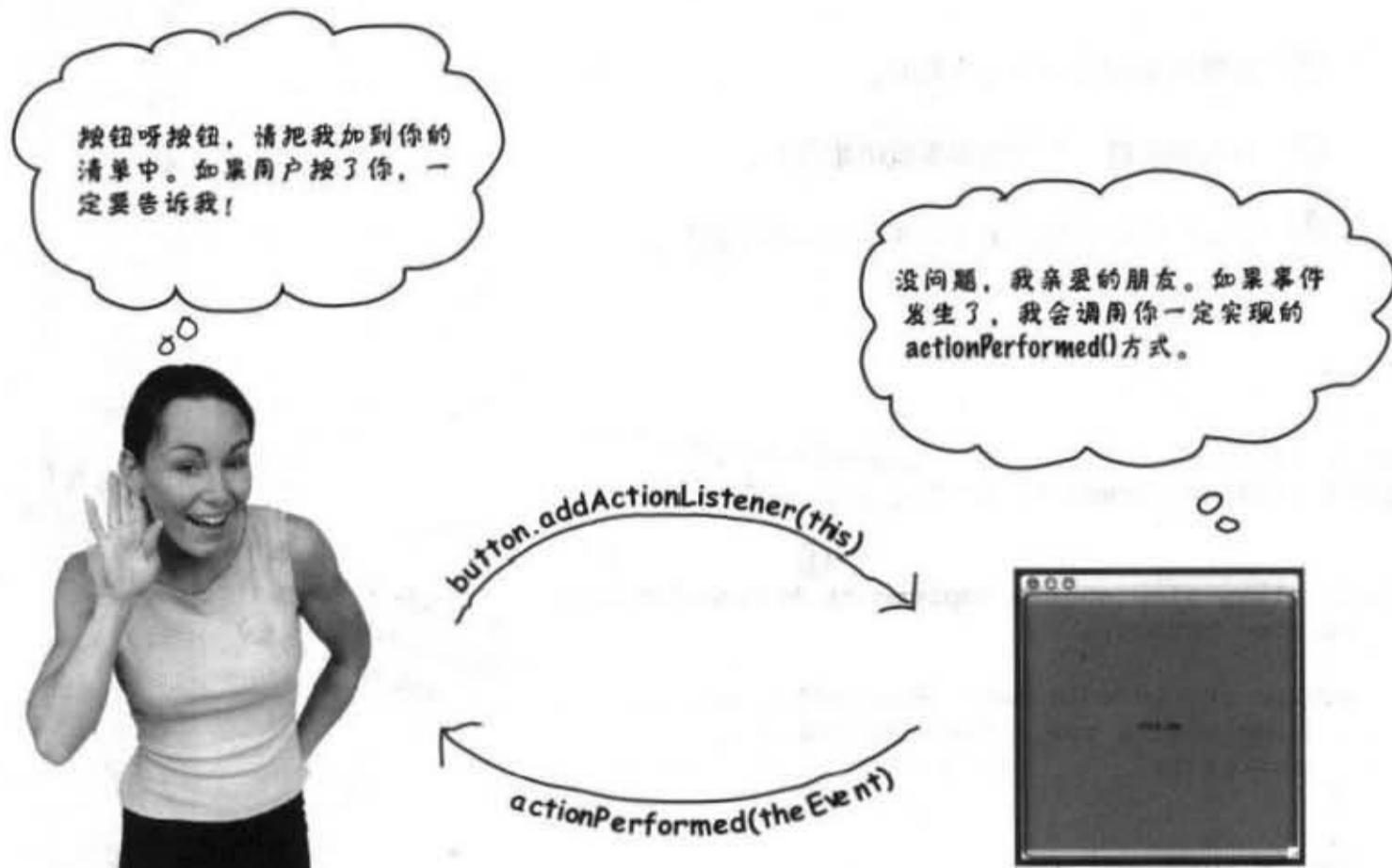
每个事件类型都有相对应的监听者接口。想要接收MouseEvent的话就实现MouseListener这个接口。想要WindowEvent吗？实现WindowListener。记得接口的规则：要实现接口就得声明这件事（各位乡亲注意啦，Dog实现Pet了！），这代表你必须把接口中所有的方法都实现出来。

某些接口不只是一个方法，因为事件本身就有不同的形态。以MouseListener为例，事件就有mousePressed、mouseReleased、MouseMoved等。虽然都是MouseEvent，每个鼠标事件都在接口中有不同的方法。如果有实现MouseListener的话，mousePressed()就会在用户按下鼠标的时候被调用。当按键放开时会调用mouseReleased()。因此鼠标事件只有MouseEvent一种事件对象，却有不同的事件方法来表示不同类型的鼠标事件。

实现监听接口让按钮有一个回头调用程序的方式。interface正是声明调用（call-back）方法的地方。



## 监听和事件源如何沟通



### 监听

如果类想要知道按钮的ActionEvent, 你就得实现 ActionListener这个接口。按钮需要知道你关注的部分, 因此要通过调用 `addActionListener(this)` 并传入 `ActionListener` 的引用 (此例中就是你自己的这个程序, 所以使用 `this`) 来向按钮注册。按钮会在该事件发生时调用该接口上的方法。而作为一个 `ActionListener`, 编译器会确保你实现此接口的 `actionPerformed()`。

### 事件源

按钮是 `ActionEvent` 的来源, 因此它必须要知道有哪些对象是需要事件通知的。此按钮有个 `addActionListener()` 方法可以提供对事件有兴趣的对象 (`listener`) 一种表达此兴趣的方法。

当按钮的 `addActionListener()` 方法被调用时 (因为某个 `listener` 的调用), 它的参数会被按钮存到清单中。当用户按下按钮时, 按钮会通过调用清单上每个监听的 `actionPerformed()` 来启动事件。

## 取得按钮的ActionEvent

- 实现 ActionListener这个接口。
- 向按钮注册（告诉它你要监听事件）。
- 定义事件处理的方法（实现接口上的方法）。

```
import javax.swing.*; import 进 ActionListener 和
import java.awt.event.*; ActionEvent 所在的包

public class SimpleGuilB implements ActionListener {
 JButton button;

 public static void main (String[] args) {
 SimpleGuilB gui = new SimpleGuilB();
 gui.go();
 }

 public void go () {
 JFrame frame = new JFrame();
 button = new JButton("click me");

 button.addActionListener(this); 向按钮注册

 frame.getContentPane().add(button);
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 frame.setSize(300,300);
 frame.setVisible(true);
 }

 public void actionPerformed(ActionEvent event) {
 button.setText("I've been clicked!");
 }
}
```

实现此接口。这表示 SimpleGuilB 是一个 ActionListener (事件只会通知有实现 ActionListener 的类)

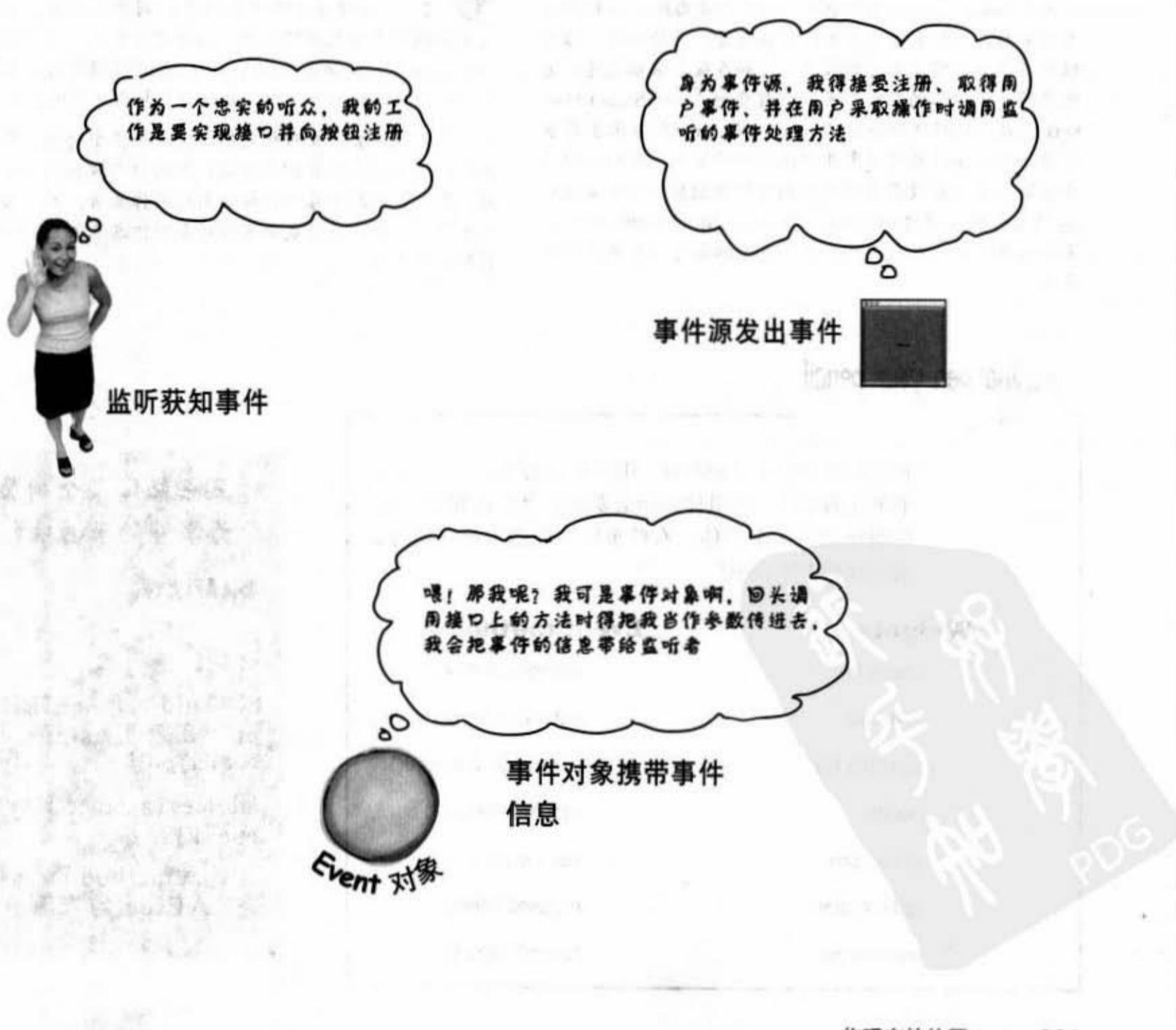
实现 interface 上的方法……这是真正处理事件的方法！

按钮会以 ActionEvent 对象作为参数来调用此方法

## 监听、事件源与事件

对一般的Java程序员来说，职业生涯中很少会有成为事件源的机会（不管你有多爱表现、多喜欢上台表演、多希望成为注目的焦点，就算经常把照片贴上网络也一样）。

接受事实吧，你的工作是监听（如果能够做好这件事，你也会成为异性朋友眼中诚恳的“好人”！）。



there are no  
Dumb Questions

**问：** 我为什么不是事件源？

**答：** 你也可以是。我们只是说大部分的时间你会是接收事件的一方而不是来源（至少在你早期的Java职业生涯中不是）。大部分你会用到的事件是由Java API中的类发出的，而你会是这些事件的监听者。无论如何，你可以设计需要自定义事件的程序，例如在股票涨幅超过一定程度的时候从你的股票交易监控程序抛出一个StockMarketEvent。此例中你会创建出StockWatcher对象来当作事件源，并对你的自定义事件创建监听的接口，提供注册的方法等。之后在股票事件发生的时候就创建一个StockEvent对象并把它通过调用stockChanged(StockEvent)的方法来传给监听者。别忘记每个事件类型都要有相对应的监听接口。

**问：** 我看不出传给事件调用方法的事件对象有什么重要性。调用mousePressed时可能有哪些信息会被用到？

**答：** 大部分情况下你不会用到事件对象。它只不过是个携带事件数据的载体。但有时你也会需要查询事件的特定细节。例如你的mousePressed()被调用时，你知道有鼠标的按钮被按下。但如果你想要知道鼠标的坐标呢？或者有时候你会想要对相同的监听注册多个对象。举例来说，计算器程序会有10个按键，且因为都做相同的事情，所以你可能不想为每个按键个别地制作监听。所以当你收到事件时，可以设计成用事件对象的信息来判别哪一个按钮触发了事件。

### Sharpen your pencil

下列的每个组件（widget，用户接口对象）是一或多个事件的来源。把组件可能会发出的事件连起来。有些组件会有多个事件，有些事件可能会有多个来源。看不懂就查字典吧！

#### Widgets

check box  
text field  
scrolling list  
button  
dialog box  
radio button  
menu item

#### Event methods

windowClosing()  
actionPerformed()  
itemStateChanged()  
mousePressed()  
keyTyped()  
mouseExited()  
focusGained()

你怎么知道某个对象是否为事件的来源呢？

查询API文件。

好的，查什么？

以“add”开头“Listener”且取用listener接口参数的方法！像是：

addKeyListener(KeyListener k)

有这种method的class就是KeyEvent的来源。

## 回到图形上面……

我们已经大概知道事件的运行方式（后面还有更详细的说明），现在回到在屏幕的绘制上。在回到事件处理前，先来花一点点时间来玩一下图形。

### 在GUI上面加东西的3种方法：

#### ① 在frame上放置widget

加上按钮、窗体、radio button等。

```
frame.getContentPane().add(myButton);
```

javax.swing这个包上面有超过一打的widget类型。

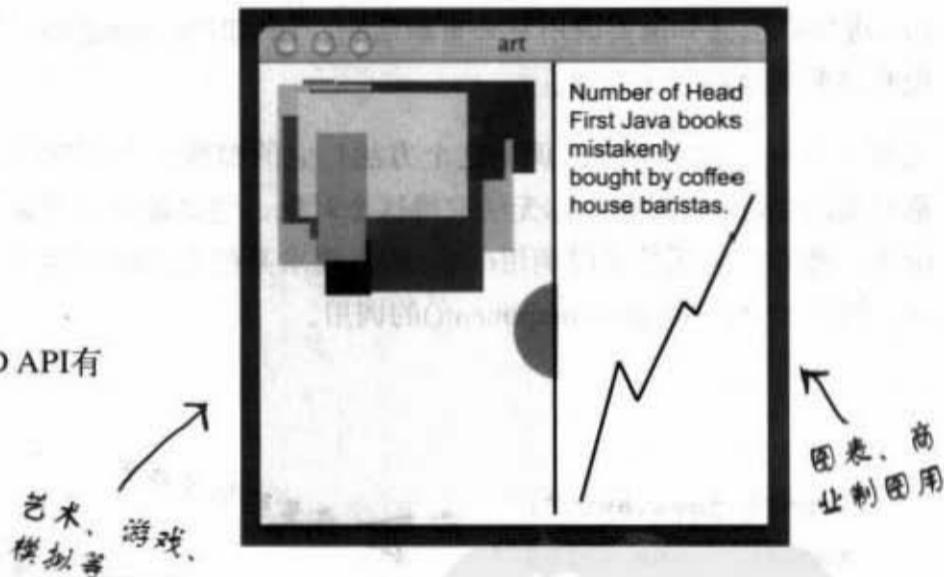


#### ② 在widget上绘制2D图形

使用graphics对象来绘制图形。

```
graphics.fillOval(70, 70, 100, 100);
```

你可以画上很多的方块和圆圈，Java2D API有很多好玩、复杂的图形方法。



#### ③ 在widget上绘制JPEG图

把图形画在widget上。

```
graphics.drawImage(myPic, 10, 10, this);
```



## 自己创建的绘图组件

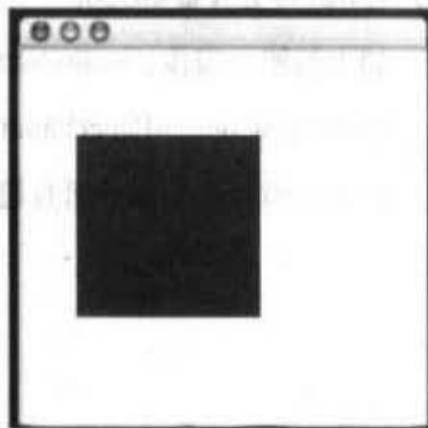
如果你要在屏幕上放上自己的图形，最好的方式是自己创建出有绘图功能的widget。你把widget放在frame上，如同按钮或其他的widget一样，不同之处在于它会按照你所要的方式绘制。你还可以让图形移动、表现动画效果或在点选的时候改变颜色。

简单得不得了。

创建JPanel的子类并覆盖掉paintComponent()这个方法。

所有绘图程序代码都在paintComponent()里面。把这个方法想象成会被系统告知要把自己画出来的方法。如果你要画的是圆圈，就写画圆圈的程序。当你的panel所处的frame显示的时候，paintComponent()就会被调用。如果用户缩小window或选择最小化，Java虚拟机也会知道要调用它来重新绘制。任何时候Java虚拟机发现有必要重绘都会这么做。

还有一件事，你不会自己调用这个方法！它的参数是个跟实际屏幕有关的Graphics对象，你无法取得这个对象，它必须由系统来交给你。然而，你还是可以调用repaint()来要求系统重新绘制显示装置，然后才会产生paintComponent()的调用。



```

import java.awt.*;
import javax.swing.*;
// 需要引用这些包

class MyDrawPanel extends JPanel {
 // 创建JPanel的子类

 public void paintComponent(Graphics g) {
 // 这是非常重要的方法，你决不能自己
 // 调用，要由系统来调用

 g.setColor(Color.orange);
 g.fillRect(20, 50, 100, 100);
 }
}
// 你可以把g想象成绘图装置，告诉它要用
// 什么颜色画出什么形状

```

## 在paintComponent()中可以做的事情

来看一下其他可以在paintComponent()中做的事情，其中最有趣的莫过于自己做些试验。试试看操弄一下数字，并查询Graphics这个类的API（稍后我们会看到更多的说明）。

### 显示JPEG

```
public void paintComponent(Graphics g) {
 Image image = new ImageIcon("catzilla.jpg").getImage();
 g.drawImage(image, 3, 4, this);
}
```

图文件名  
这个坐标代表图案左上角的位置离panel的左方边缘3个像素，离顶端4个像素

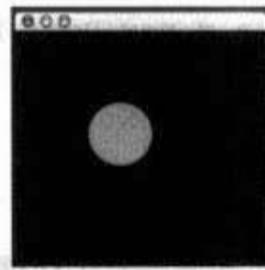


### 在黑色背景画上随机色彩的圆圈

```
public void paintComponent(Graphics g) {
 g.fillRect(0, 0, this.getWidth(), this.getHeight());
 int red = (int) (Math.random() * 255);
 int green = (int) (Math.random() * 255);
 int blue = (int) (Math.random() * 255);

 Color randomColor = new Color(red, green, blue);
 g.setColor(randomColor);
 g.fillOval(70, 70, 100, 100);
}
```

v. 默认颜色填充  
前两个参数是起点的坐标，后面两个参数分别是宽度和高度，此处取得本身的宽高，因此会把panel填满  
传入3个参数来代表RGB值



填满参数指定的椭圆型区域

## 在每个Graphics引用的后面都有个Graphics2D对象

paintComponent()的参数被声明为Graphics类型 (java.awt.Graphics)。

```
public void paintComponent(Graphics g) { }
```

因此参数g是个Graphics对象。这代表它可能是个Graphics的子类（因为多态的缘故），事实上就是这样。

由g参数所引用的对象实际上是个Graphics2D的实例。

为何要知道？因为有些在Graphics2D引用上可以做的事情不能在Graphics引用上做。Graphics2D对象可以做的事情比Graphics对象更多，实际上躲在Graphics引用的后面是个Graphics2D对象。

记得多态的问题，编译器会根据引用的类型而不是实际对象来判定你能够调用哪些方法。如果你有个Dog对象是由Animal引用变量来引用方法的：

```
Animal a = new Dog();
```

那你就不能让a吠：

```
a.bark();
```

就算你明知道那是个Dog也一样。但你还是可以把它转回成Dog：

```
Dog d = (Dog) a;
d.bark();
```

因此Graphics对象的底限是这样的：

如果你要调用Graphics2D类的方法，就不能直接使用g参数。但你可以将它转换成Graphics2D变量。

```
Graphics2D g2d = (Graphics2D) g;
```

可以对Graphics引用调用的方法：

```
drawImage()
drawLine()
drawPolygon
drawRect()
drawOval()
fillRect()
fillRoundRect()
setColor()
```

转换成Graphics2D对象：

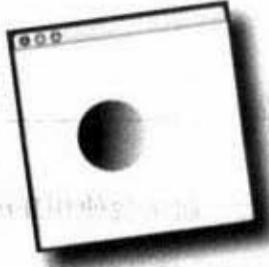
```
Graphics2D g2d = (Graphics2D) g;
```

可以对Graphics2D引用调用的方法：

```
fill3DRect()
draw3DRect()
rotate()
scale()
shear()
transform()
setRenderingHints()
```

（这不是完整的列表，查阅API文件可以取得最完整的说明）

纯色算不了什么，还有渐层  
颜色可以做出很棒（或很  
俗）的效果



实际上是个Graphics2D对象

↓

```
public void paintComponent(Graphics g) {
 Graphics2D g2d = (Graphics2D) g;
 ↗ 将它的类型转换成Graphics2D
 GradientPaint gradient = new GradientPaint(70, 70, Color.blue, 150, 150, Color.orange);
 ↑ ↑ ↗ ↗
 起点 开始的颜色 终点 渐层最后的颜色
 ↗ 将虚拟的“笔刷”换成渐层
 g2d.setPaint(gradient);
 g2d.fillOval(70, 70, 100, 100);
 ↗
 用目前的笔刷设定来填满椭
 圆型的区域
}
```

