笔记6

第六章 接口与内部类

接口这种技术主要用来描述类有什么功能的 而并不给出每个功能的具体实现 一个类可以实现一个或多个接口
并在接口的地方 随时使用实现了相应接口的对象

克隆对象 对象的克隆是指创建一个新对象 且新对象的状态与原始对象的状态相同 当对克隆的新对象进行修改时 不会影响原始对象的状态

内部类机制 内部类定义在另外一个类的内部 其中的方法可以访问他们外部类的域 这是一项比较复杂的技术 **内部类技术主要用于设计具有相互协作关系的类集合** 特别是在编写GUI事件的代码时 使用它将可以让代码看起来更加简练专业

最后是代理 这是一种实现任意接口的对象 代理是一种非常专业的构造工具

接口

接口不是类 而是对类的一组需求的描述 这些类要遵守接口描述的统一格式进行定义
在很多的服务提供商中经常有这么一句话 如果类遵循某个特定接口 那么就履行这项义务
比如Arrays类的sort方法承诺可以对数组进行排序 但是要求满足如下前提 对象所属的类必须实现了Compareable接口 这个接口中定义的compareTo方法是不能比较浮点值的

接口中的所有方法自动属于public 因此 在接口中声明方法时 不比提供关键字pu blic
接口可以定义多个方法 接口不能包含实例域 也不能在接口中实现方法 这个任务应该由实现接口的类来实现

因此 可以将接口看成没有实例域的抽象类 但是抽象类是只能被类继承一个的 灵活性较小 接口是可以声明多个接口的

让一个类实现一个接口 有如下俩个步骤
1 将类声明为实现给定的接口
2 对接口中的所有方法进行定义
接口关键字implements

至于为什么要实现一个Comparable接口才能进行排序的原因是 Java是一种强类型的语言 在**调用方法的时候 编译器将会检查这个方法是否存在**

比如sort方法中可能存在如下的语句
if(a[i].compareTo(a[j]) >0)
{
// rearrange a[i] and a[j]
}

编译器必须确认a[i]一定存在compareTo方法 如果a是一个Comparable对象的数组 就可以确保拥有CompareTo方法 因为每个实现Compareable的接口就必须要实现这个方法

注意：在语言标准中 compareTo的比较方式有可能出现异常
在compareTo中 对于任意的x和y 实现都必须能够保证sgn(x.compareTo(y))=-sgn(y.compareTo(x)) 这里的sgn是一个数值的符号
因为在继承过程中有可能出现问题

比如 如下：

```
public class Manager extends Employee {



public  int compareTo(Employee other) {
Manager otherManager = (Manager) other;
//....
}
```

这就是不符合反对称规则的 如果x是一个Employee对象 y是一个Manager对象 调用x.compareTo(y)不会抛出异常 它只是将x与y都作为雇员进行比较 但是反过来 y.compareTo(x)将会抛出一个ClassCastException

这个跟第五章的equals方法一样 修改的方式也一样
有俩种情况

1 如果子类之间的比较含义不一样 那就属于不同类对象的非法比较 每个compareTo方法都应该在开始前进行检测
if(getClass !=other.getClass())throw new ClassCastException();

如果存在这一种通用算法 它能够对俩个不同的子类对象进行比较 则应该在超类提供一个compareTo方法 并将这个方法声明为final

比如：不管薪水多少都想让经理大于雇员的话 如果一定用按照职务排列的话 那就应该在Employee类中提供一个rank方法 每个子类覆盖rank，并实现一个比较rank值的compareTo方法

接口的特性

接口是不能new出一个实例的 但是可以声明接口的变量
接口变量必须引用实现了接口的类对象
与类的继承一样 接口也可以被扩展 这里允许存在多条从具有较高通用性接口到较高专用性接口的链
就是如下所示
public interface A{

void a1（）；
}
public interface B extends A{
void b1（）；
}

接口可以包含常量 接口中的变量自动被设置为public static final

每个类都只能拥有一个超类 但是可以有多个接口

对象克隆

原始变量和拷贝变量引用同一对象

如果想创建一个对象的copy 它的最初对象是与原型对象是一样的 但是以后可以各自改变各自的状态 那就使用clone方法
这个clone方法是Object中的protected方法
也就是说 在用户编写的代码中不能直接调用它 只有Employee类才能克隆 由于这个类对具体的类对象一无所知
所以只能讲各个域进行对应的拷贝 如果所有的数据源都是数值或基本类型 拷贝没有任何问题 但是如果在对象中包含了子对象的引用 拷贝的结果就会使得俩个域引用同一个子对象 原始对象与克隆对象共享这部分信息