---

```
public void paintComponent(Graphics g) {
 Graphics2D g2d = (Graphics2D) g;
 int red = (int) (Math.random() * 255);
 int green = (int) (Math.random() * 255);
 int blue = (int) (Math.random() * 255);
 Color startColor = new Color(red, green, blue);
 red = (int) (Math.random() * 255);
 green = (int) (Math.random() * 255);
 blue = (int) (Math.random() * 255);
 Color endColor = new Color(red, green, blue);
 GradientPaint gradient = new GradientPaint(70, 70, startColor, 150, 150, endColor);
 g2d.setPaint(gradient);
 g2d.fillOval(70, 70, 100, 100);
}
```

这跟上面差不多，只是渐层颜色  
是随机挑选的

## 要点

## 事件

- GUI从创建window开始，通常会使用JFrame  

```
JFrame frame = new JFrame();
```
- 你可以这样加入按钮、文字字段等组件：  

```
frame.getContentPane().add(button);
```
- JFrame与其他组件不同，不能直接加上组件，要用它的content pane。
- 要显示window，你得指定尺寸和执行显示动作  

```
frame.setSize(300, 300);
frame.setVisible(true);
```
- 监听GUI事件才能知道用户对接口做了什么事情。
- 你必须要对事件源注册所要监听的事件。事件源是一种会根据用户操作而触发事件的机制。
- 监听接口让事件源能够调用给你。
- 要对事件源注册就调用事件源的注册方法，它的方法一定是add<EventType>Listener这种形式。以按钮的ActionEvent注册为例：  

```
button.addActionListener(this);
```
- 通过实现所有的事件处理方法来实现监听接口。对ActionEvent而言，方法可能像这样：  

```
public void actionPerformed(ActionEvent
 event) {
 button.setText("Clicked!");
}
```
- 传递给事件处理方法的事件对象带有事件的信息，其中包括了事件源。

## 图形

- 二维图形可以直接画在图形组件上。
- .gif与.jpeg文件可以直接放在组件上。
- 用 JPanel的子类覆盖paintComponent()方法绘制自定义的图形。
- paintComponent()方法会由GUI系统调用，你不可以自己调用。它的参数是个你不能自己创建的Graphics对象。
- Graphics对象有些你可以调用的方法，像是：  

```
graphics.setColor(Color.blue);
g.fillRect(20, 50, 100, 120);
```
- 使用Image来绘制.jpg：  

```
Image image = new ImageIcon("pic.
jpg").getImage();
g.drawImage(image, 3, 4, this);
```
- paintComponent()的Graphics参数实际上是个Graphics2D。
- 调用Graphics2D的方法前，你必须把Graphics对象转换为Graphics2D。  

```
Graphics2D g2d = (Graphics2D) g;
```

我们可以获得事件，也可以绘制图形。

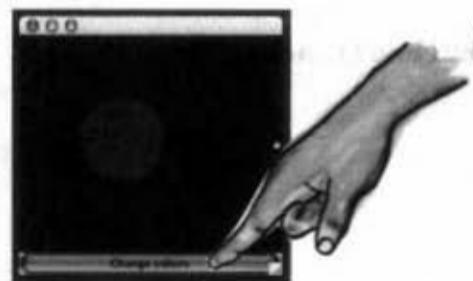
但可以在获得事件时绘制图形吗？

现在让我们试试看在事件发生时改变面板的图案。让圆圈在用户按下按钮时改变颜色。下面列出程序的流程：

#### 启动程序



- 1 这是个运用到panel和button组件的frame。将监听向按钮注册，然后显示frame并等待用户点击。



- 2 用户点击按钮，因此创建出一个事件对象并调用监听的事件处理程序。



- 3 事件处理过程调用frame的repaint()，然后系统会调用panel的paintComponent()。

- 4 好了！因为paintComponent()又运行了一次，所以圆圈被填上不同的颜色。



一个frame上面怎么可以  
摆两个widget吗？

## GUI的布局：超过一个以上 widget的frame

下一章会讨论GUI的布局（layout），但我们现在可以先快速地看一遍。frame默认有5个区域可以安置widget。每个区域只能安置一项，但是别担心！该项目可以是能够安置包括面板在内的3项东西的面板，所以你可以在面板上面放面板。事实上，我们在安置按钮的时候就作弊了：

```
frame.getContentPane().add(button);
```

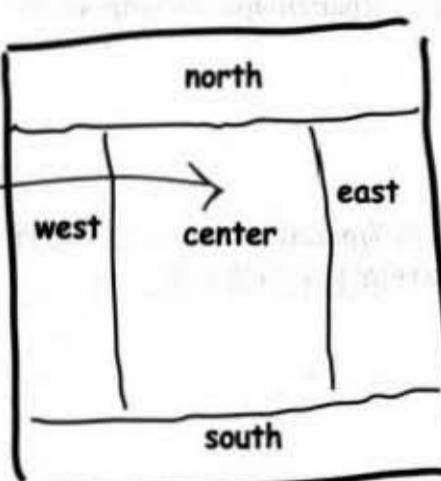
其实你不应该这么做

明确指定加到默认content  
pane上的位置是比较好的方  
式

如果使用的是单一参数的  
add()方法会自动地把widget加  
到中心区域

```
frame.getContentPane().add(BorderLayout.CENTER, button);
```

两个参数的add()方法可  
以指定使用的区域



Sharpen your pencil

写出一个程序可以像369页那样把  
按钮和面板加到frame上。

## 按下按钮圆圈就会改变颜色

自定义的面板是放在  
frame的中心区域



按钮安置于  
SOUTH区域

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleGui3C implements ActionListener {
 JFrame frame;

 public static void main (String[] args) {
 SimpleGui3C gui = new SimpleGui3C();
 gui.go();
 }

 public void go() {
 frame = new JFrame();
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 JButton button = new JButton("Change colors");
 button.addActionListener(this); ← 把监听加到按钮上

 MyDrawPanel drawPanel = new MyDrawPanel();

 frame.getContentPane().add(BorderLayout.SOUTH, button); ← 依照指定区域把
 frame.getContentPane().add(BorderLayout.CENTER, drawPanel); ← widget放上去
 frame.setSize(300,300);
 frame.setVisible(true);
 }

 public void actionPerformed(ActionEvent event) {
 frame.repaint();
 }
}

```

当用户按下按钮时就要求frame重  
新绘制

```

class MyDrawPanel extends JPanel {
 public void paintComponent(Graphics g) {
 // 填入彩色, 见367页
 }
}

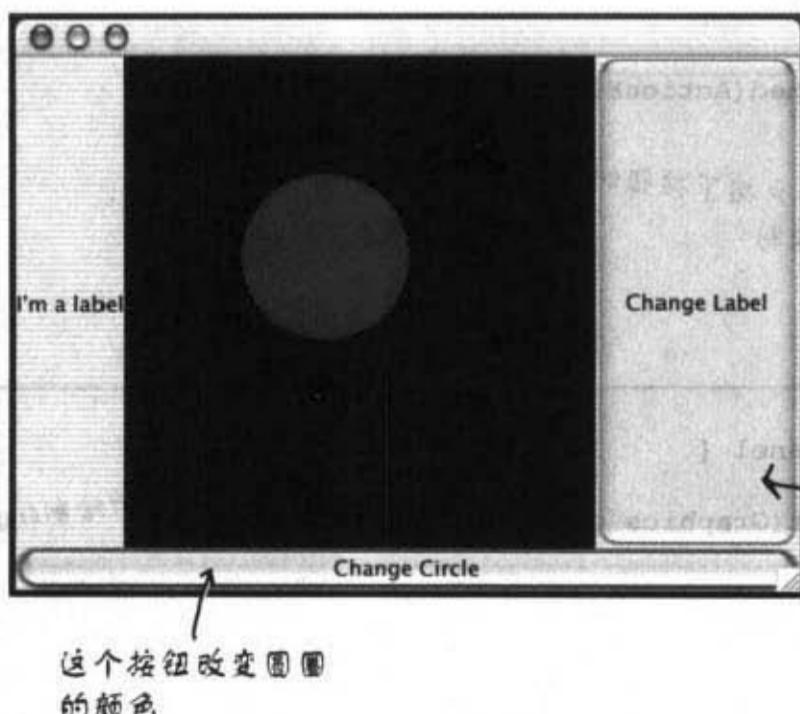
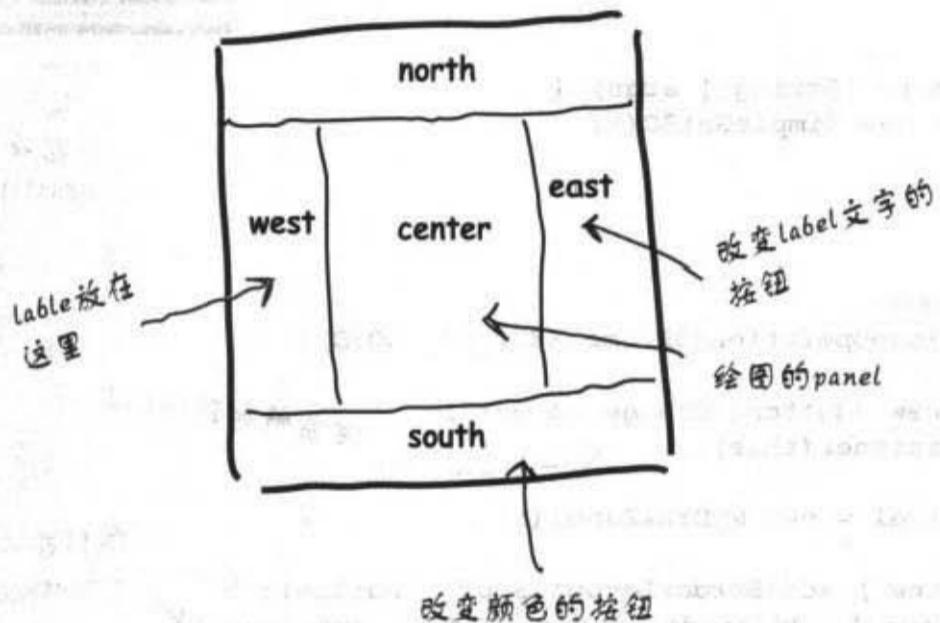
```

这个方法会在重新绘制frame时  
候被调用

## 尝试两个按钮

South的按钮没有改变，还会要求frame重新绘制。第二个按钮（贴在east处）会改变label上面的文字（label是一种显示文字的widget）。

所以现在有4个widget



还得要感知两个事件

只有一个actionPerformed()方法的时候可以这么做吗？

当每个按钮执行不同工作时，要如何对两个不同的按钮分别取得事件？

### ● 选项一：实现两个actionPerformed()方法。

```
class MyGui implements ActionListener {
 // 一堆程序代码

 public void actionPerformed(ActionEvent event) {
 frame.repaint();
 }

 public void actionPerformed(ActionEvent event) {
 label.setText("That hurt!");
 }
}
```

不可以这样！

不能这么做！你不能在实现同一个类的同一个方法两次，这过不了编译这一关。就算可以，事件源怎么分得出要调用哪一个？

### ● 选项二：对两个按钮注册同一个监听口。

```
class MyGui implements ActionListener {
 // 声明一组实例变量

 public void go() {
 // 创建GUI
 colorButton = new JButton();
 labelButton = new JButton();
 colorButton.addActionListener(this); // 注册同一个监听口
 labelButton.addActionListener(this); // 还有一些GUI程序……
 }

 public void actionPerformed(ActionEvent event) {
 if (event.getSource() == colorButton) {
 frame.repaint();
 } else {
 label.setText("That hurt!");
 }
 }
}
```

查询事件对象来看实际上事件是哪个事件源发出的

可以是可以啦，但这看起来不太像面向对象。用单一的事件处理程序对付不同的东西意味着执行太多不同工作的方法。如果想要改变某个工作，很可能把全部工作都弄乱。这样解决会对可读性和维护工作产生危害。

当每个按钮执行不同工作时，要如何对两个不同的按钮分别取得事件？

● 选项三：创建不同的ActionListener。

```
class MyGui {
 JFrame frame;
 JLabel label;
 void gui() {
 // 实际程序代码
 }
} // 关闭类
```

```
class ColorButtonListener implements ActionListener {
 public void actionPerformed(ActionEvent event) {
 frame.repaint();
 }
}
```

不行！这个类没有对MyGui上frame变量的引用

```
class LabelButtonListener implements ActionListener {
 public void actionPerformed(ActionEvent event) {
 label.setText("That hurt!");
 }
}
```

有问题！此类没有label的引用变量

这些类没有办法存取到所需的变量。你可以加以改正，但必须要给予每个监听类对GUI类的引用，才能让actionPerformed()方法中的监听能够使用类的引用存取它的变量。这却又会破坏封装的特性，因此我们或许得需要更好的getter函数（例如getFrame()或getLabel()等），并且你或许也需要对监听类加上一个构造函数以便能够在监听初始化的同时传入GUI的引用。不过这样只会加深混乱和复杂的程度。

一定有比较好的方法吧？



内部类万岁！

## 内部类是我们的救星！

一个类可以嵌套在另一个类的内部。这很简单，只要确定内部类的定义是包在外部类的括号中就可以。

单纯的内部类：

```
class MyOuterClass {
 class MyInnerClass {
 void go() {
 }
 }
}
```

内部类完全被外部的  
类包起来

内部类可以使用外  
部所有的方法与变  
量，就算是私用的  
也一样。

内部类把存取外部  
类的方法和变量当  
作是开自家冰箱。

内部的类对外部的类有一张特殊的通行证，能够自由地存取它的内容，就算是私用的内容也一样。内部类可以把外部的方法或变量当作是自己的。这就是为何内部的类非常好用的原因，除了跟正常的类没有差别之外，还多了特殊的存取权。

### 内部类可以使用外部的变量

```
class MyOuterClass {
 private int x;

 class MyInnerClass {
 void go() {
 x = 42; ← 把x当作自己的!
 }
 } // 关闭内部类
} // 关闭外部类
```

## 内部类的实例一定会绑在外部类的实例上\*

要记住，当我们讨论内部类可以存取外部类的内容时，意思是说内部类的实例可以存取外部类实例的内容。但是哪个实例呢？

任意一个内部类可以存取其他外部类的方法和变量吗？不行！只能存取它所属的那个！



内部对象与外部对象  
有发生过超友谊的关  
系



### ① 创建外部类的实例

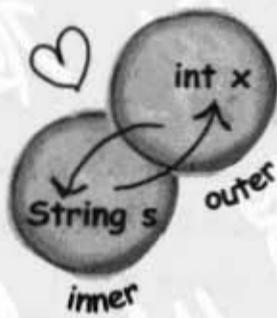


### ② 使用外部类的实例来 创建内部类的实例



### ③ 外部和内部对象有着亲密的连 接。

这两个对象在堆上有特  
殊的关系，内外可以交  
互使用变量



\*有一种非常特殊的异常情况——内部类是定义在静态的方法中。本书不打算讨论这个，你也可能一辈子都不会遇到。

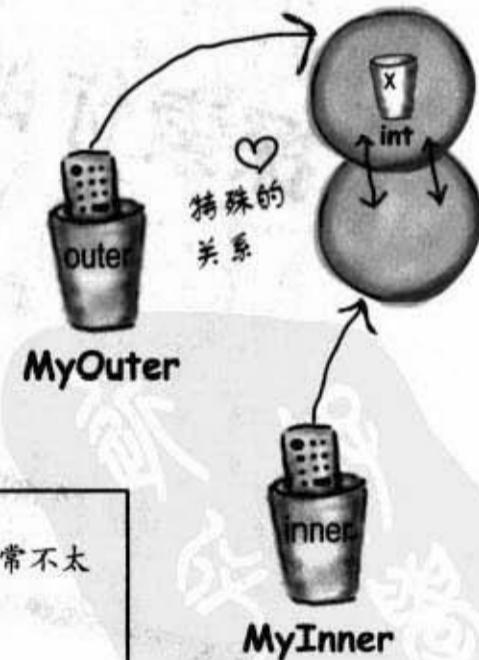
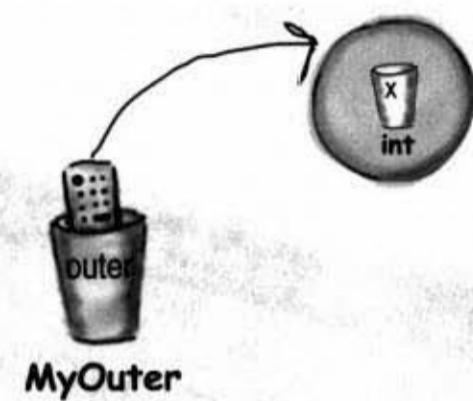
## 如何创建内部类的实例

如果你从外部类程序代码中初始化内部的类，此内部对象会绑在该外部对象上。例如，如果某个方法的程序代码会初始化内部的类，此内部对象会绑在执行该方法的实例上。

外层类的程序代码可以用初始化其他类完全相同的方法初始它所包容的内部类。

```
class MyOuter {
 private int x; // 外部有个私用的x实例变量
 MyInner inner = new MyInner(); // 创建内部的实例
 public void doStuff() {
 inner.go(); // 调用内部的方法
 }

class MyInner {
 void go() {
 x = 42; // 内部可以使用外部的x变量
 } // 关闭内部类
} // 关闭外部类
```



### 附注

你也可以从外部类以外的程序代码来初始内部实例，但这要使用特殊的语法。通常不太会有机会要这么做，但还是先让你知道一下。

```
class Foo {
 public static void main (String[] args) {
 MyOuter outerObj = new MyOuter();
 MyOuter.MyInner innerObj = outerObj.new MyInner();
 }
}
```

## 现在就可以实现两个按钮的程序

```

public class TwoButtons { ← 现在主要的GUI类并不实现
 ActionListener

 JFrame frame;
 JLabel label;

 public static void main (String[] args) {
 TwoButtons gui = new TwoButtons ();
 gui.go ();
 }

 public void go() {
 frame = new JFrame ();
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 JButton labelButton = new JButton("Change Label");
 labelButton.addActionListener(new LabelListener ());

 JButton colorButton = new JButton("Change Circle");
 colorButton.addActionListener(new ColorListener ());

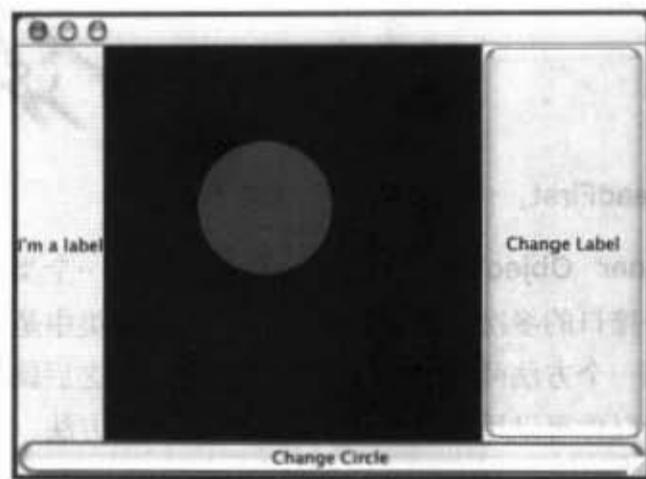
 label = new JLabel("I'm a label");
 MyDrawPanel drawPanel = new MyDrawPanel ();

 frame.getContentPane().add(BorderLayout.SOUTH, colorButton);
 frame.getContentPane().add(BorderLayout.CENTER, drawPanel);
 frame.getContentPane().add(BorderLayout.EAST, labelButton);
 frame.getContentPane().add(BorderLayout.WEST, label);

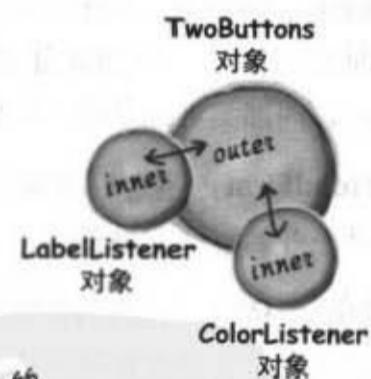
 frame.setSize(300,300);
 frame.setVisible(true);
 }

 class LabelListener implements ActionListener {
 public void actionPerformed(ActionEvent event) {
 label.setText("Ouch!");
 }
 } // 关闭内部类
 class ColorListener implements ActionListener {
 public void actionPerformed(ActionEvent event) {
 frame.repaint();
 }
 } // 关闭内部类
}

```



相对于将this传给监听的注册方法，现在传的是对应的实例



终于可以在单一的类中做出不同的 ActionListener!

内部可以存取 label

直接存取frame，不需要明确指定外类的引用



## 本周的来宾：内部类的实例

**HeadFirst:** 内部类有什么重要？

**Inner Object:** 怎么说呢，我们提供在一个类中实现同一接口的多次机会。要知道，在一般的类中是不能实现同一个方法两次的。但使用了内部的类之后就可以了，所以你可以用不同方法实现同一个接口方法。

**HeadFirst:** 为什么要实现同一个方法两次？

**Inner Object:** 回想一下GUI事件处理程序。如果你想要让3个按钮有不同的事件行为，就要使用3个内层类来个别实现ActionListener，也就是每个类实现自己的actionPerformed()方法。

**HeadFirst:** 所以说事件处理程序是唯一的理由了？

**Inner Object:** 当然不是。这只是个明显的例子。任何时候你需要一个独立却又好像另一个类成员之一的类时，内部类可能是唯一的解。

**HeadFirst:** 还是搞不懂，如果需要独立的类，为什么不一开始就独立地创建？

**Inner Object:** 因为要实现同一个接口好几次。就算不是这样，你也会需要两个不同的类来表示两项不同的事物，这样才是好的面向对象。

**HeadFirst:** 哇！我以为面向对象代表重用与维护呢。两个不同的类可以分开维护，但包在一起不就纠缠住了吗？并且被包起来的类不是就不能重用吗？

**Inner Object:** 没错，被包起来的内部类无法像独立的类一样重用，因为它会与外部的类紧密地结合。但是……

**HeadFirst:** 这就是我说的，如果不能重用又何必这样呢？我是说这根本就是为了解决接口的错误而产生的啊。

**Inner Object:** 就像我已经说过的，你要用IS-A和多态的观点来看这件事。

**HeadFirst:** 可以，这是因为……

**Inner Object:** 因为外部与内部的类需要通过不同的IS-A测试！以GUI的监听为例，按钮的监听声明要注册什么？也就是说要传给addActionListener()什么东西？

**HeadFirst:** 这种情况下要传进一个ActionListener。你的重点是什么？

**Inner Object:** 重点在于polymorphically，有个方法只能采用特定的类型。有时这可以通过ActionListener的IS-A测试，但最重要的是，如果你的类必须IS-A别的类呢？

**HeadFirst:** 为什么不让你的类去继承该类呢？

**Inner Object:** 如果它本来就已经继承不相干的类呢？

**HeadFirst:** 噢，我明白了，接口的实现可以超过一个，但类仅能继承一个而已。

**Inner Object:** 很好！没错，你不能同时又是Dog又是按钮，但有时又必须这样。Dog可以继承Animal却有个内部的类来代表按钮的行为，因此在有需要的时候Dog就可以派出内部的类来代表按钮。也就是说Dog虽然不能x.takeButton(this)但是可以x.takeButton(new DogInnerButton())。

**HeadFirst:** 可不可以再举个更清楚的例子？

**Inner Object:** 还记得 JPanel 的子类吗？现在我们还把它写成独立的类。这还好，因为此类不需要对 GUI 的实例变量有特殊的存取权。但若是要呢？如果我们要让这个面板存取主程序的坐标变量来执行动画呢？在这种情况下，我们可以让绘图的面板成为内部的类，且是 JPanel 的子类，而外部类就可以自由地去当别的类的子类。

**HeadFirst:** 我懂了！还有，绘图的面板也不是那么可以独立的重用，因为它实际上只是写给特定的 GUI 程序用。

**Inner Object:** 很好，这块饼干给你吃。

**HeadFirst:** 接下来我们可以继续踢爆你与外部实体间的亲密关系。

**Inner Object:** 你们这些人是怎么了？有线电视的新闻台看太多了是不是？

**HeadFirst:** 唉呦，你又不是不知道观众最喜欢听八卦。所以有人把你创建出来之后你就算是它“包养”的是吧？

**Inner Object:** 这倒是真的，有人把这当作是一种婚约。

**HeadFirst:** 就说是婚约吧。你们能够离婚再嫁吗？

**Inner Object:** 不行，这是一辈子的事情。

**HeadFirst:** 谁的一辈子？外部吗？

**Inner Object:** 我自己的。我不能绑在其他外部对象上。唯一的解脱只有 garbage collection。

**HeadFirst:** 那外部对象呢？它可以绑其他的内部对象吗？

**Inner Object:** 它可以。而且是同时间进行的，香港人叫它“一王多后”，满意了吧？

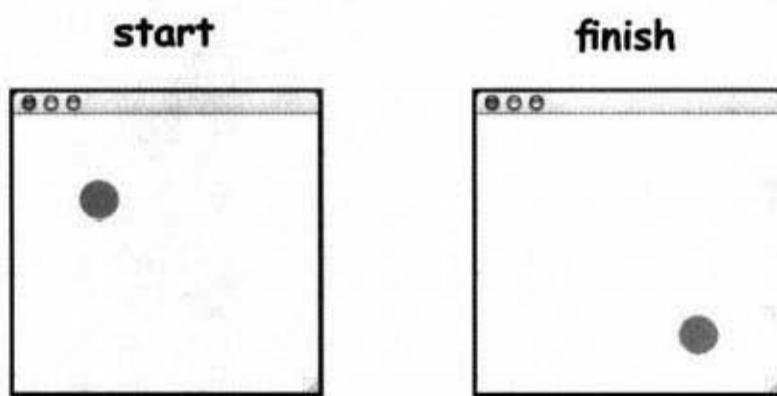
**HeadFirst:** 是你自己先开始多重实现的话题啊。所以外部类同时有多个按钮要多个内部类来照顾事件是很合理的。谢谢，今天的访谈到此结束。



## 以内部类执行动画效果

我们已经看过为何内部类对事件的监听是很方便的，因为你会对相同的事件处理程序实现一次以上。现在让我们来看看当内部类被用来当作某种外部类无法继承的子类时是多么好用。换句话说，内外部的类可以搞定不同的继承层次！

我们的目标是要创建出简单的动画，让圆圈从画面左上方移动到右下方。



动画效果是如何运动的：

- ① 在特定坐标点绘制对象。

```
g.fillOval(20, 50, 100, 100);
 ↑
 离左上方 20, 50 个
 像素
```

- ② 在不同的坐标点重新绘制对象。

```
g.fillOval(25, 55, 100, 100);
 ↑
 离左上方 25, 55 个像素
 (稍微向右下方移动)
```

- ③ 在坐标尚未到达终点前重复上列步骤。

there are no  
Dumb Questions

问：为什么要学动画？  
我又不想开发游戏。

答：你也许不会去开发游戏，但可能会碰到仿真器，它也会持续地显示处理运算的结果或者你会需要创建出持续更新的图形来显示内存的耗用状况。这一类的事情都会遇到类似的方法。

其实说穿了这只是个展示内部类功用最简单的方式。

其实我们真正需要的是这样……

```
class MyDrawPanel extends JPanel {
 public void paintComponent(Graphics g) {
 g.setColor(Color.orange);
 g.fillOval(x,y,100,100);
 }
}
```

每次paintComponent()被调用时，把圆形画在不同的位置上



但是要如何取得新的坐标呢？

又是谁要来调用repaint()？

你是否能够自己设计出一个简单的解决方案让图形从画面的左上方移动到右下方？答案在下一页，自己还没有想过之前不要偷看！

提示一：把绘图的面板当作是内部的类。

提示二：别在paintComponent()里面放任何种类的循环。

把答案写下来：

## 完整的动画程序

```

import javax.swing.*;
import java.awt.*;

public class SimpleAnimation {
 int x = 70; } ← 在主要的GUI中创建两个实例
 int y = 70; } ← 变量用来代表图形的坐标

 public static void main (String[] args) {
 SimpleAnimation gui = new SimpleAnimation ();
 gui.go ();
 }

 public void go () {
 JFrame frame = new JFrame ();
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 MyDrawPanel drawPanel = new MyDrawPanel (); ← 此处跟前面一样创建出frame上的
 frame.getContentPane ().add (drawPanel); widget
 frame.setSize (300, 300);
 frame.setVisible (true);

 这里是重点！
 for (int i = 0; i < 130; i++) { ← 重复130次
 x++;
 y++; ← 递增坐标值
 drawPanel.repaint (); ← 要求重新绘制面板

 try {
 Thread.sleep (50); ← 加上延迟刻意放慢，不然一下就会
 } catch (Exception ex) { } 跑完，第15章会说明线程
 }

 // 关闭go () 方法
}

内部类
class MyDrawPanel extends JPanel {

 public void paintComponent (Graphics g) {
 g.setColor (Color.green);
 g.fillOval (x, y, 40, 40); ← 使用外部的坐标值来更新
 }
} // 关闭内部类
} // 关闭外部类

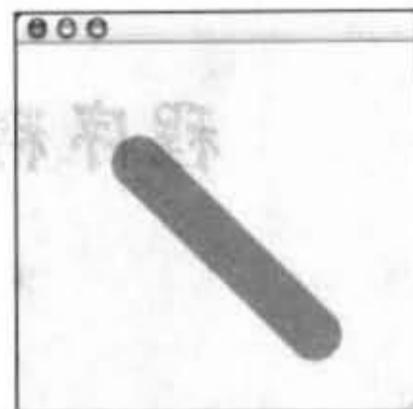
```

## 哎呀！它留下了痕迹

哪里弄错了？这个程序有个 bug。

## 我们忘记擦掉原来的图形， 所以会有痕迹

解决的办法是在每次画上新的圆圈之前把整个面板用原来的背景底色填满。下面的程序代码在方法的前面加上两行指令：先把颜色设定为白色，然后填满整个方块区域。也就是说从0,0位置开始以白色填入面板长宽大小的区域。

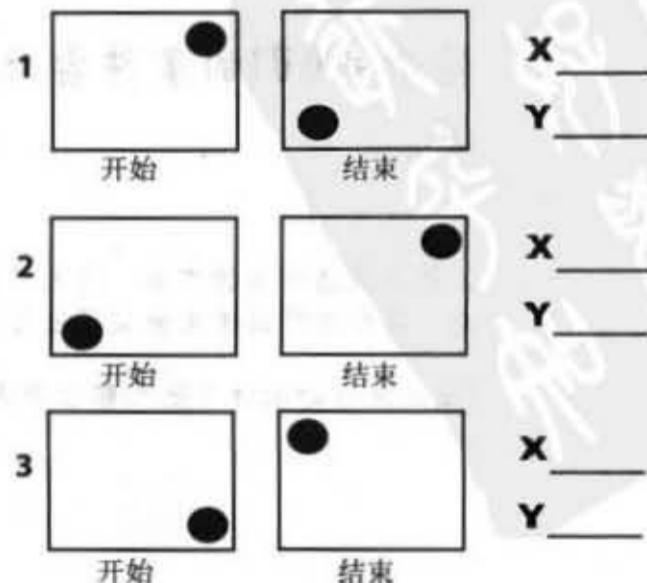
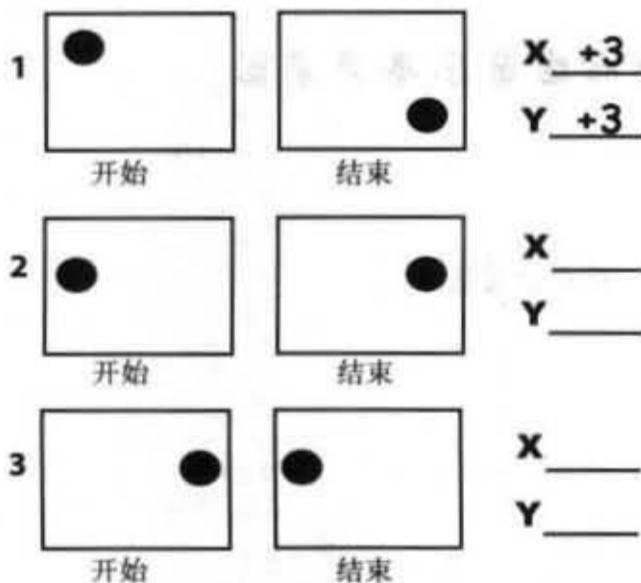


```
public void paintComponent(Graphics g) {
 g.setColor(Color.white);
 g.fillRect(0,0,this.getWidth(), this.getHeight());
 g.setColor(Color.green);
 g.fillOval(x,y,40,40); ↑
 ↑
 从 JPanel 继承下来的方法
}
```

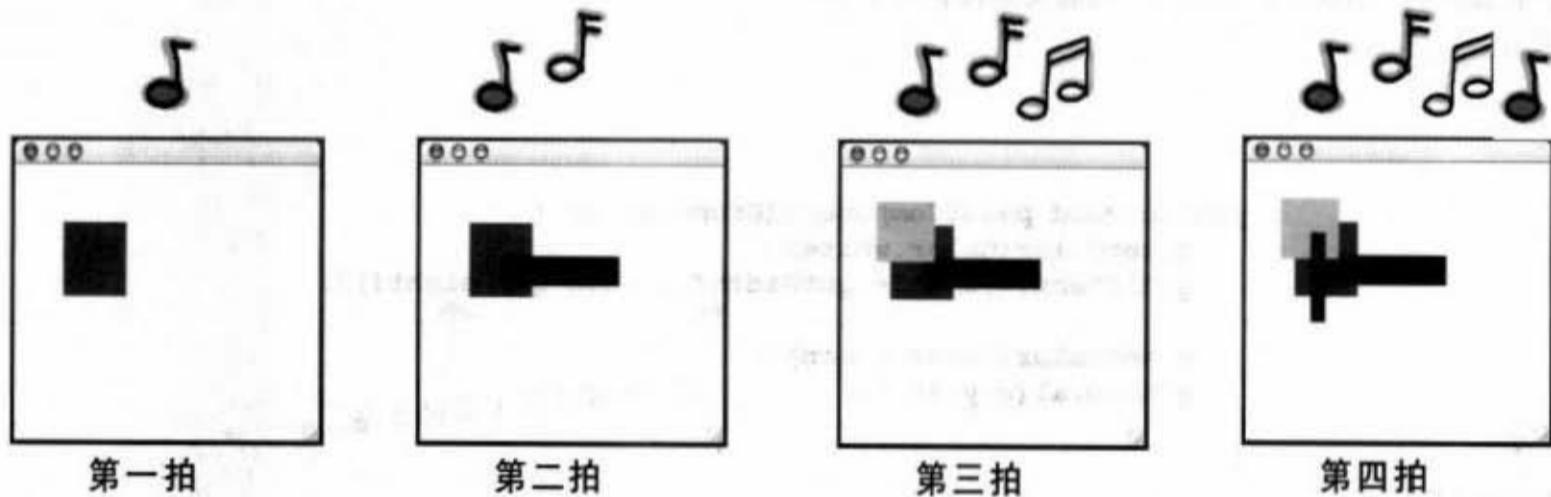


### Sharpen your pencil (有空的时候挑战看看) —————

你可以怎样修改坐标的递增值来产生下面所示不同的效果？



## 程序料理



第一拍

第二拍

第三拍

第四拍

让我们来制作音乐录像带！使用Java随机产生的圆形来跟着节奏起舞。

这个非GUI的事件会向事件源注册而由音乐本身来触发。

这部分是选择性的内容。但我们认为很值得一看。你应该会很喜欢，并且也可以拿来向父母女友炫耀一番……

(反正不管你做什么他们都会夸奖的)

## 监听非GUI的事件

对啦，这比不上音乐录像带，但我们还是会做出一个随着音乐节奏绘制随机图形的程序。简单地说，这个程序会监听音乐节奏并在每个拍子上画出随机的方块图形。

这会带来新的问题。目前为止我们只监听过 GUI 的事件，但现在则需要监听特定类型的 MIDI 事件。监听非GUI事件的最后结果就跟监听 GUI 事件是一样的：你会实现出监听者的接口，向事件源注册，然后等待事件源调用你的事件处理程序（定义在监听者的接口中的方法）。

监听音乐节奏的最简单方式是注册并监听实际的 MIDI 事件，因此只要 sequencer 收到事件，我们的程序也会取得并绘制图形。但是……有个问题。实际上有个 bug 会让我们无法监听我们自己制造的 MIDI 事件（NOTE ON）。

所以我们得做一点小小的修正。我们可以监听另外一种类型的 MIDI 事件，它被称为 ControllerEvent。我们的解决方案是注册 ControllerEvent，然后确保每个 NOTE ON 事件都有对应的 ControllerEvent 事件会在同一拍上面触发。要怎样确保这件事呢？如同其他事件一样把它加到 track 上！也就是说，我们的 sequence 会像下面这样：

BEAT 1 - NOTE ON, CONTROLLER EVENT

BEAT 2 - NOTE OFF

BEAT 3 - NOTE ON, CONTROLLER EVENT

BEAT 4 - NOTE OFF……

如此继续下去。

这个音乐艺术程序需要下列功能：

- 制作一系列的MIDI信息/事件来播放任意的钢琴音（或是你自行设定的其他乐器）。
- 对事件注册一个监听者。
- 开始sequencer的播放操作。
- 每当监听者的事件处理程序被调用时，在面板上面画一个随机的方块并调用repaint。

制作程序的3个方式：

- 第一版：简单地制作出MIDI事件，因为要做出很多个。
- 第二版：注册并监听事件，但没有图形。从命令栏对每一拍输出一个信息。
- 第三版：最终版本。在第二版上加上图形输出。

## 制作信息/事件的简单 (偷懒) 方法

制作出信息和事件并把它们加到track上是一件很枯燥的工作。对每个信息都得做出信息的实例(ShortMessage)、调用setMessage()、制作MidiEvent，然后将事件加到track上。上一章的程序代码中我们对每个信息逐步执行这些操作。如此需要8行程序才能做出一个信息！4行做NOTE ON事件、再4行做NOTE OFF事件。

```
ShortMessage a = new ShortMessage();
a.setMessage(144, 1, note, 100);
MidiEvent noteOn = new MidiEvent(a, 1);
track.add(noteOn);

ShortMessage b = new ShortMessage();
b.setMessage(128, 1, note, 100);
MidiEvent noteOff = new MidiEvent(b, 16);
track.add(noteOff);
```

### 每个事件都要有的操作：

#### ● 创建信息实例。

```
ShortMessage first = new ShortMessage();
```

#### ● 调用setMessage()。

```
first.setMessage(192, 1, instrument, 0)
```

#### ● 制作信息的MidiEvent实例。

```
MidiEvent noteOn = new MidiEvent(first, 1);
```

#### ● 把事件加到track上。

```
track.add(noteOn);
```

## 创建静态的实用方法来制作信息并返回 MidiEvent

```
public static MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
 MidiEvent event = null;
 try {
 ShortMessage a = new ShortMessage();
 a.setMessage(comd, chan, one, two);
 event = new MidiEvent(a, tick);
 } catch (Exception e) { }
 return event;
}
```

这4个参数是给信息用的

哇！5个参数！

tick是何时播放信息

使用参数来创建信息

返回事件

## 范例：如何使用静态的makeEvent()方法

此处没有事件处理或绘图，只有15个攀升的音阶组成的队列。这个程序代码的重点是学习如何使用makeEvent()方法。下两版的程序代码会比较简单也比较简单。

```

import javax.sound.midi.*; ← 别忘记要import
public class MiniMusicPlayer1 {

 public static void main(String[] args) {
 try {
 Sequencer sequencer = MidiSystem.getSequencer(); ← ④) 建立并打开队列
 sequencer.open();

 Sequence seq = new Sequence(Sequence.PPQ, 4); ← ④) 创建队列并track
 Track track = seq.createTrack(); ← ④) 创建队列并track

 for (int i = 5; i < 61; i+= 4) { ← ④) 创建一堆连续的音符事件
 track.add(makeEvent(144,1,i,100,i));
 track.add(makeEvent(128,1,i,100,i + 2)); ← 调用makeEvent()来产生信息和事件
 } // 结束循环
 sequencer.setSequence(seq); } // 开始播放
 sequencer.setTempoInBPM(220);
 sequencer.start();
 } catch (Exception ex) {ex.printStackTrace();}
 } // 关闭main

 public static MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
 MidiEvent event = null;
 try {
 ShortMessage a = new ShortMessage();
 a.setMessage(comd, chan, one, two);
 event = new MidiEvent(a, tick);

 } catch (Exception e) { }
 return event;
 }
} // 关闭类

```

Controller的事件

## 第二版：注册并取得 ControllerEvent方法

```
import javax.sound.midi.*;
public class MiniMusicPlayer2 implements ControllerEventListener {
 public static void main(String[] args) {
 MiniMusicPlayer2 mini = new MiniMusicPlayer2();
 mini.go();
 }
 public void go() {
 try {
 Sequencer sequencer = MidiSystem.getSequencer();
 sequencer.open();
 int[] eventsIWant = {127};
 sequencer.addControllerEventListener(this, eventsIWant);
 Sequence seq = new Sequence(Sequence.PPQ, 4);
 Track track = seq.createTrack();
 for (int i = 5; i < 60; i += 4) {
 track.add(makeEvent(144, 1, i, 100, i));
 track.add(makeEvent(176, 1, 127, 0, i));
 track.add(makeEvent(128, 1, i, 100, i + 2));
 } // 结束循环
 sequencer.setSequence(seq);
 sequencer.setTempoInBPM(220);
 sequencer.start();
 } catch (Exception ex) { ex.printStackTrace(); }
 } // 关闭
 public void controlChange(ShortMessage event) {
 System.out.println("la");
 }
 public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
 MidiEvent event = null;
 try {
 ShortMessage a = new ShortMessage();
 a.setMessage(comd, chan, one, two);
 event = new MidiEvent(a, tick);
 } catch (Exception e) { }
 return event;
 }
} // 关闭类
```

← 我们必须要监听ControllerEvent，因此实现了这个接口

向sequencer注册事件。注册的方法服用监听者与代表想要监听的事件的int数组，我们只需要127事件的int数组。

插入事件编号为127的自定义ControllerEvent (176)，它不会做任何事情，只是让我们知道有音符被播放，因为它的tick跟NOTE ON是同时进行的

感知事件时在命令打印出字符串的事件处理程序

程序代码与上一版不同之处用底色强调出来

## 第三版：与音乐同步输出图形

最终版本是用第二版加上GUI部分而成的。我们创建出frame，加上绘图的面板，并在取得事件的同时画出新的方块并要求重绘画面。另外改变的地方是从连续攀升变成随机产生的音符。

除了简单的GUI之外，最重要的程序变化在于让绘图面板实现ControllerEventListener而不是由程序本身来实现。因此当内部类所做的绘图板获知事件时，它会知道该做些什么事。

完整的程序代码列在下一页。

### 绘图面板的内部类

```

class MyDrawPanel extends JPanel implements ControllerEventListener {
 boolean msg = false; ← 获知事件时才会为真

 public void controlChange(ShortMessage event) {
 msg = true; ←
 repaint(); ← 获知事件时设为真并调用重绘程序
 }

 public void paintComponent(Graphics g) {
 if (msg) { ← 因为也有其他东西会引发重绘，所以要判断是否为
 ControllerEvent所引发的
 Graphics2D g2 = (Graphics2D) g;
 int r = (int) (Math.random() * 250);
 int gr = (int) (Math.random() * 250);
 int b = (int) (Math.random() * 250); ← 其余的程序代码是在产生随机的颜色并画出方块
 g.setColor(new Color(r, gr, b));
 int ht = (int) ((Math.random() * 120) + 10);
 int width = (int) ((Math.random() * 120) + 10);
 int x = (int) ((Math.random() * 40) + 10);
 int y = (int) ((Math.random() * 40) + 10);
 g.fillRect(x, y, width, ht);
 msg = false;
 } // if结束
 } // 关闭方法
} // 关闭内部类

```



```

import javax.sound.midi.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;

public class MiniMusicPlayer3 {

 static JFrame f = new JFrame("My First Music Video");
 static MyDrawPanel ml;

 public static void main(String[] args) {
 MiniMusicPlayer3 mini = new MiniMusicPlayer3();
 mini.go();
 } // 关闭方法

 public void setUpGui() {
 ml = new MyDrawPanel();
 f.setContentPane(ml);
 f.setBounds(30,30, 300,300);
 f.setVisible(true);
 } // 关闭方法

 public void go() {
 setUpGui();

 try {

 Sequencer sequencer = MidiSystem.getSequencer();
 sequencer.open();
 sequencer.addControllerEventListener(ml, new int[] {127});
 Sequence seq = new Sequence(Sequence.PPQ, 4);
 Track track = seq.createTrack();

 int r = 0;
 for (int i = 0; i < 60; i+= 4) {

 r = (int) ((Math.random() * 50) + 1);
 track.add(makeEvent(144,1,r,100,i));
 track.add(makeEvent(176,1,127,0,i));
 track.add(makeEvent(128,1,r,100,i + 2));
 } // 结束循环

 sequencer.setSequence(seq);
 sequencer.setTempoInBPM(120);
 sequencer.start();
 } catch (Exception ex) {ex.printStackTrace();}
 } // 关闭方法
}

```

这是第三版程序代码的完整列表。它是直接使用第二版来改造的。试着不偷看前面的内容来自己加上注释，偷看最可耻。

```

public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
 MidiEvent event = null;
 try {
 ShortMessage a = new ShortMessage();
 a.setMessage(comd, chan, one, two);
 event = new MidiEvent(a, tick);
 } catch (Exception e) { }
 return event;
} // 关闭方法

class MyDrawPanel extends JPanel implements ControllerEventListener {
 boolean msg = false;

 public void controlChange(ShortMessage event) {
 msg = true;
 repaint();
 }

 public void paintComponent(Graphics g) {
 if (msg) {
 Graphics2D g2 = (Graphics2D) g;
 int r = (int) (Math.random() * 250);
 int gr = (int) (Math.random() * 250);
 int b = (int) (Math.random() * 250);
 g.setColor(new Color(r, gr, b));
 int ht = (int) ((Math.random() * 120) + 10);
 int width = (int) ((Math.random() * 120) + 10);
 int x = (int) ((Math.random() * 40) + 10);
 int y = (int) ((Math.random() * 40) + 10);
 g.fillRect(x, y, ht, width);
 msg = false;
 } // close if
 } // 关闭方法
} // 关闭内部类
} // 关闭类

```



一组Java组件精心打扮出席化装舞会，中场时间有人提议要玩猜猜我是谁的游戏，你可以根据它们对自己的描述来猜测出提示的是哪位。规则是每个组件都得说实话，若某些提示同时对数个组件都为真的话，则将它们全部填入。

今晚出席舞会的有：

这一章提到过的每个家伙都有可能出现！

整个GUI都掌握在我的手上。

每个事件类型都有一个。

监听者的关键方法。

这个方法会设定JFrame的大小。

你会帮这个方法写程序代码，但是不会调用它。

当用户确实做了某个操作之后，它就会发生。

大部分都是事件源。

把数据带回监听者。

addXxxListener()会说对象是个……

监听者是如何注册的？

所有绘图程序代码的去处。

通常会绑定在某个实例上。

(Graphic g)的这个g，其实是个……

推动paintComponent()的那只手。

大部分Swing呆的地方。



## 练习

```

import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class InnerButton {
 JFrame frame;
 JButton b;

 public static void main(String [] args) {
 InnerButton gui = new InnerButton();
 gui.go();
 }

 public void go() {
 frame = new JFrame();
 frame.setDefaultCloseOperation(
 JFrame.EXIT_ON_CLOSE);

 b = new JButton("A");
 b.addActionListener();

 frame.getContentPane().add(
 BorderLayout.SOUTH, b);
 frame.setSize(200,100);
 frame.setVisible(true);
 }

 class BListener extends ActionListener {
 public void actionPerformed(ActionEvent e) {
 if (b.getText().equals("A")) {
 b.setText("B");
 } else {
 b.setText("A");
 }
 }
 }
}

```

## 我是编译器



这左边的Java程序代码代表一份完整的源文件。你的任务是要扮演编译器角色并判断这支程序是否可以编译过关。如果有问题，哪里要修改？如果没有问题，那它是做什么的？

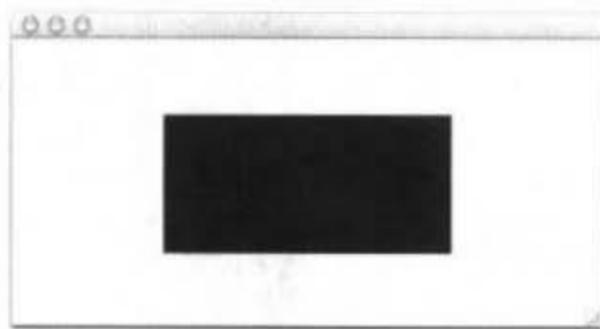


## 泳池迷宫



你的任务是要从游泳池中挑出程序片段并将它填入右边的空格中。同一个片段有可能使用两次以上，且泳池中有些多余的片段。填完空格的程序必须能够编译与执行并产生出下面的输出。

输出：一个完美、了不起的蓝色方块，还会慢慢变小变白。



```

g.fillRect(x,y,x-500,y-250)
g.fillRect(x,y,500-x*2,250-y*2)
g.fillRect(500-x*2,250-y*2,x,y)
x++ g.fillRect(0,0,250,500)
y++ g.fillRect(0,0,500,250)

g.setColor(blue) g
g.setColor(white) draw
g.setColor(Color.blue) drawP.paint()
g.setColor(Color.white) frame draw.repaint()
panel drawP.repaint()

```

```

import javax.swing.*;
import java.awt.*;
public class Animate {
 int x = 1;
 int y = 1;
 public static void main (String[] args) {
 Animate gui = new Animate ();
 gui.go();
 }
 public void go() {
 JFrame _____ = new JFrame();
 frame.setDefaultCloseOperation(
 JFrame.EXIT_ON_CLOSE);
 _____.getContentPane().add(drawP);
 _____;
 _____.setVisible(true);
 for (int i=0; i<124; _____) {
 _____;
 _____;
 }
 try {
 Thread.sleep(50);
 } catch (Exception ex) { }
 }
}
class MyDrawP extends JPanel {
 public void paintComponent (Graphic
 _____) {
 _____;
 _____;
 _____;
 _____;
 }
}

```



## 练习解答

## 我是谁？

JFrame

Listener interface

actionPerformed( )

setSize( )

paintComponent( )

event

swing component

event object

event source

addActionListener( )

paintComponent( )

inner class

Graphics2D

repaint( )

javax.swing

## 我是编译器

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
```

```
class InnerButton {
```

```
 JFrame frame;
 JButton b;
```

```
 public static void main(String [] args) {
 InnerButton gui = new InnerButton();
 gui.go();
 }
```

```
 public void go() {
 frame = new JFrame();
 frame.setDefaultCloseOperation(
 JFrame.EXIT_ON_CLOSE);

 b = new JButton("A");
 b.addActionListener(new BListener());
```

```
 frame.getContentPane().add(
 BorderLayout.SOUTH, b);
 frame.setSize(200,100);
 frame.setVisible(true);
 }
```

```
 class BListener implements ActionListener {
 public void actionPerformed(ActionEvent e) {
 if (b.getText().equals("A")) {
 b.setText("B");
 } else {
 b.setText("A");
 }
 }
 }
}
```

修好之后，这个  
程序会产生一个  
按钮并在点选的  
时候在A与B之  
间切换

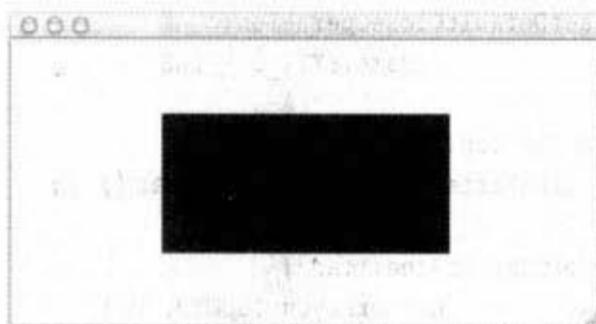


## 泳池迷宫

```

import javax.swing.*;
import java.awt.*;
public class Animate {
 int x = 1;
 int y = 1;
 public static void main (String[] args) {
 Animate gui = new Animate ();
 gui.go ();
 }
 public void go () {
 JFrame frame = new JFrame ();
 frame.setDefaultCloseOperation (
 JFrame.EXIT_ON_CLOSE);
 MyDrawP drawP = new MyDrawP ();
 frame.getContentPane ().add (drawP);
 frame.setSize (500, 270);
 frame.setVisible (true);
 for (int i = 0; i < 124; i++, x++, y++) {
 x++;
 drawP.repaint ();
 try {
 Thread.sleep (50);
 } catch (Exception ex) { }
 }
 }
 class MyDrawP extends JPanel {
 public void paintComponent (Graphics g) {
 g.setColor (Color.white);
 g.fillRect (0, 0, 500, 250);
 g.setColor (Color.blue);
 g.fillRect (x, y, 500 - x * 2, 250 - y * 2);
 }
 }
}

```



## 13 swing

### 运用Swing



Swing真的很简单。虽然看起来没什么难度，等到执行时你会发现“位置怎么跑掉了？”很容易写也会很难控制的原因在于“布局管理器”。这个对象可以控制Java的GUI上的widget的大小与位置。它帮你做了很多事情，但并不一定是你想要的结果。你想让两个按钮保持同样的大小，但却没有。打算让文字字段的长度保持在3寸长，结果却是9寸。但是只要稍微运作一下，就可以让布局管理器按照你的想法执行。这一章会讨论Swing和布局管理器，以及更多有关widget的事情。

## Swing 的组件

组件 (component, 或称元件) 是比我们之前所称的widget更为正确的术语。它们就是你会放在 GUI 上面的东西。这些东西是用户会看到并与其交互的，像是Text Field、button、scrollable list、radio button等。事实上所有的组件都是继承自javax.swing.JComponent。

### 组件是可以嵌套的

在Swing中，几乎所有组件都能够安置其他的组件。也就是说，你可以把任何东西放在其他东西上。但在大部分的情况下，你会把像是按钮或列表等用户交互组件放在框架和面板等背景组件上。

除了JFrame之外，交互组件与背景组件的差异不太明确。举例来说 JPanel通常用在背景上，但是也可以与用户交互。就跟其他组件一样，你也可以向 JPanel注册鼠标的点选等事件。

从技术上来说，widget是个Swing的组件，几乎所有的GUI组件都来自于java.swing.JComponent。

### 创建GUI四个步骤的回顾

#### ① 创建window (JFrame)。

```
JFrame frame = new JFrame();
```

#### ② 创建组件。

```
JButton button = new JButton("click me");
```

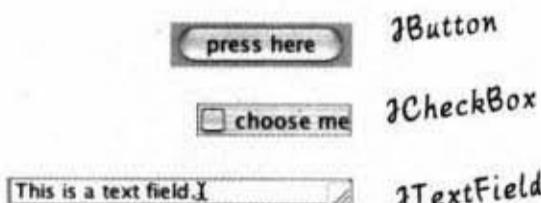
#### ③ 把组件加到frame上。

```
frame.getContentPane().add(BorderLayout.EAST, button);
```

#### ④ 显示出来。

```
frame.setSize(300,300);
frame.setVisible(true);
```

把组件：



放到背景组件上：



## 布局管理器 (Layout Managers)

布局管理器是个与特定组件相关联的Java对象，它大多数是背景组件。布局管理器用来控制所关联组件上携带的其他组件。也就是说，如果某个框架带有面板，而面板带有按钮，则面板的布局管理器控制着按钮的大小与位置，而框架的布局管理器则控制着面板的大小与位置。按钮因为没有携带其他组件，所以不需要布局管理器。

如果面板带有5项组件，就算这5项都有自己的布局管理器，它们的大小与位置都还是由面板的布局管理器来管理。

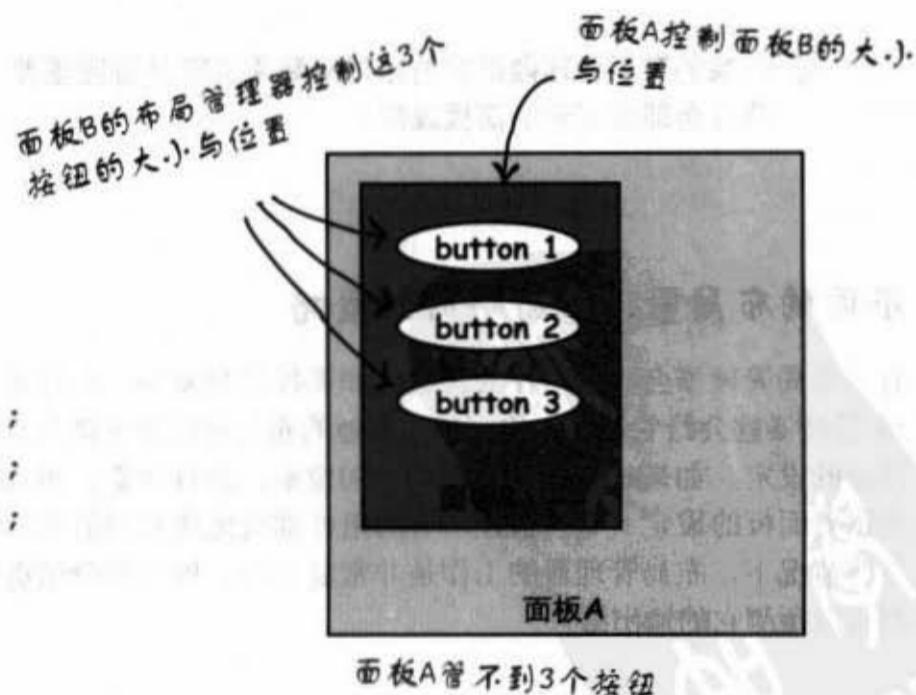
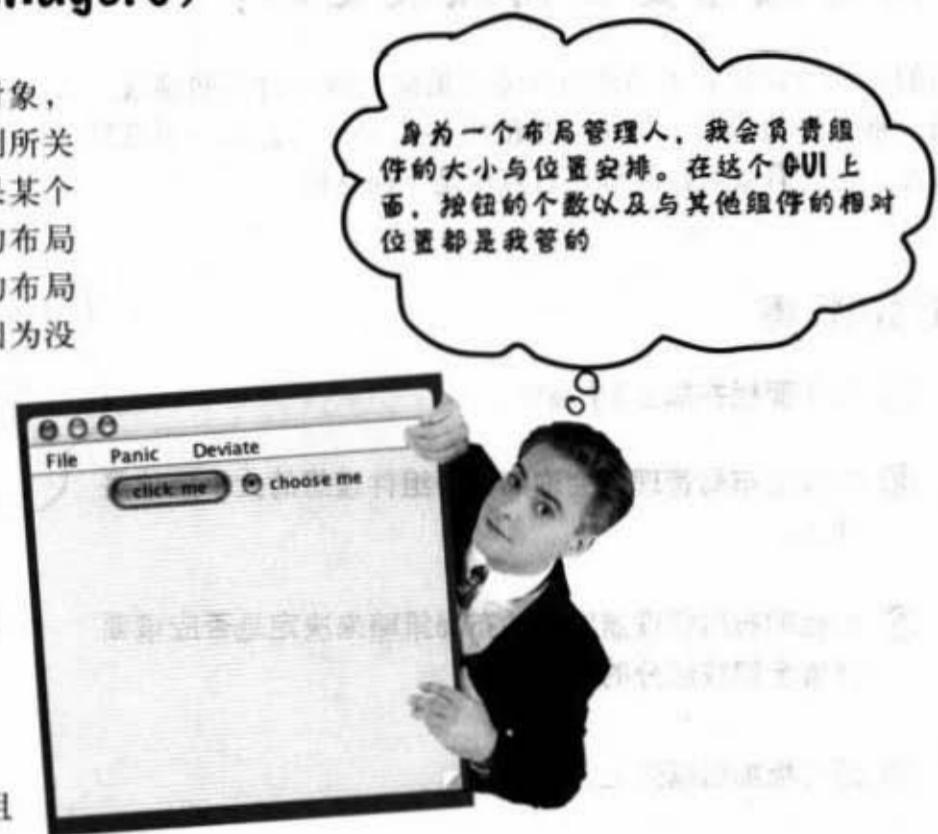
携带的意思就是加入到上面的，面板携带按钮就是因为按钮像下面这样被加到面板上。

```
myPanel.add(button);
```

布局管理器有几种不同的类型，每个背景组件都可以有自定义规则的布局管理器。例如某个布局管理器会让所有的面板维持相同的大小，而另一个布局管理器会让组件自行设定大小，但却又垂直对齐。

下面是嵌套布局的例子：

```
JPanel panelA = new JPanel();
JPanel panelB = new JPanel();
panelB.add(new JButton("button 1"));
panelB.add(new JButton("button 2"));
panelB.add(new JButton("button 3"));
panelA.add(panelB);
```



## 布局管理器是如何做决定的？

不同的布局管理器有不同的组件安置策略（例如对齐网格线、大小一致、垂直堆放等）。但被安排的组件至少可以表达一下意见。一般来说，处理背景组件的程序有点像是下面这样。

### 布局的情境

- ① 制作面板并加上3个按钮。
- ② 面板的布局管理器会询问每个组件理想的大小应该是什么。
- ③ 面板的布局管理器以它的布局策略来决定是否应该要尊重全部或部分的按钮理想。
- ④ 把面板加到框架上。
- ⑤ 框架的布局管理器询问面板的理想尺寸。
- ⑥ 框架的布局管理器以它的布局策略来决定是否应该要尊重全部或部分的面板理想。

嗯……嗯……第一个按钮得要有50mm宽、文字字段希望挑高至少有6mm、框架想要粉红色的天花板……喂！这不关我的事吧？



传说中的布局  
管理员

### 不同的布局管理器有不同的策略

有些布局管理器会尊重组件的想法。如果按钮想要 $30 \times 50$ 像素，布局管理器就会给它这么大的面积。其他的布局管理器可能只会尊重部分的设定。如果此时按钮想要 $30 \times 50$ 像素，会有30宽，但高度得跟着面板的设定。有些会让所有的组件都设定成相同的宽度。在某些情况下，布局管理器的工作是非常复杂的。但大部分情况下你都可以预测它的输出结果。

## 世界三大首席管理器： border、flow和box

### BorderLayout

这个管理器会把背景组件分割成5个区域。每个被管理的区域只能放上一个组件。由此管理员安置的组件通常不会取得默认的大小。这是框架默认的布局管理器！



每个区域只有一个组件

### FlowLayout

这个管理器的行为跟文书处理程序的版面配置方式差不多。每个组件都会依照理想的大小呈现，并且会从左到右依照加入的顺序以可能会换行的方式排列。因此在组件放不下的时候会被放到下一行。这是面板默认的布局管理器！



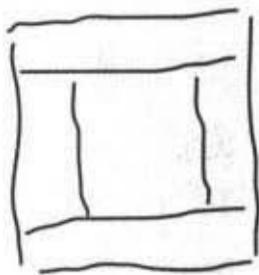
从左至右排列，有必要时会换行

### BoxLayout

它就像FlowLayout一样让每个组件使用默认的大小，并且按照加入的顺序来排列。但BoxLayout是以垂直的方向来排列（也可以水平，但通常我们只在乎垂直方式）。不像FlowLayout会自动地换行，它让你插入某种类似换行的机制来强制组件从新的一行开始排列。



从上到下每行一个



## BorderLayout布局的5个区域： 东区、西区、北区、南区与中央区

将一个按钮加入东区：

```
import javax.swing.*;
import java.awt.*; ← 它在java.awt这个包里面

public class Button1 {

 public static void main (String[] args) {
 Button1 gui = new Button1();
 gui.go();
 }

 public void go() {
 JFrame frame = new JFrame();
 JButton button = new JButton("click me");
 frame.getContentPane().add(BorderLayout.EAST, button);
 frame.setSize(200,200);
 frame.setVisible(true);
 }
}
```

 **Brain Barbell**

(1) BorderLayout是如何设定按钮的大小?  
(2) 有哪些因素是必须考虑的?  
(3) 它为什么不会更宽或更高?



## 注意按钮字变多时所发生的事……

```
public void go() {
 JFrame frame = new JFrame();
 JButton button = new JButton("click like you mean it");
 frame.getContentPane().add(BorderLayout.EAST, button);
 frame.setSize(200, 200);
 frame.setVisible(true);
}
```



只有改变按钮的文字



宽度没问题，但高度由管理器决定



## 尝试进驻北方

```
public void go() {
 JFrame frame = new JFrame();
 JButton button = new JButton("There is no spoon...");
 frame.getContentPane().add(BorderLayout.NORTH, button);
 frame.setSize(200, 200);
 frame.setVisible(true);
}
```



按钮的高度跟默认一样，但  
与frame同宽

## 让按钮要求更多的高度

怎么做？按钮已经是最宽了——跟框架一样，但  
我们可以用更大的字体来让它更高。

```
public void go() {
 JFrame frame = new JFrame();
 JButton button = new JButton("Click This!");
 Font bigFont = new Font("serif", Font.BOLD, 28);
 button.setFont(bigFont);
 frame.getContentPane().add(BorderLayout.NORTH, button);
 frame.setSize(200, 200);
 frame.setVisible(true);
}
```



更大的字体会强迫框架留更  
多的高度给按钮

宽度还是一样，但更高了



那中间区域会发生  
什么事？

## 中间区域只能捡剩下的（稍后介绍特殊情况）

```
public void go() {
 JFrame frame = new JFrame();

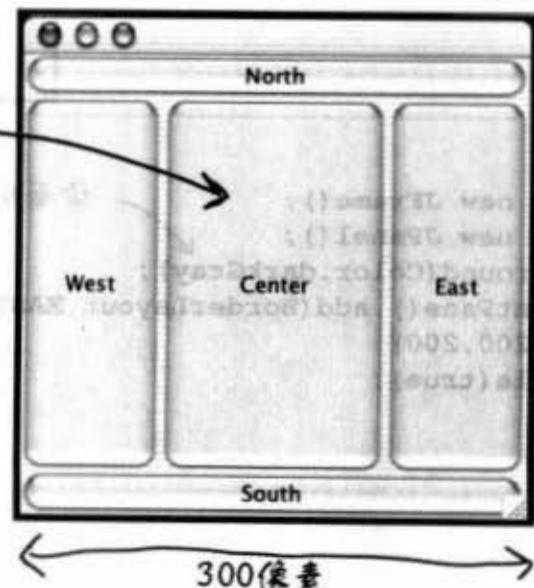
 JButton east = new JButton("East");
 JButton west = new JButton("West");
 JButton north = new JButton("North");
 JButton south = new JButton("South");
 JButton center = new JButton("Center");

 frame.getContentPane().add(BorderLayout.EAST, east);
 frame.getContentPane().add(BorderLayout.WEST, west);
 frame.getContentPane().add(BorderLayout.NORTH, north);
 frame.getContentPane().add(BorderLayout.SOUTH, south);
 frame.getContentPane().add(BorderLayout.CENTER, center);

 frame.setSize(300, 300);
 frame.setVisible(true);
}
```

中央的组件大小要看扣除  
周围之后还剩下些什么

东西会取得预设的宽度  
南北会取得预设的高度



南北会先定位，所  
以东西的高度还要  
扣除南北的高度



## FlowLayout布局组件 的流向：依次从左至 右、从上至下

### 把面板加入东区：

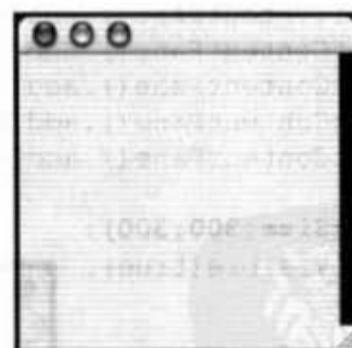
JPanel的布局管理器的默认布局是FlowLayout布局。当我们把面板加到框架时，面板的大小与位置还是受BorderLayout布局的管理。但面板内部的组件（通过panel.add(someComponent)加入）是由面板的FlowLayout布局来管理的。我们先把空的面板放到框架的东区，下一页会再加组件到面板上。

面板还没有东西在上面，所  
以不会要求太多的区域

```
import javax.swing.*;
import java.awt.*;

public class Panell {
 public static void main (String[] args) {
 Panell gui = new Panell();
 gui.go();
 }

 public void go() {
 JFrame frame = new JFrame();
 JPanel panel = new JPanel();
 panel.setBackground(Color.darkGray);
 frame.getContentPane().add(BorderLayout.EAST, panel);
 frame.setSize(200,200);
 frame.setVisible(true);
 }
}
```



让面板变成深灰色以便观察

## 把按钮加到面板上

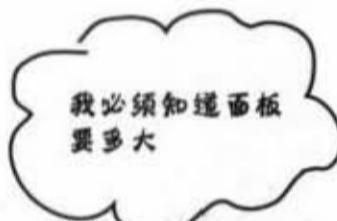
```

public void go() {
 JFrame frame = new JFrame();
 JPanel panel = new JPanel();
 panel.setBackground(Color.darkGray);

 JButton button = new JButton("shock me");
 panel.add(button);
 frame.getContentPane().add(BorderLayout.EAST, panel);

 frame.setSize(250, 200);
 frame.setVisible(true);
}

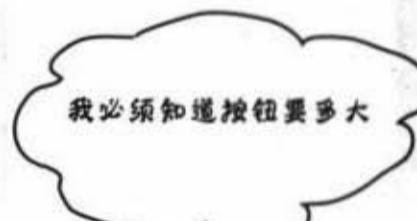
```



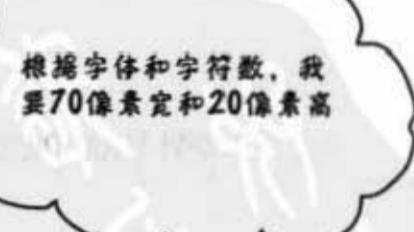
控制



Panel 对象



控制



Button 对象

框架的BorderLayout布局管理员

面板的FlowLayout布局管理员

# 如果加两个按钮到面板上？

```

public void go() {
 JFrame frame = new JFrame();
 JPanel panel = new JPanel();
 panel.setBackground(Color.darkGray);

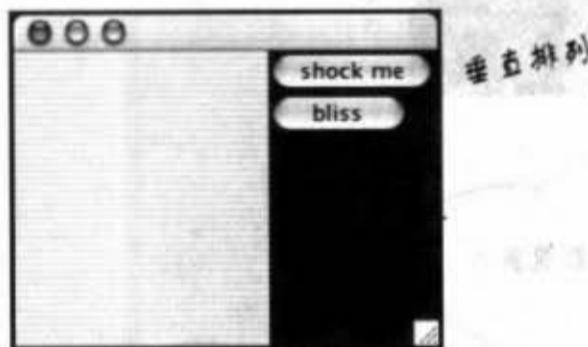
 JButton button = new JButton("shock me"); 创建两个按钮
 JButton buttonTwo = new JButton("bliss");

 panel.add(button); 加到面板上
 panel.add(buttonTwo);

 frame.getContentPane().add(BorderLayout.EAST, panel);
 frame.setSize(250, 200);
 frame.setVisible(true);
}

```

想要的效果：



实际的效果：



注意到字少的宽度比较小，顺序布局会让按钮取得刚好所需的大...

 Sharpen your pencil

如果上面的程序改成下面这样，GUI会长得什么样子？

```

JButton button = new JButton("shock me");
JButton buttonTwo = new JButton("bliss");
JButton buttonThree = new JButton("huh?");
panel.add(button);
panel.add(buttonTwo);
panel.add(buttonThree);

```



把你认为的结果画出来！



BoxLayout布局是救星！

就算够宽它还是会垂直排列

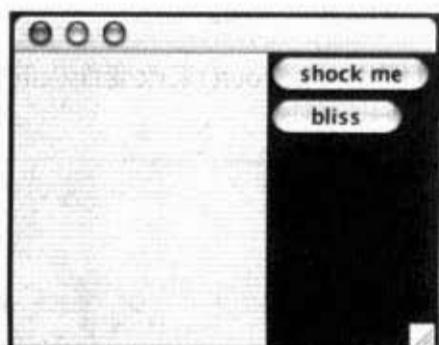
不像FlowLayout布局，就算水平宽度足以容纳组件，它还是会用新的行来排列组件

所以你现在必须把面板的布局管理器从默认的FlowLayout布局改成BoxLayout布局

```
public void go() {
 JFrame frame = new JFrame();
 JPanel panel = new JPanel();
 panel.setBackground(Color.darkGray);
 panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
 JButton button = new JButton("shock me");
 JButton buttonTwo = new JButton("bliss");
 panel.add(button);
 panel.add(buttonTwo);
 frame.getContentPane().add(BorderLayout.EAST, panel);
 frame.setSize(250, 200);
 frame.setVisible(true);
}
```

把布局管理器换掉

它的构造函数需要知道要管理哪个组件以及使用哪个轴



面板又变窄了，因为不需要水平地塞入组件，因此只要够放最宽的那个就可以了。

there are no  
Dumb Questions

## 要点

**问：** 框架为什么不能像面板那样直接地加上组件？

**答：** JFrame会这么特殊是因为它是让事物显示在画面上的接点。因为Swing的组件纯粹由Java构成，JFrame必须连接到底层的操作系统以便来存取显示装置。我们可以把面板想做是安置在JFrame上的100%纯Java层。或者把JFrame想做是支撑面板的框架。你甚至可以用自定义的 JPanel来换掉框架的面板：

```
myFrame.getContentPane(myPanel);
```

**问：** 我能够换掉框架的布局管理器吗？如果我想让框架用顺序替换边界呢？

**答：** 最简单的方法是创建一个面板，让此面板成为框架的content pane，使得GUI以你想要的方式运行。

**问：** 如果想要有不同的理想大小应该怎么办？组件是否有setSize()方法？

**答：** 是有setSize()，但布局管理器会把它忽略掉。组件理想的大小与你想要的大小是有差距的。理想的大小是根据组件确实所需的大小来计算的（组件自行计算）。布局管理器会调用组件的getPreferredSize()方法，而此方法并不会考虑你之前对setSize()的调用。

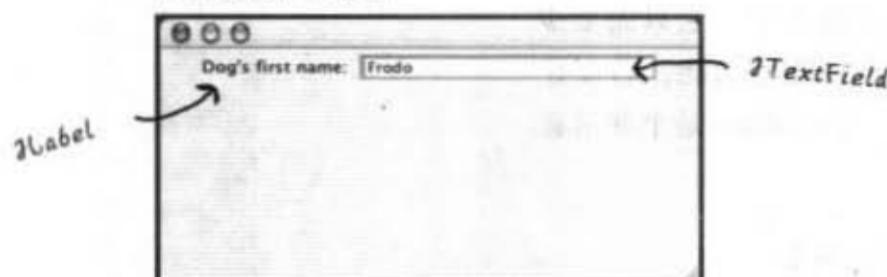
**问：** 能不能直接定位？能不能关掉布局管理器？

**答：** 你可以调用setLayout(null)直接设定画面位置和大小。但使用布局管理器还是比较好的方式。

## 操作Swing组件

你已经看过布局管理器的基本说明，因此现在就让我们来看一下几个最常用的组件：text field、可滚动的text area、checkbox以及list。我们不打算把整个API都拿出来说一遍，只讲几个重点。

### JTextField



#### 构造函数

```
JTextField field = new JTextField(20);
JTextField field = new JTextField("Your name");
```

20代表20字宽而不是像素

#### 如何使用

- ① 取得文本内容。

```
System.out.println(field.getText());
```

- ② 设定内容。

```
field.setText("whatever");
field.setText("");
```

清空字段

- ③ 取得用户输入完毕按下return或enter键的事件。

如果想要知道用户的每个按键操作也可以注册按键的事件

```
field.addActionListener(myActionListener);
```

- ④ 选取文本字段的内容。

```
field.selectAll();
```

- ⑤ 把GUI目前焦点拉回到文本字段以便让用户进行输入操作。

```
field.requestFocus();
```

## 文本域

### JTextArea



不像 JTextField, JTextArea可以有超过一行以上的文字。它只需要少许的设定就可制作出来，因为这样没有做滚动条或换行功能。若要让 JTextArea滚动，就必须要把它粘在ScrollPane上。ScrollPane是个非常喜欢滚动的对象，并也会考虑文本区域的滚动需求。

#### 构造函数

```
JTextArea text = new JTextArea(10, 20);
```

代表10行高  
20字宽

#### 如何使用

##### ● 只有垂直的滚动条。

```
JScrollPane scroller = new JScrollPane(text);
text.setLineWrap(true); // 启动自动换行
scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
panel.add(scroller); // 这很重要，加入的是带有文本域的滚动条而不是文本域！
```

##### ● 替换掉文字内容。

```
text.setText("Not all who are lost are wandering");
```

##### ● 加入文字。

```
text.append("button clicked");
```

##### ● 选取内容。

```
text.selectAll();
```

##### ● 把GUI目前焦点拉回到文本字段以便让用户进行输入操作。

```
text.requestFocus();
```

## JTextArea范例

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class TextAreal implements ActionListener {
 JTextArea text;

 public static void main (String[] args) {
 TextAreal gui = new TextAreal();
 gui.go();
 }

 public void go() {
 JFrame frame = new JFrame();
 JPanel panel = new JPanel();
 JButton button = new JButton("Just Click It");
 button.addActionListener(this);
 text = new JTextArea(10,20);
 text.setLineWrap(true);

 JScrollPane scroller = new JScrollPane(text);
 scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
 scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

 panel.add(scroller);

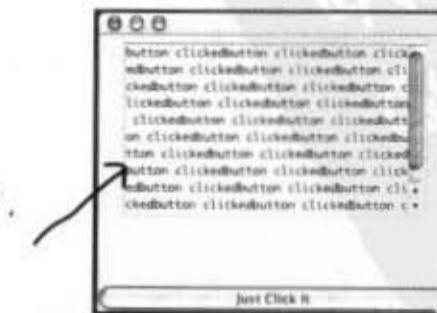
 frame.getContentPane().add(BorderLayout.CENTER, panel);
 frame.getContentPane().add(BorderLayout.SOUTH, button);

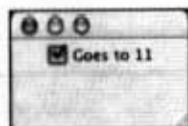
 frame.setSize(350,300);
 frame.setVisible(true);
 }

 public void actionPerformed(ActionEvent ev) {
 text.append("button clicked \n ");
 }
}

```

↑  
在按下按钮时插入一个换行字符。  
不然的话都会粘在一起。



**JCheckBox****构造函数**

```
JCheckBox check = new JCheckBox("Goes to 11");
```

**如何使用**

① 监听 item 的事件（被选取或变成非选取）。

```
check.addItemListener(this);
```

② 处理事件（判别是否被选取）。

```
public void itemStateChanged(ItemEvent ev) {
 String onOrOff = "off";
 if (check.isSelected()) onOrOff = "on";
 System.out.println("Check box is " + onOrOff);
}
```

③ 用程序来选取或不选取。

```
check.setSelected(true);
check.setSelected(false);
```

there are no  
Dumb Questions

**问：** 布局管理器产生的问题是是不是比所解决的问题更多？如果我得处理这么多的问题，那我得考虑是否直接设定位置和大小算了。

**答：** 以布局管理器取得完全符合想法的布局是个很大的挑战。但要考虑到它还帮你做了哪些事情。就算是计算组件要出现在画面的什么位置也是很复杂的工作。举例来说，布局管理器能够防止组件互相覆盖。也就是说，它知道如何管理组件（以及框架）的间距。你当然可以自己做到这样的功能，但如果组件多到一定程度你还会想要手动的来调整吗？这只有对Java虚拟机有好处而已！

为什么？因为组件在不同平台上长得不太一样，像是在某平台上刚好并排按钮边缘在另外一个平台上可能就重叠了。

这还不是最麻烦的，只要想象用户调整window大小时会发生什么事。你应该会很庆幸有机会不用自己写这么多与真正商业逻辑无关的程序代码吧？

**JList**

*JList*的构造函数需要一个任意类型的数组。不一定要是String, 但会用String来表示项目

**构造函数**

```
String [] listEntries = {"alpha", "beta", "gamma", "delta",
 "epsilon", "zeta", "eta", "theta "};

list = new JList(listEntries);
```

**如何使用**

① 让它显示垂直的滚动条。

与 JTextArea 相同, 要放在 JScrollPane 上面

```
JScrollPane scroller = new JScrollPane(list);
scroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panel.add(scroller);
```

② 设定显示的行数。

```
list.setVisibleRowCount(4);
```

③ 限制用户只能选取一个项目。

```
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

④ 对选择事件做注册。

```
list.addListSelectionListener(this);
```

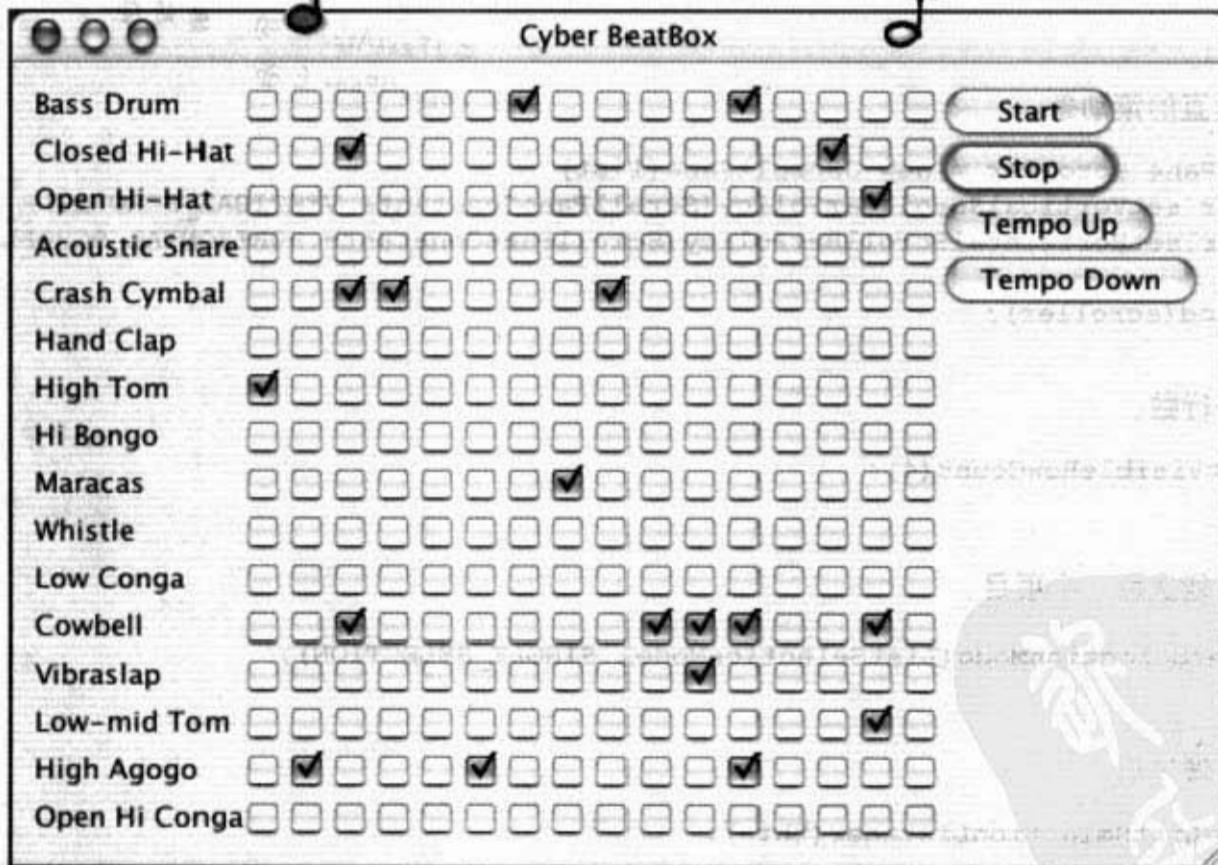
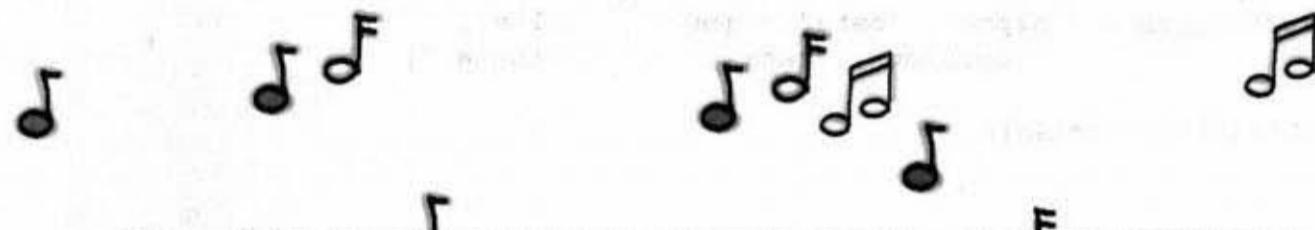
如果没有加上这个 if 测试,  
你会得到两次的事件

⑤ 处理事件 (判断选了哪个项目)。

```
public void valueChanged(ListSelectionEvent lse) { ←
 if(!lse.getValueIsAdjusting()) {
 String selection = (String) list.getSelectedValue(); ←
 System.out.println(selection);
 }
}
```

这会返回一个 Object, 不  
一定是个 String

# 程序料理



这一段是选择性的内容，我们会来创造出完整版本的BeatBox。我们在讨论对象存储的章节会学到如何存储与打开节奏设定。在讨论网络的章节中我们会把BeatBox改成聊天室的客户端程序。

## 创建BeatBox

接下来有这一版BeatBox的程序行表。其中带有启动、停止和改变节奏的按钮。这是一个完整的程序行表，并有注释。以下是这个程序的概要。

- 创建出带有256个复选框的GUI。初始的时候这些复选框都是未勾选的，乐器的名称用到16个JLabel，还有4个按钮。
- 对上面的4个按钮注册ActionListener。我们无需个别的监听复选框，因为我们不会试着动态地（在复选框被点选时）马上改变发声的样式。相反，我们会等到用户按下start按钮之后才会检查这256个复选框的状态并制作出MIDI的track。
- 设定MIDI系统（之前就已经做过了），这包括取得Sequencer、创建Sequence以及track。我们会用到Java 5.0之后才有的setLoopCount()这个sequencer的方法。它能让你指定重复播放的次数。我们也会用到节奏因子(tempo factor)来调整节奏的速度，并维持重复时的节奏。
- 当用户按下start时，启动真正的操作。此按钮的事件处理程序会调用buildTrackAndStart()方法。在该方法中，我们会逐个（一次一行）取得256个复选框的状态，然后使用这些信息来创建MIDI的track（使用之前编写的makeEvent()）。一旦track完成之后，我们会启动sequencer来持续播放直到用户按下stop为止。

## BeatBox的程序代码

```
import java.awt.*;
import javax.swing.*;
import javax.sound.midi.*;
import java.util.*;
import java.awt.event.*;

public class BeatBox {

 JPanel mainPanel;
 ArrayList<JCheckBox> checkboxList; 把checkbox储存到ArrayList中
 Sequencer sequencer;
 Sequence sequence;
 Track track;
 JFrame theFrame;

 String[] instrumentNames = {"Bass Drum", "Closed Hi-Hat",
 "Open Hi-Hat", "Acoustic Snare", "Crash Cymbal", "Hand Clap",
 "High Tom", "Hi Bongo", "Maracas", "Whistle", "Low Conga",
 "Cowbell", "Vibraslap", "Low-mid Tom", "High Agogo",
 "Open Hi Conga"};
 int[] instruments = {35, 42, 46, 38, 49, 39, 50, 60, 70, 72, 64, 56, 58, 47, 67, 63};

 public static void main (String[] args) {
 new BeatBox2().buildGUI(); 实际的乐器关键字，比如说35是bass, 42是Closed Hi-Hat
 }

 public void buildGUI() {
 theFrame = new JFrame("Cyber BeatBox");
 theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 BorderLayout layout = new BorderLayout();
 JPanel background = new JPanel(layout);
 background.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10)); 设定面板上摆放组件时的空边框

 checkboxList = new ArrayList<JCheckBox>();
 Box buttonBox = new Box(BoxLayout.Y_AXIS);

 JButton start = new JButton("Start");
 start.addActionListener(new MyStartListener());
 buttonBox.add(start);

 JButton stop = new JButton("Stop");
 stop.addActionListener(new MyStopListener());
 buttonBox.add(stop); 一般的GUI程序代码

 JButton upTempo = new JButton("Tempo Up");
 upTempo.addActionListener(new MyUpTempoListener());
 buttonBox.add(upTempo);

 JButton downTempo = new JButton("Tempo Down");
 }
}
```

```

downTempo.addActionListener(new MyDownTempoListener());
buttonBox.add(downTempo);

Box nameBox = new Box(BoxLayout.Y_AXIS);
for (int i = 0; i < 16; i++) {
 nameBox.add(new Label(instrumentNames[i]));
}

background.add(BorderLayout.EAST, buttonBox);
background.add(BorderLayout.WEST, nameBox);

theFrame.getContentPane().add(background);

GridLayout grid = new GridLayout(16,16);
grid.setVgap(1);
grid.setHgap(2);
mainPanel = new JPanel(grid);
background.add(BorderLayout.CENTER, mainPanel);

for (int i = 0; i < 256; i++) {
 JCheckBox c = new JCheckBox();
 c.setSelected(false);
 checkboxList.add(c);
 mainPanel.add(c);
} // 循环结束

```

也是一般的程序代码

创建checkbox组，设定成未勾选的为false并加到ArrayList和面板上

```

setUpMidi();

theFrame.setBounds(50,50,300,300);
theFrame.pack();
theFrame.setVisible(true);
} // 关闭方法

```

```

public void setUpMidi() {
 try {
 sequencer = MidiSystem.getSequencer();
 sequencer.open();
 sequence = new Sequence(Sequence.PPQ, 4);
 track = sequence.createTrack();
 sequencer.setTempoInBPM(120);
 } catch (Exception e) {e.printStackTrace();}
} // 关闭方法

```

一般的MIDI设置程序代码

## BeatBox的程序代码

重点在这里！此处会将复选框状态转换为MIDI事件并加到track上

```
public void buildTrackAndStart() {
 int[] trackList = null; ← 创建出16个元素的数组来存储一项乐器的值。如果该
 sequence.deleteTrack(track); } 节应该要演奏，其值会是关键字值，否则值为零
 track = sequence.createTrack(); } 清除掉旧的track做一个新的

 for (int i = 0; i < 16; i++) { ← 对每个乐器都执行一次
 trackList = new int[16]; } 没有代表乐器的关键字

 int key = instruments[i]; ← 设定代表乐器的关键字

 for (int j = 0; j < 16; j++) { ← 对每一拍执行一次
 JCheckBox jc = (JCheckBox) checkboxList.get(j + (16*i));
 if (jc.isSelected()) {
 trackList[j] = key; } 如果有勾选，将关键字值放到数组的该位置上。
 } else { trackList[j] = 0; } 不然的话就补零
 }
 } // 关闭内部循环

 makeTracks(trackList); ← 创建此乐器的事件并加到track上
 track.add(makeEvent(176, 1, 127, 0, 16));
 } // 关闭外部循环

 track.add(makeEvent(192, 9, 1, 0, 15)); } 确保第16拍有事件，否则beatbox不会重
 try { 复播放

 sequencer.setSequence(sequence);
 sequencer.setLoopCount(sequencer.LOOP_CONTINUOUSLY); ← 指定无穷的重叠次数
 sequencer.start();
 sequencer.setTempoInBPM(120);
 } catch(Exception e) { e.printStackTrace(); } 开始播放！

} // 关闭 buildTrackAndStart方法 } 第一个内部类，按钮的监听者

public class MyStartListener implements ActionListener {
 public void actionPerformed(ActionEvent a) {
 buildTrackAndStart();
 }
} // 关闭内部类
```

```

public class MyStopListener implements ActionListener {
 public void actionPerformed(ActionEvent a) {
 sequencer.stop();
 }
} // 关闭内部类

public class MyUpTempoListener implements ActionListener {
 public void actionPerformed(ActionEvent a) {
 float tempoFactor = sequencer.getTempoFactor();
 sequencer.setTempoFactor((float)(tempoFactor * 1.03));
 }
} // 关闭内部类

public class MyDownTempoListener implements ActionListener {
 public void actionPerformed(ActionEvent a) {
 float tempoFactor = sequencer.getTempoFactor();
 sequencer.setTempoFactor((float)(tempoFactor * .97));
 }
} // 关闭内部类

```

另一个内部类，也是按钮的监听者

节奏因子，预设为1.0，每次调整3%

创建某项乐器的所有事件

```

public void makeTracks(int[] list) {

 for (int i = 0; i < 16; i++) {
 int key = list[i];

 if (key != 0) {
 track.add(makeEvent(144, 9, key, 100, i)); } 创建NOTE ON和NOTE OFF事
 track.add(makeEvent(128, 9, key, 100, i+1)); } 件并加入到track上
 }
 }

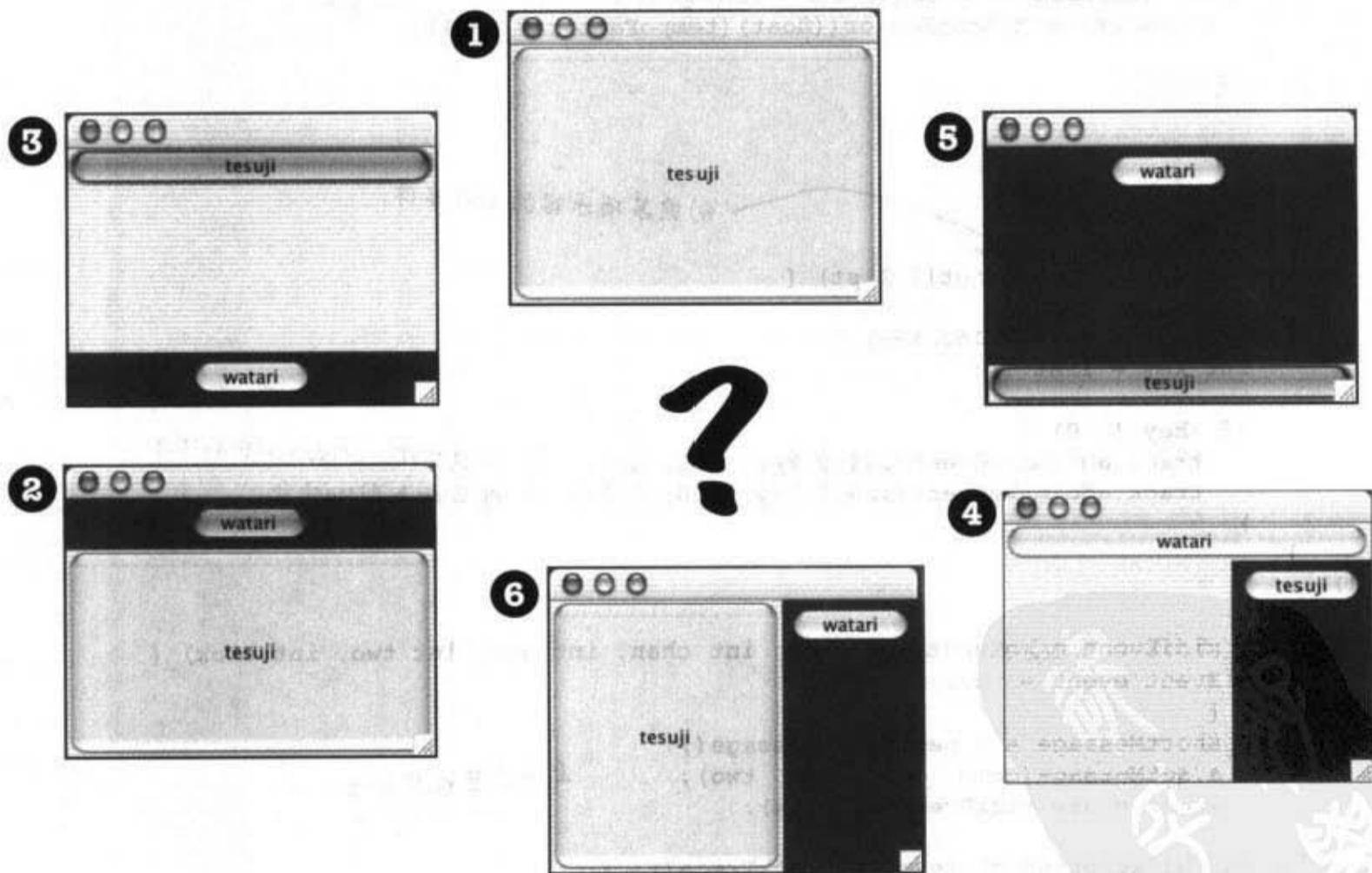
public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
 MidiEvent event = null;
 try {
 ShortMessage a = new ShortMessage();
 a.setMessage(comd, chan, one, two); 上一章已经用过了
 event = new MidiEvent(a, tick);
 } catch (Exception e) {e.printStackTrace();}
 return event;
}
} // 关闭类

```



## 哪一个程序用了哪一个layout?

下面的6个画面中有5个是由下一页的程序段所产生的。找出哪个画面是由哪个程序段落所产生的。



## 程序段落

**D**

```
JFrame frame = new JFrame();
 JPanel panel = new JPanel();
 panel.setBackground(Color.darkGray);
 JButton button = new JButton("tesuji");
 JButton buttonTwo = new JButton("watari");
 frame.getContentPane().add(BorderLayout.NORTH, panel);
 panel.add(buttonTwo);
 frame.getContentPane().add(BorderLayout.CENTER, button);
```

**B**

```
JFrame frame = new JFrame();
 JPanel panel = new JPanel();
 panel.setBackground(Color.darkGray);
 JButton button = new JButton("tesuji");
 JButton buttonTwo = new JButton("watari");
 panel.add(buttonTwo);
 frame.getContentPane().add(BorderLayout.CENTER, button);
 frame.getContentPane().add(BorderLayout.EAST, panel);
```

**C**

```
JFrame frame = new JFrame();
 JPanel panel = new JPanel();
 panel.setBackground(Color.darkGray);
 JButton button = new JButton("tesuji");
 JButton buttonTwo = new JButton("watari");
 panel.add(buttonTwo);
 frame.getContentPane().add(BorderLayout.CENTER, button);
```

**A**

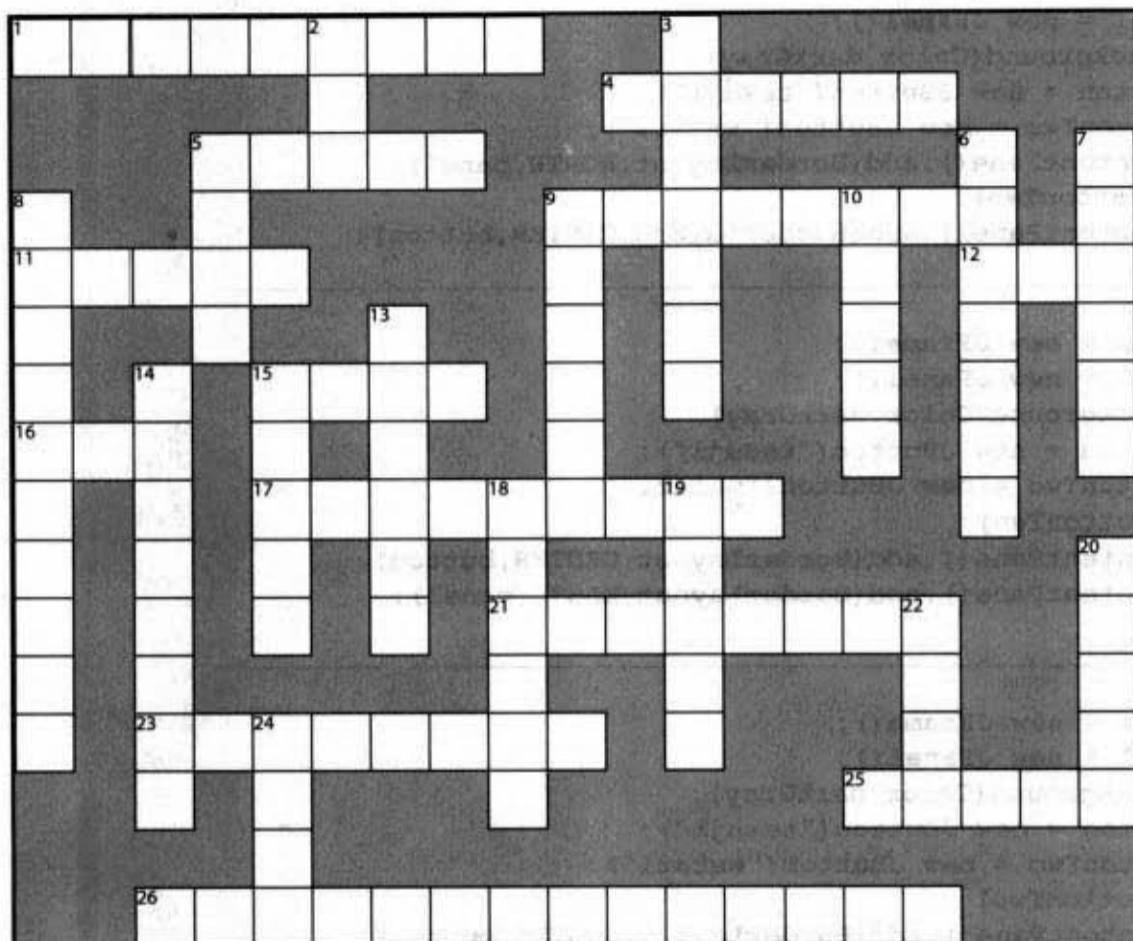
```
JFrame frame = new JFrame();
 JPanel panel = new JPanel();
 panel.setBackground(Color.darkGray);
 JButton button = new JButton("tesuji");
 JButton buttonTwo = new JButton("watari");
 panel.add(button);
 frame.getContentPane().add(BorderLayout.NORTH, buttonTwo);
 frame.getContentPane().add(BorderLayout.EAST, panel);
```

**E**

```
JFrame frame = new JFrame();
 JPanel panel = new JPanel();
 panel.setBackground(Color.darkGray);
 JButton button = new JButton("tesuji");
 JButton buttonTwo = new JButton("watari");
 frame.getContentPane().add(BorderLayout.SOUTH, panel);
 panel.add(buttonTwo);
 frame.getContentPane().add(BorderLayout.NORTH, button);
```



# GUI-Cross 7.0



你行吗？

## 横排提示：

- 1. Artist's sandbox
- 4. Border's catchall
- 5. Java look
- 9. Generic waiter
- 11. A happening
- 12. Apply a widget
- 15. JPanel's default
- 16. Polymorphic test

## 竖排提示：

- 17. Shake it baby
- 21. Lots to say
- 23. Choose many
- 25. Button's pal
- 26. Home of actionPerformed

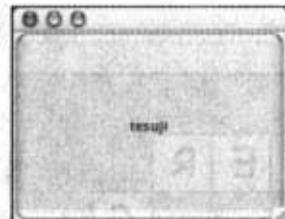
## 竖排提示：

- 13. Manager's rules
- 14. Source's behavior
- 15. Border by default
- 18. User's behavior
- 19. Inner's squeeze
- 20. Backstage widget
- 22. Mac look
- 24. Border's right



## 练习解答

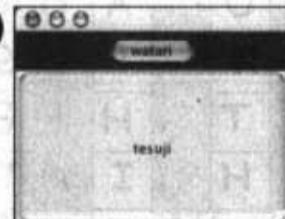
1



**C**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

2



**D**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.NORTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
```

3



**E**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.SOUTH,panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.NORTH,button);
```

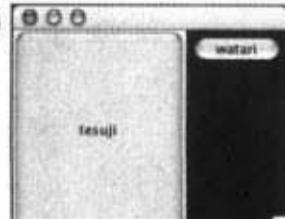
4



**A**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(button);
frame.getContentPane().add(BorderLayout.NORTH,buttonTwo);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

6

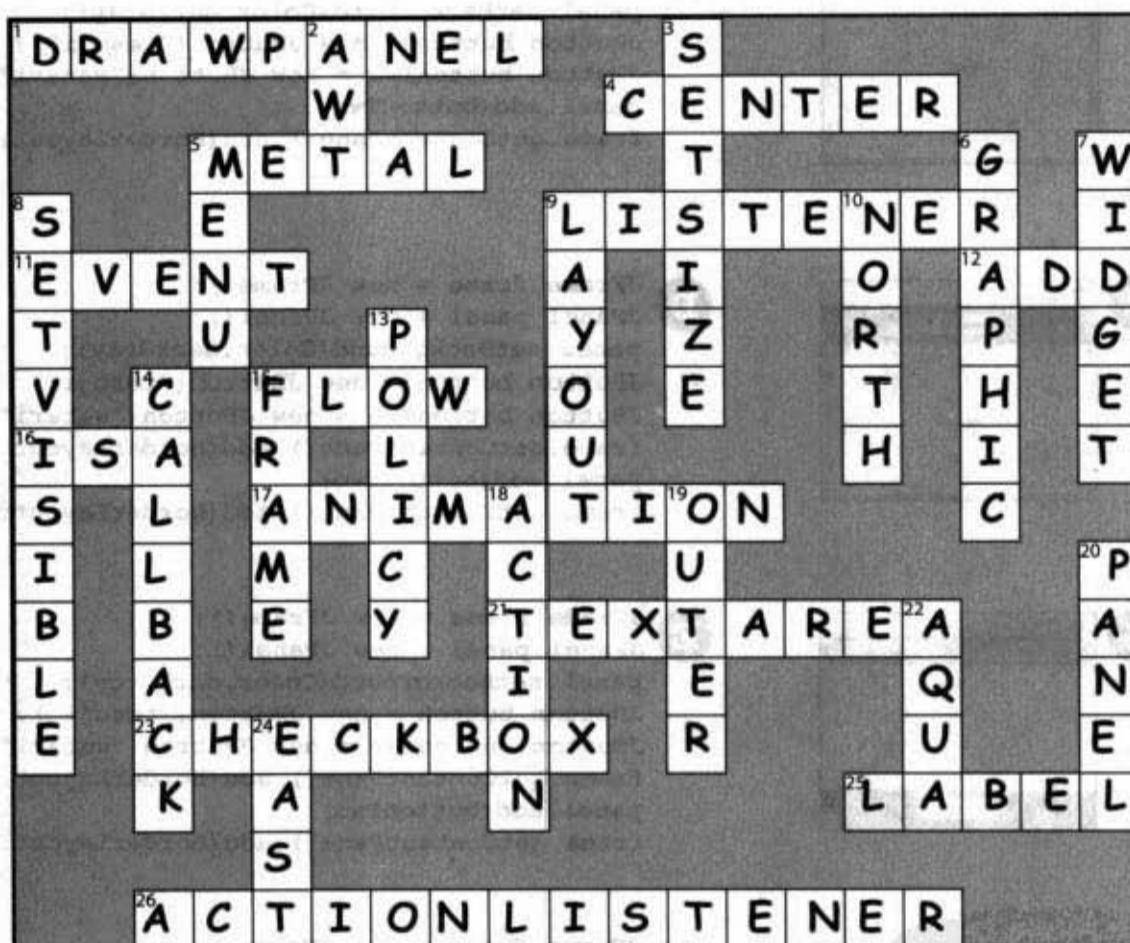


**B**

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER,button);
frame.getContentPane().add(BorderLayout.EAST, panel);
```



## GUI-Cross 7.0 解答



## 14 序列化和文件的输入/输出

### 保存对象



如果再让我读全是数据的文件，我就把电话装进他的嘴里。他知道我可以储存完整的对象，但就是不让我这么做……希望他吃得惯！

**对象可以被序列化也可以展开。**对象有状态和行为两种属性。行为存在于类中，而状态存在于个别的对象中。所以需要存储对象状态的时候会发生什么事？如果你正在编写游戏，就得有存储和恢复游戏的功能。如果你编写的是创建图表的程序，也必须要有储存/打开的功能。如果程序需要储存状态，你可以来硬的，对每一个对象，逐个地把每项变量的值写到特定格式的文件中。或者，你也可以用面向对象的方式来做——只要把对象本身给冻干/辗平/保存/脱水，并加以重组/展开/恢复/泡开成原状。但有时这还得来硬的，特别是在程序所储存的文件需要给某些非Java的应用程序所读取时，所以这一章会讨论这两种方式。

## 抓住节奏

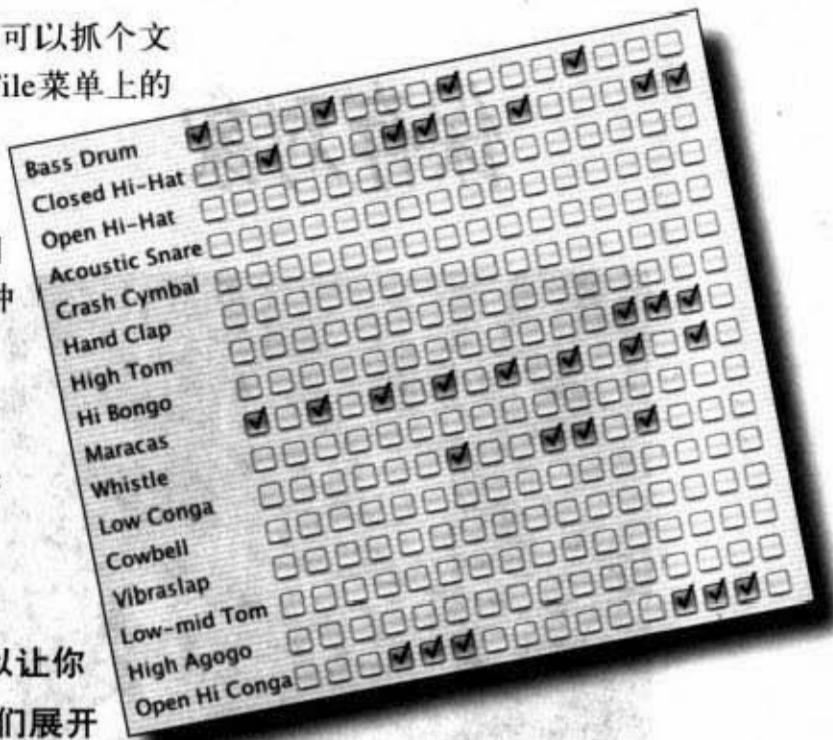
你已经奏出完美的乐章，现在会想把它储存起来。你可以抓个文房四宝把它记下来，但也可以按下储存按钮（或按下File菜单上的Save）。然后你帮文件命名，并希望这个文件不会让屏幕变成蓝色的画面。

储存状态的选择有很多种，这可能要看你会如何使用储存下来的状态而决定。我们会在这一章讨论下面两种选项：

如果只有自己写的Java程序会用到这些数据：

### ① 用序列化 (serialization) 。

将被序列化的对象写到文件中。然后就可以让你的程序去文件中读取序列化的对象并把它们展开回到活生生的状态。



如果数据需要被其他程序引用：

② 写一个纯文本文件。用其他程序可以解析的特殊字符写到文件中。例如写成用tab字符来分隔的档案以便让电子表格或数据库应用程序能够应用。

当然还有其他的选择。你可以将数据存进任何格式中。举例来说，你可以把数据用字节而不是字符来写入，或者你也可以将数据写成Java的primitive主数据类型，有一些方法可以提供int、long、boolean等的写入功能。但不管用什么方法，基本所需的输入/输出技巧都一样：把数据写到某处，这可能是个磁盘上的文件，或者是来自网络上的串流。读取数据的方向则刚好相反。当然此处所讨论的部分不涉及使用数据库的情况。

## 存储状态

假设你有个程序，是个幻想冒险游戏，要过很多关才能完成。在游戏进行的过程中，游戏的人物会累积经验值、宝物、体力等。你不会想让游戏每次重新启动时都得要从头来过——这样根本没人玩。因此你需要一种方法来保存人物的状态，并且在重新开启时能够将状态回复到上次存储时的原状。因为你是程序员，所以你的工作是要让存储与恢复尽可能的简单容易。

### ① 选项一

把3种序列化的人物对象写入文件中。

创建一个文件，让序列化的3种对象写到此文件中。这文件在你以文本文件形式阅读时是无意义的：

```
“isr GameCharacter
“%g 8M IpowerIjava/lang/
String;[weaponstIjava/lang/
String;x p2tlfur[Ljava.lang.String;“V 
 (Gxptbowtswordtdustsq~» tTrolluq-tb
are handstbig axsq-xtMagicianuq-tspe
llstinvisibility
```

### ② 选项二

写入纯文本文件

创建文件，写入3行文字，每个人物一行，以逗点来分开属性：

```
50,Elf,bow,sword,dust
200,Troll,bare hands,big ax
120,Magician,spells,invisibility
```

### GameCharacter

```
int power
String type
Weapon[] weapons

getWeapon()
useWeapon()
increasePower()
// more
```

假设有精灵、矮人、  
和魔法师等3种人物  
要记录：

power, 50  
type: Elf  
weapons: bow,  
sword, dust

对象

power, 200  
type: Troll  
weapons: bare  
hands, big ax

对象

power, 120  
type: Magician  
weapons: spells,  
invisibility

对象

序列化的文件是很难让一般人阅读的，  
但它比纯文本文件更容易让程序恢复。  
这3种人物的状态，也比较安全，因为一般人  
不会知道要如何动手脚改数据。

## 将序列化对象写入文件

下面是将对象序列化（存储）的方法步骤。不用硬记下来，后面的内容会有详细的说明。

如果文件不存在，它会自动被  
创建出来

### 1 创建出 FileOutputStream

```
FileOutputStream fileStream = new FileOutputStream("MyGame.ser");
```

↑  
创建存取文件的  
FileOutputStream 对象

### 2 创建 ObjectOutputStream

```
ObjectOutputStream os = new ObjectOutputStream(fileStream);
```

↑  
它能让你写入对象，但无法直接地连  
接文件，所以需要参数的指引

### 3 写入对象

```
os.writeObject(characterOne);
os.writeObject(characterTwo);
os.writeObject(characterThree);
```

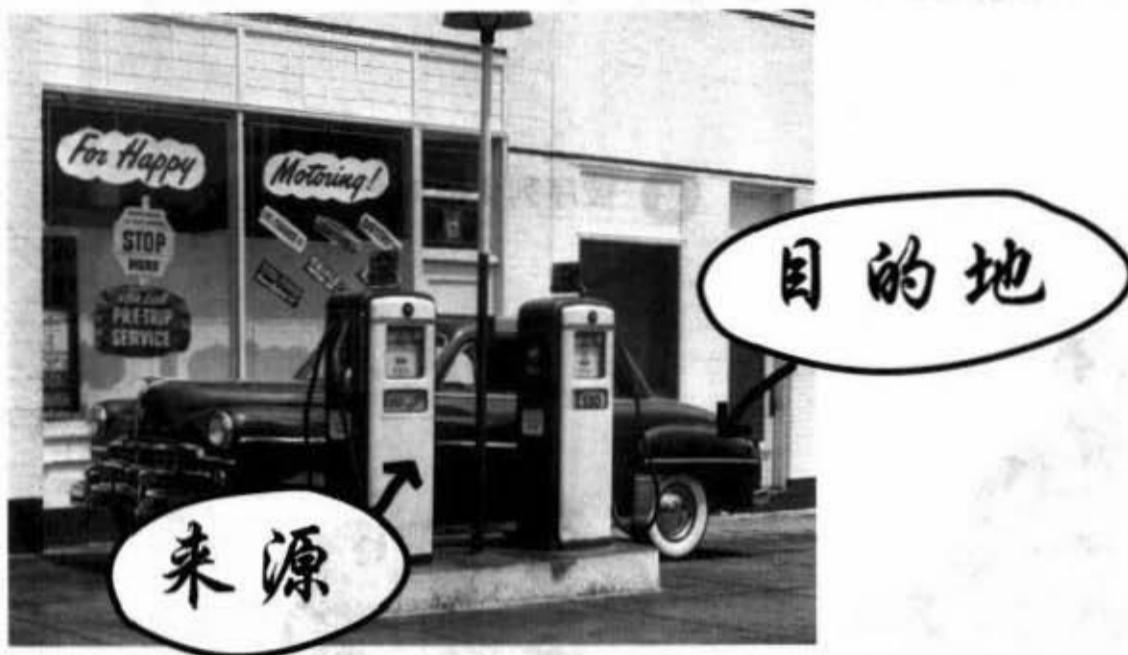
↑  
将变量所引用的对象序列化并写入  
MyGame.ser 这个文件

### 4 关闭 ObjectOutputStream

```
os.close();
```

↑  
关闭所关联的输出串流

## 数据在串流中移动



将串流 (stream) 连接起来代表来源与目的地 (文件或网络端口) 的连接。串流必须要连接到某处才能算是个串流。

Java的输入/输出API带有连接类型的串流，它代表来源与目的地之间的连接，连接串流将串流与其他串流连接起来。

一般来说，串流要两两连接才能作出有意义的事情——其中一个表示连接，另一个则是要被调用方法的。为何要两个？因为连接的串流通常都是很低层的。以`FileOutputStream`为例，它有可以写入字节的方法。但我们通常不会直接写字节，而是以对象层次的观点来写入，所以需要高层的连接串流。

那又为何不以单一的串流来执行呢？这就要考虑到良好的面向对象设计了。每个类只要做好一件事。`FileOutputStream`把字节写入文件。`ObjectOutputStream`把对象转换成可以写入串流的数据。当我们调用`ObjectOutputStream`的`writeObject`时，对象会被打成串流送到`FileOutputStream`来写入文件。

这样就可以通过不同的组合来达到最大的适应性！如果只有一种串流类的话，你只好祈祷API的设计人已经想好所有可能的排列组合。但通过链接的方式，你可以自由地安排串流的组合与去向。



## 对象被序列化的时候发生了什么事？

1 在堆上的对象



在堆上的对象有状态——实例变量的值。这些值让同一类的不同实例有不同的意义。

2 被序列化的对象

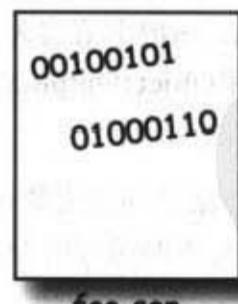


序列化的对象保存了实例变量的值，因此之后可以在堆上带回一模一样的实例。



```
Foo myFoo = new Foo();
myFoo.setWidth(37);
myFoo.setHeight(70);
```

值被抽出送到 stream 上



宽度与高度的实例变量值与 Java 虚拟机所需的信息（像是类的名称）被保存在 foo.ser 文件中

```
FileOutputStream fs = new FileOutputStream("foo.ser");
ObjectOutputStream os = new ObjectOutputStream(fs);
os.writeObject(myFoo);
```

创建出 FileOutputStream 链接到  
ObjectOutputStream 以让它写入对  
象

## 对象的状态是什么？ 有什么需要保存？

事情开始有趣了。存储primitive主数据类型值37和70是很简单的。但如果对象有引用到其他对象的实例变量时要怎么办？如果这些对象还带有其他对象又该如何？

想想看吧。对象基本上有哪些部分是独特的？有哪些东西需要被带回来才能让对象回到和存储时完全相同的状态？当然它会有不同的内存位置，但这无关紧要。我们在乎的是堆上是否有与存储时一模一样的对象状态。



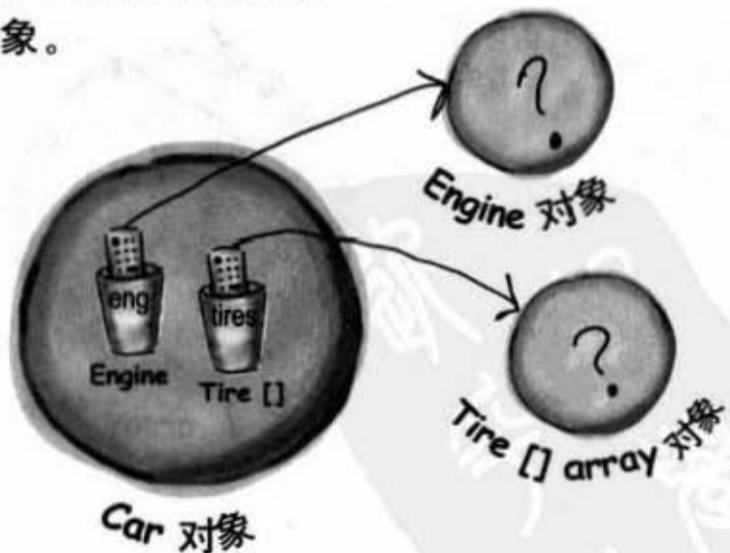
### Brain Barbell

要怎样才能让Car对象能够存储成可以恢复原始状态的形式？

考虑一下有哪些东西是必要保存的。

如果引擎对象还有对汽缸的引用时该如何？而Tire数组对象里面会有什么东西？

Car对象有两个实例变量引用到其他的对象。



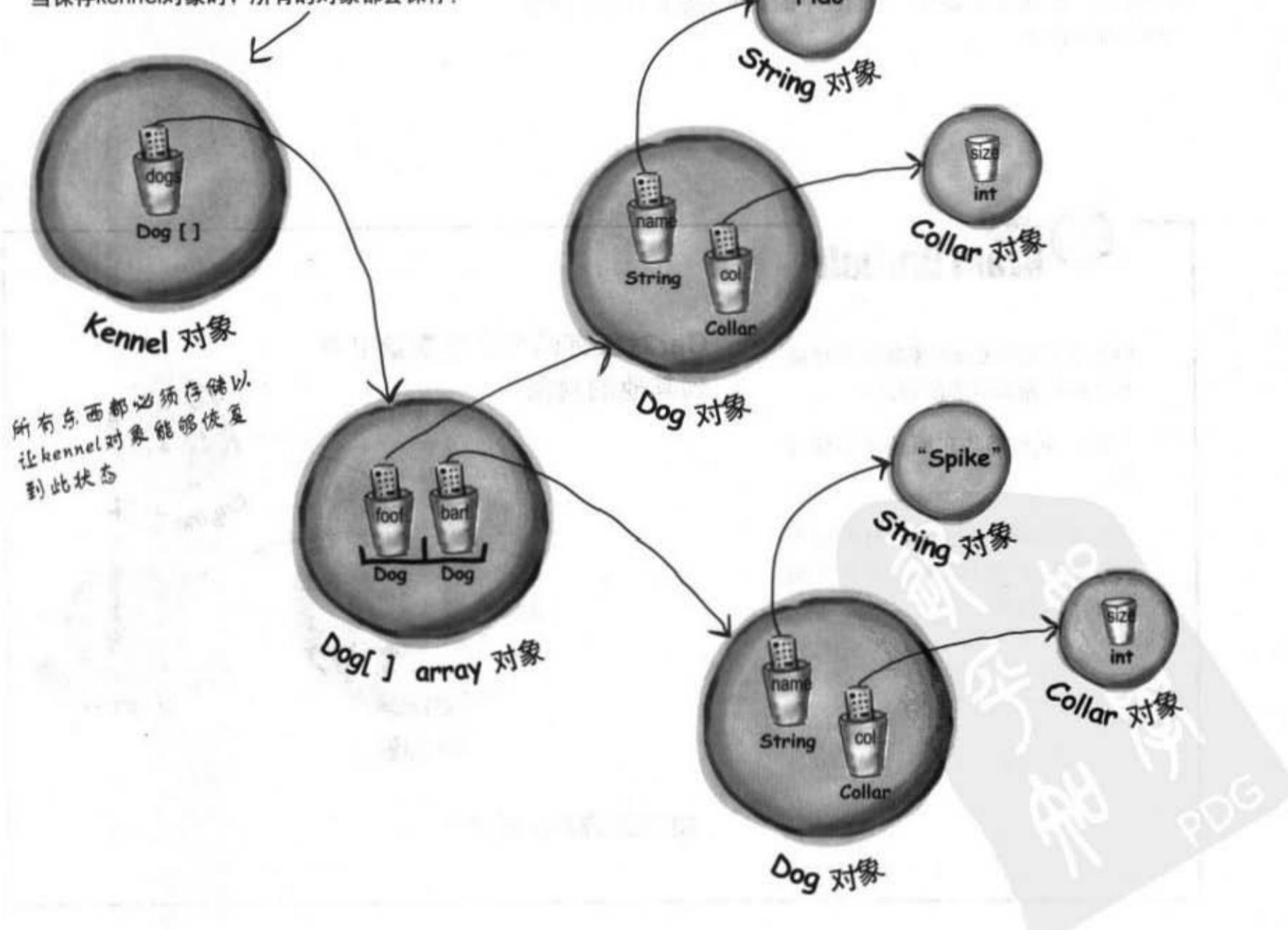
如何保存Car对象？

当对象被序列化时，被该对象引用的实例变量也会被序列化。且所有被引用的对象也会被序列化……最棒的是，这些操作都是自动进行的！

Kernel对象带有对Dog数组对象的引用。Dog[]中有两个Dog对象的引用。每个Dog对象带有String和Collar对象的引用。String对象维护字符的集合，而Collar对象持有一个int。

序列化程序会将对象版图上的所有东西存储起来。被对象的实例变量所引用的所有对象都会被序列化。

当保存kernel对象时，所有的对象都会保存！



# 如果要让类能够被序列化，就实现Serializable

Serializable接口又被称为marker或tag类的标记用接口，因为此接口并没有任何方法需要实现的。它的唯一目的就是声明有实现它的类是可以被序列化的。也就是说，此类型的对象可以通过序列化的机制来存储。如果某类是可序列化的，则它的子类也自动地可以序列化（接口的本意就是如此）。

```
objectOutputStream.writeObject(myBox);
```

任何放在此处的对象都必须实现序列化，否则在执行期一定会出问题

```

import java.io.*; ← 必须要import它

public class Box implements Serializable { ← 没有方法需要被实现，只是用来告诉Java虚拟机它可以被序列化

 private int width; ← 这些值会被保存
 private int height;

 public void setWidth(int w) {
 width = w;
 }

 public void setHeight(int h) {
 height = h;
 }

 public static void main (String[] args) {

 Box myBox = new Box();
 myBox.setWidth(50);
 myBox.setHeight(20); ← 可能会抛出异常

 try {
 FileOutputStream fs = new FileOutputStream("foo.ser");
 ObjectOutputStream os = new ObjectOutputStream(fs);
 os.writeObject(myBox);
 os.close();
 } catch(Exception ex) {
 ex.printStackTrace();
 }
 }
}
```

如果不存在就会被创建出来

设定链接

序列化是全有或全无的

你能想象只有部分状态被正确保存的下场吗？



哇哇哇哇……光想想就吓死我了。  
如果Cat恢复的时候没有把嘴巴弄回来怎么办？那不就成了哈啰猫吗？  
好恐怖！

整个对象版图都必须正确地序列化，不然就得全部失败。

如果Duck对象不能序列化，Pond对象就不能被序列化。

```

import java.io.*;
public class Pond implements Serializable {
 private Duck duck = new Duck(); ← Pond对象可被序列化
 public static void main (String[] args) {
 Pond myPond = new Pond();
 try {
 FileOutputStream fs = new FileOutputStream("Pond.ser");
 ObjectOutputStream os = new ObjectOutputStream(fs);
 os.writeObject(myPond); ← 将myPond序列化的同时Duck也会被
 os.close();
 } catch (Exception ex) {
 ex.printStackTrace();
 }
 }
 public class Duck {
 // duck 的程序代码
 }
}

```

它有个Duck实例变量

序列化

执行main()的下场：

```

File Edit Window Help Regret
% java Pond
java.io.NotSerializableException: Duck
at Pond.main(Pond.java:13)

```



如果某实例变量不能或不应该被序列化，就把它标记为transient（瞬时）的

如果你需要序列化程序能够跳过某个实例变量，就把它标记成transient的变量

```
import java.net.*;
class Chat implements Serializable {
 这会将此变量标记为不需要序列化的 → transient String currentID;
 这个变量会被序列化 → String userName;
 // 还有更多程序代码.....
}
```

如果你有无法序列化的变量不能被存储，可以用transient这个关键词把它标记出来，序列化程序会把它跳过。

为什么有些变量不能被序列化？可能是设计者忘记实现Serializable。或者动态数据只可以在执行时求出而不能或不必存储。虽然Java函数库中大部分的类可以被序列化，你还是无法将网络联机之类的东西保存下来。它得要在执行期当场创建才有意义。一旦程序关闭之后，联机本身不再有用，下次执行时需要重新创建出来。

*there are no*  
**Dumb Questions**

**问：**如果序列化这么重要，为什么不默认成每个类都有？为什么不让Object实现Serializable以让所有类都自动地成为可序列化的？

**答：**就算大部分的类都应该实现Serializable，你还是有选择性。且你必须有意识地对各个类做决定。如果它变成默认的，那要如何关掉呢？毕竟界面是用来指示功能性而不是所缺乏的功能。

**问：**为什么我写过不需要序列化的类？

**答：**也许是因为安全性的理由让你不想把密码对象存储在文件上。或者某些对象的存储是没有意义的，所以你不会把它实现成可序列化的。这个问题要问施主你自己。

**问：**为什么我爸爸不是李嘉诚？为什么我长得这么帅却是秃头？

**答：**这个问题要问施主你自己。

**问：**如果我使用了一个不能序列化的类，我是否能把它给子类出一个标记为可序列化的类？

**答：**可以！如果该类是可以被继承（没有被标记为final），你就可以制作出可被序列化的子类。但这又带来另外一个问题：为什么它一开始不是可序列化的？

**问：**这位同学你问得很好，为什么会有不可序列化类的可序列化子类？

**答：**这，这，这……首先要看类被还原的时候会发生什么事（稍后会讨论）。简单讲，当对象被还原且它的父类不可序列化时，父类的构造函数会跟创建新的对象一样地执行。如果类没有什么好理由不能被序列化，制作可序列化的子类会是个好方法。

**问：**我现在才了解，如果你将某个变量标记为transient的，那就代表在序列化的过程中该变量会被略过。然后会发生什么事？我们用transient标记变量来解决不能序列化的实例变量问题，但我们不是在回复对象的时候需要该变量吗？序列化的重点不就在于保存对象的状态吗？

**答：**没错！问题就在这里，但是幸好有解决方法。如果你把某个对象序列化，transient的引用实例变量会以null返回，而不管存储当时它的值是什么。这代表整个对象版图中连接到该特定实例变量的部分不会被存储。

这样可能会有问题，所以我们有两个解决方案：

(1) 当对象被带回来的时候，重新初始化实例变量回到默认的状态。例如Dog可能会有Collar，但因为Collar无关紧要，所以再个新的给Dog也不会影响程序逻辑。

(2) 如果transient变量的值很重要，例如Collar的颜色是有意义且每个Dog不一样的，你就得需要同时把它的值也保存下来。然后在带回Dog对象时重新创建Collar，再把颜色值设定给Collar。

**问：**如果两个对象都有引用实例变量指向相同的对象会怎样？例如两个Cat都有相同的Owner对象？那Owner会被存储两次吗？

**答：**好问题！序列化聪明得足以分辨两个对象是否相同。在此情况下只有一个对象会被存储，其他引用会复原成指向该对象。

**问：**你最喜欢什么课？

**答：**数学课。因为数学老师常常请假。

## 解序列化 (Deserialization) : 还原对象

将对象序列化整件事情的重点在于你可以在事后，在不同的Java虚拟机执行期（甚至不是同一个Java虚拟机），把对象恢复到存储时的状态。解序列化有点像是序列化的反向操作。



如果文件不存在  
就会抛出异常

### 1 创建 FileInputStream

```
FileInputStream fileStream = new FileInputStream("MyGame.ser");
```

它知道如何连接文件

### 2 创建 ObjectInputStream

```
ObjectInputStream os = new ObjectInputStream(fileStream);
```

它知道如何读取对  
象，但是要靠链接的  
stream提供文件存取

### 3 读取对象

```
Object one = os.readObject();
Object two = os.readObject();
Object three = os.readObject();
```

每次调用readObject()都会从stream中读出下  
一个对象，读取顺序与写入顺序相同，次  
数超过会抛出异常

### 4 转换对象类型

```
GameCharacter elf = (GameCharacter) one;
GameCharacter troll = (GameCharacter) two;
GameCharacter magician = (GameCharacter) three;
```

返回值是Object类型，因此  
必须要转换类型

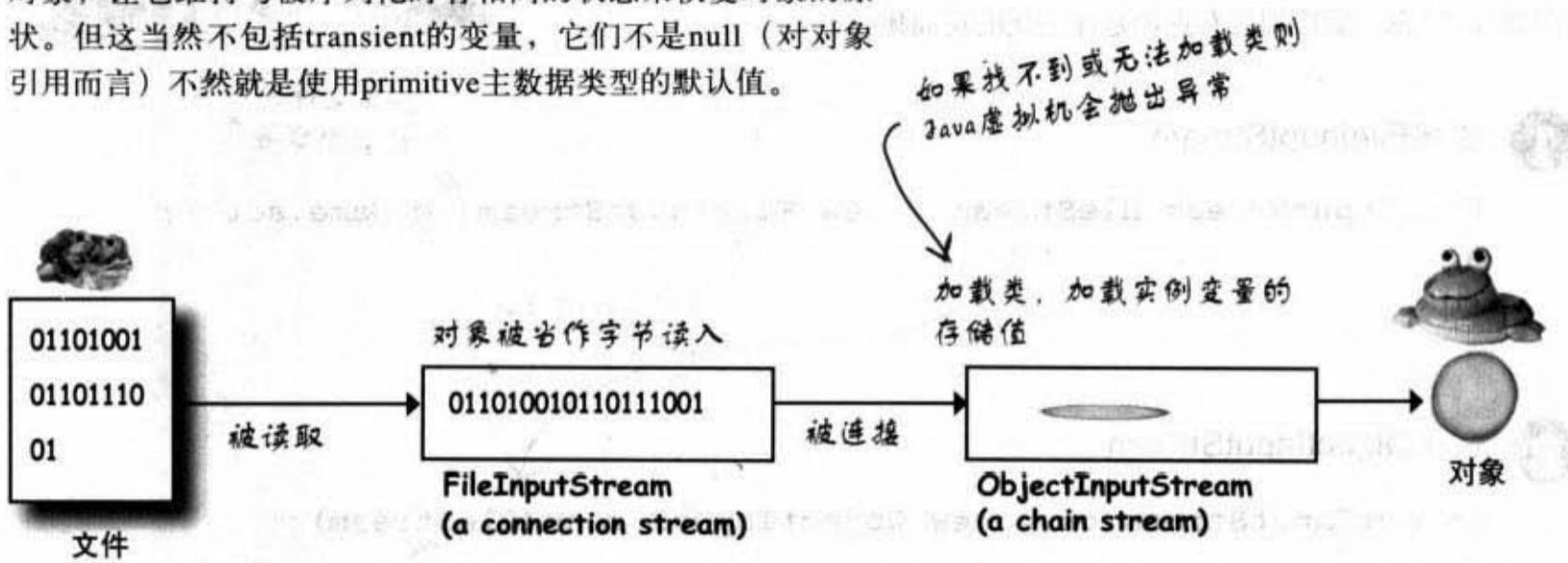
### 5 关闭 ObjectInputStream

```
os.close();
```

FileInputStream会自动跟着关闭

## 解序列化的时候发生了什么事？

当对象被解序列化时，Java虚拟机会通过尝试在堆上创建新的对象，让它维持与被序列化时有相同的状态来恢复对象的原状。但这当然不包括transient的变量，它们不是null（对对象引用而言）不然就是使用primitive主数据类型的默认值。



- ① 对象从stream中读出来。
- ② Java虚拟机通过存储的信息判断出对象的class类型。
- ③ Java虚拟机尝试寻找和加载对象的类。如果Java虚拟机找不到或无法加载该类，则Java虚拟机会抛出例外。
- ④ 新的对象会被配置在堆上，但构造函数不会执行！  
很显然的，这样会把对象的状态抹去又变成全新的，  
而这不是我们想要的结果。我们需要的是对象回到  
存储时的状态。

**5** 如果对象在继承树上有个不可序列化的祖先类，则该不可序列化类以及在它之上的类的构造函数（就算是可序列化也一样）就会执行。一旦构造函数连锁启动之后将无法停止。也就是说，从第一个不可序列化的父类开始，全部都会重新初始状态。

**6** 对象的实例变量会被还原成序列化时点的状态值。`transient`变量会被赋值`null`的对象引用或`primitive`主数据类型的默认为`0`、`false`等值。

there are no  
Dumb Questions

**问：**为什么类不会存储成对象的一部分？这样就不会出现找不到类的问题了？

**答：**当然也可以设计成这个样子，但这样是非常的浪费且会有很多额外的工作。虽然把对象序列化写在本机的硬盘上面不是什么很困难的工作，但序列化也有将对象送到网络联机上的用途。如果每个序列化对象都带有类，带宽的消耗可能就是个大问题。

对于通过网络传递序列化对象来说，事实上是有一种机制可以让类使用URL来指定位置。该机制用在Java的Remote Method Invocation (RMI，远程程序调用机制)，让你可以把序列化的对象当作参数的一部分来传递。若接收此调用的Java虚拟机没有这个类的话，它可以自动地使用URL来取回并加载该类（第17章会讨论RMI）。

**问：**那静态变量呢？它们会被序列化吗？

**答：**不会。要记得`static`代表“每个类一个”而不是“每个对象一个”。当对象被还原时，静态变量会维持类中原本的样子，而不是存储时的样子。

## 存储与恢复游戏人物

```

import java.io.*;

public class GameSaverTest {
 public static void main(String[] args) {
 GameCharacter one = new GameCharacter(50, "Elf", new String[] {"bow", "sword", "dust"});
 GameCharacter two = new GameCharacter(200, "Troll", new String[] {"bare hands", "big ax"});
 GameCharacter three = new GameCharacter(120, "Magician", new String[] {"spells", "invisibility"});

 // 假设此处有改变人物状态值的程序代码

 try {
 ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream("Game.ser"));
 os.writeObject(one);
 os.writeObject(two);
 os.writeObject(three);
 os.close();
 } catch (IOException ex) {
 ex.printStackTrace();
 }
 one = null; // 设定成null, 因此无法存取堆上的
 two = null; // 这些对象
 three = null;
 }

 try {
 ObjectInputStream is = new ObjectInputStream(new FileInputStream("Game.ser"));
 GameCharacter oneRestore = (GameCharacter) is.readObject();
 GameCharacter twoRestore = (GameCharacter) is.readObject();
 GameCharacter threeRestore = (GameCharacter) is.readObject();

 System.out.println("One's type: " + oneRestore.getType()); // 看看是否成功
 System.out.println("Two's type: " + twoRestore.getType());
 System.out.println("Three's type: " + threeRestore.getType());
 } catch (Exception ex) {
 ex.printStackTrace();
 }
}

```

创建人物……

再从文件中把对象读回来

对象

对象

对象

## GameCharacter类

```

import java.io.*;

public class GameCharacter implements Serializable {
 int power;
 String type;
 String[] weapons;

 public GameCharacter(int p, String t, String[] w) {
 power = p;
 type = t;
 weapons = w;
 }

 public int getPower() {
 return power;
 }

 public String getType() {
 return type;
 }

 public String getWeapons() {
 String weaponList = "";

 for (int i = 0; i < weapons.length; i++) {
 weaponList += weapons[i] + " ";
 }
 return weaponList;
 }
}

```

这是个测试序列化用的类，并没有实际的游戏功能

# 对象的序列化

## 要点

- 你可以通过序列化来存储对象的状态。
- 使用ObjectOutputStream来序列化对象（java.io）。
- Stream是连接串流或是链接用的串流。
- 连接串流用来表示源或目的地、文件、网络套接字连接。
- 链接用串流用来衔接连接串流。
- 用FileOutputStream链接ObjectOutputStream来将对象序列化到文件上。
- 调用ObjectOutputStream的writeObject(theObject)来将对象序列化，不需调用FileOutputStream的方法。
- 对象必须实现序列化这个接口才能被序列化。如果父类实现序列化，则子类也就自动地有实现，而不管是否有明确的声明。
- 当对象被序列化时，整个对象版图都会被序列化。这代表它的实例变量所引用的对象也会被序列化。
- 如果有不能序列化的对象，执行期间就会抛出异常。
- 除非该实例变量被标记为transient。否则，该变量在还原的时候会被赋予null或primitive主数据类型的默认值。
- 在解序列化时（deserialization），所有的类都必须能让Java虚拟机找到。
- 读取对象的顺序必须与写入的顺序相同。
- readObject()的返回类型是Object，因此解序列化回来的对象还需要转换成原来的类型。
- 静态变量不会被序列化，因为所有对象都是共享同一份静态变量值。

## 将字符串写入文本文件

通过序列化来存储对象是Java程序在来回执行间存储和恢复数据最简单的方式。但有时你还得把数据存储到单纯的文本文件中。假设你的Java程序必须把数据写到文本文件中以让其他可能是非Java的程序读取。例如你的servlet（在Web服务器上执行的Java程序）会读取用户在网页上输入的数据，并将它写入文本文件以让网站管理人能够用电子表格来分析数据。

写入文本数据（字符串）与写入对象是很类似的，你可以使用`FileWriter`来代替`OutputStream`（当然不会把它链接到`ObjectOutputStream`上）。

如果游戏人物数据写成常人可识别的文本文件大概就会像这样：

```
50,Elf,bow,sword,dust
200,Troll,bare hands,big ax
120,Magician,spells,invisibility
```

写序列化的对象：

```
objectOutputStream.writeObject(someObject);
```

写字符串：

```
fileWriter.write("My first String to save");
```

```
import java.io.*; ← 需要加载这个包

class WriteAFile {
 public static void main (String[] args) {
 try {
 FileWriter writer = new FileWriter("Foo.txt");
 writer.write("hello foo!"); ← 以字符串作参数
 writer.close(); ← 记得要关掉
 } catch (IOException ex) {
 ex.printStackTrace();
 }
 }
}
```

输入/输出相关的操作都必须包在try/catch块中

如果不存在就  
会被创建

写入文本文件

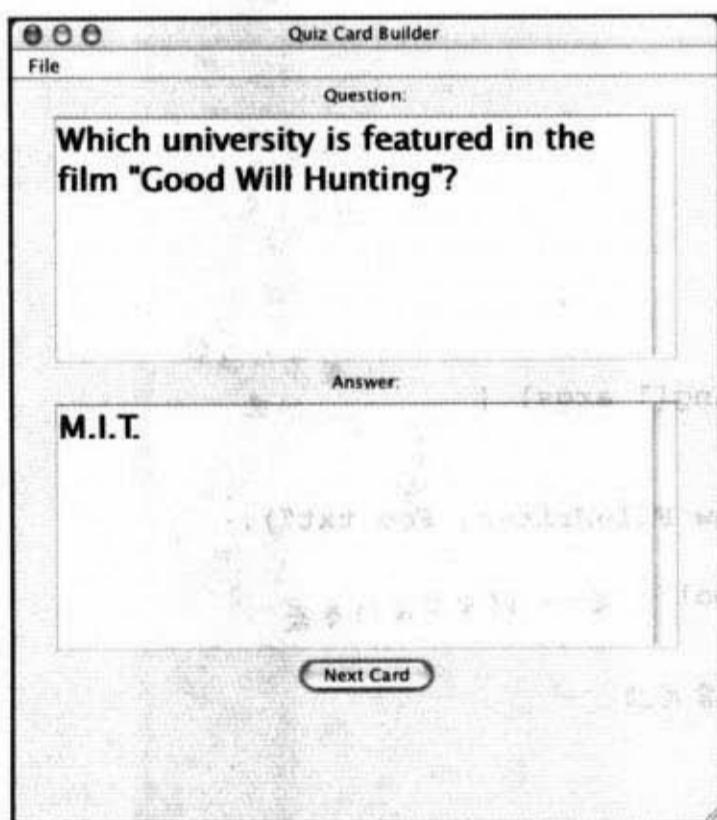
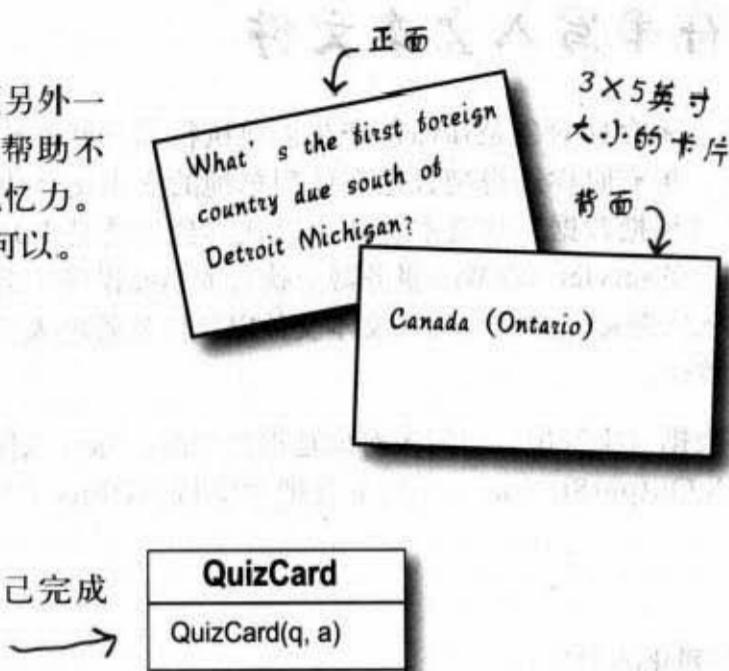
## 文本文件范例：e-Flashcard

你知道老外在学校用的flashcard吗？那是一面有问题另外一面有答案的卡片。这种卡片对于理解式学习效果的帮助不大，但没有其他道具会比它还能加强机械化死背的记忆力。如果你想要来硬的，这倒还不错。拿来打发时间也还可以。

我们要用3个类来写个电子版flashcard：

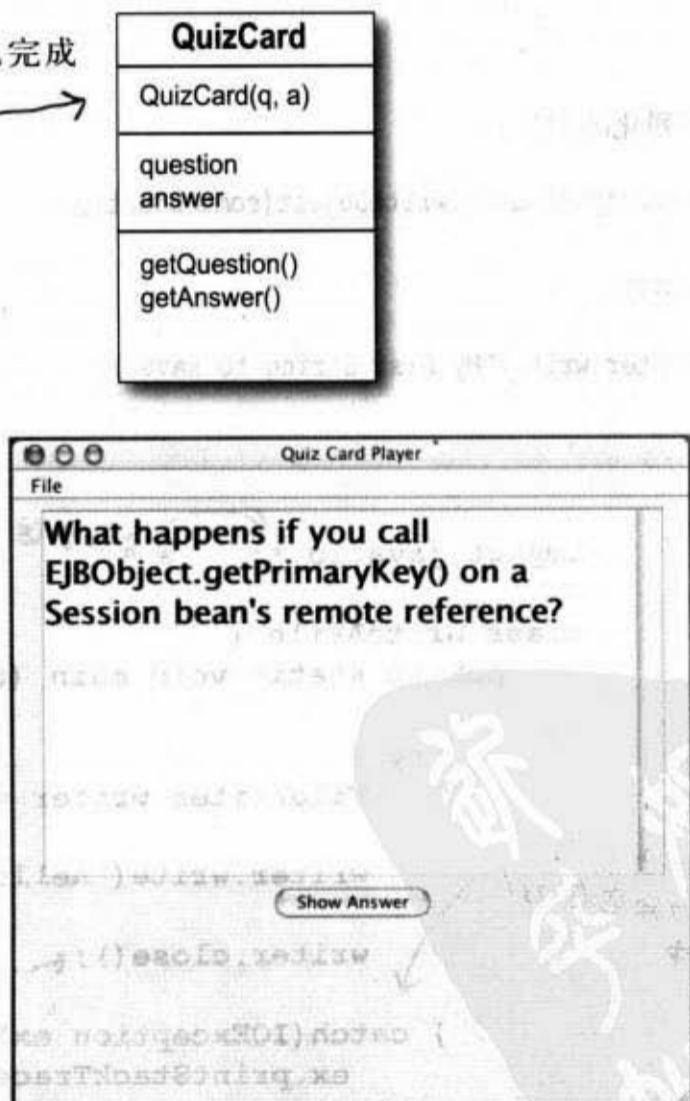
- (1) QuizCardBuilder, 设计并存储卡片的工具。
- (2) QuizCardPlayer, 加载并播放卡片的引擎。
- (3) QuizCard, 表示卡片数据的类。

我们会查看builder和player的程序代码，并让你自己完成QuizCard，它会写成右边这样。



QuizCardBuilder

File菜单下有个Save选项能够让你存储一组数据到文本文件中。



QuizCardPlayer

File菜单下有个Load选项能够从文本文件加载一组卡片数据。

## Quiz Card Builder (程序代码大纲)

```

public class QuizCardBuilder {
 public void go() {
 // 创建并显示gui
 }
 内部类
 private class NextCardListener implements ActionListener {
 public void actionPerformed(ActionEvent ev) {
 // 向列表中增加当前卡片并清除文本域
 }
 }
 内部类
 private class SaveMenuListener implements ActionListener {
 public void actionPerformed(ActionEvent ev) {
 // 生成对话框
 // 输入用户名并保存设置
 }
 }
 内部类
 private class NewMenuListener implements ActionListener {
 public void actionPerformed(ActionEvent ev) {
 // 清除card列表和文本域
 }
 }
 内部类
 private void saveFile(File file) {
 // 把列表输出到一个文本文件
 }
}

```

创建并显示GUI，包括事件  
监听者的设置和注册

按下“Next Card”时会被触发。  
代表用户完成卡片并继续下一张  
新卡片

会被菜单上的“Save”触发，代  
表用户想要存储目前这一组卡  
片

会被菜单上的“New”触发，这  
会打开新的一组卡片（还要清空  
文字块）

实际编写文件的程序

## Quiz Card Builder 代码

```
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

public class QuizCardBuilder {

 private JTextArea question;
 private JTextArea answer;
 private ArrayList<QuizCard> cardList;
 private JFrame frame;

 public static void main (String[] args) {
 QuizCardBuilder builder = new QuizCardBuilder();
 builder.go();
 }

 public void go() {
 // 创建gui

 frame = new JFrame("Quiz Card Builder");
 JPanel mainPanel = new JPanel();
 Font bigFont = new Font("sanserif", Font.BOLD, 24);
 question = new JTextArea(6,20);
 question.setLineWrap(true);
 question.setWrapStyleWord(true);
 question.setFont(bigFont);

 JScrollPane qScroller = new JScrollPane(question);
 qScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
 qScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

 answer = new JTextArea(6,20);
 answer.setLineWrap(true);
 answer.setWrapStyleWord(true);
 answer.setFont(bigFont);

 JScrollPane aScroller = new JScrollPane(answer);
 aScroller.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
 aScroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

 JButton nextButton = new JButton("Next Card");
 cardList = new ArrayList<QuizCard>();

 JLabel qLabel = new JLabel("Question:");
 JLabel aLabel = new JLabel("Answer:");

 mainPanel.add(qLabel);
 mainPanel.add(qScroller);
 mainPanel.add(aLabel);
 mainPanel.add(aScroller);
 mainPanel.add(nextButton);
 nextButton.addActionListener(new NextCardListener());
 JMenuBar menuBar = new JMenuBar();
 JMenu fileMenu = new JMenu("File");
 JMenuItem newMenuItem = new JMenuItem("New");
 }
}
```

所有的GUI程序都在这里。  
也许你要多注意一下有关  
Menu菜单的部分