默认的克隆操作是浅拷贝 并没有克隆包含对象中的内部信息
浅拷贝会发生什么 如果原始对象与浅克隆对象共享的子对象是**不可变的 将不会尝试任何问题**
更常见的是子对象可变 因此必须重新定义clone方法 以便实现克隆子对象的深拷贝
对于每一个类
：
1 默认的clone方法是否满足要求
2 默认的clone方法是否能够通过调用可变子对象的clone得到修补
3 是否不应该使用clone
实际上 选项3是默认的 如果要选择1和2 类必须：
1 实现Cloneable接口
2 使用public访问修饰符重新定义clone方法

如果想要所以的方法都可以克隆对象 就必须重新实现clone方法 改为public
Cloneable接口的出现于接口的正常使用没有任何关系 这个接口在这里只是做为一个标记 表明类设计者知道要进行克隆处理 如果一个对象需要克隆 但是没有实现Cloneable接口 就会产生一个已检测异常

即使clone的默认实现（浅拷贝）能满足需求 也应该实现Cloneable接口 将clone重定义为public 并调用
super.clone()
如下：

```
public  Employee clone () throws CloneNotSupportedException{
return (Employee) super.clone();
}
```

但是为了实现深拷贝 必须克隆所有可变的实例域

```
public  Employee clone () throws CloneNotSupportedException {
```

```
Employee cloned = (Employee) super.clone();
cloned.hireDay = (Date) hireDay.clone();
return cloned;
}
```

只用clone中含有没有实现Cloneable接口的对象 Object的clone方法就会抛出一个CloneNot_SuppertException 异常
如果是替换成捕获异常的话

```
public  Employee clone ()   {
try {
return (Employee) super.clone();
    } catch (CloneNotSupportedException e) {
e.printStackTrace();
    }
}
```

这种写法是比较适合final类 因为无法继承 但是最好还是在这个地方保留throws说明符 如果不支持克隆 子类需要具有抛出CloneNotSuppertException的选择权

必须谨慎的实现子类的克隆 比如 一旦为Employee类定义了clone方法 任何人都可以利用它克隆Manager对象 Empoyee的克隆不一定能够完成这个任务 这将取决于Manager类中包含哪些域 如果Manager中有一些需要深拷贝的域的话 或者已一些没有实现Cloneable接口的域 没有人能够保证子类实现的clone方法一定争取
在自定义的类中是否实现clone方法 如果客户需要深拷贝就应该实现它 克隆不怎么使用。。

接口与回调

回调是一种常见的设计模式 在这种模式中 可以指出某个特定事件发生时应该采取的动作

内部类

内部类方法可以定义在访问在该类定义所在的作用域中的数据 包括私有的数据
内部类可以对同一个包中的其他类隐藏起来
当想要定义一个回调函数且不想编写大量代码时 使用匿名内部类比较便捷
4.1节 给出一个简单的内部类 它将访问外部类的实例域
4.2节 给出内部类的特殊语法规则
4.3节 领略一下内部类的内部探讨一下如果将其转换为常规类
4.4节 讨论局部内部类 它可以访问作用域中的局部变量
4.5节 介绍匿名内部类 说明用于实现回调的基本方法

使用内部类访问对象状态

内部类即可以访问自身的数据域 也可以访问创建它的外围类对象的数据域

外围类的引用在构造器中设置 编译器修改了所有内部类的构造器 添加一个外围类引用的参数

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import  javax.swing.Timer;
/**
 * Created by han on 2016/8/14.
 */
public class InnerClassTest {
public static void main(String[] args) {
TakingClock clock = new TakingClock(1000, true);
clock.start();

JOptionPane.showMessageDialog(null, "Quit program?");
System.exit(0);
    }
}
class TakingClock{

private  int interval;
private  boolean beep;
```

```
public TakingClock( int interval,boolean beep) {
this.beep = beep;
this.interval = interval;
    }


public  void start() {

ActionListener listener = new TimePrinter();
Timer t = new Timer(interval, listener);
t.start();
    }

public  class TimePrinter implements  ActionListener{


@Override
public void actionPerformed(ActionEvent e) {
Date now = new Date();
System.out.println("At the tone ,the time is" + now);
if(beep) Toolkit.getDefaultToolkit().beep();
        }
    }
}
```

内部类的特殊语法规则

注意下面的代码

```
if(beep) Toolkit.getDefaultToolkit().beep();
```

在 内部类的特殊语法规则下是可以变为

```
if(TakingClock.this.beep) Toolkit.getDefaultToolkit().beep();
```

可以采用下列语法格式更加明确的编写内部对象的构造器
outerObject.new InnerClass(construction parameters);
比如

```
if(TakingClock.this.beep) Toolkit.getDefaultToolkit().beep();
```

通常 this限定词是多余的
不过 **可以显示的地命名将外围类引用设置为其他的对象**

```
TakingClock clock = new TakingClock(1000, true);
TakingClock.TimePrinter l = clock.new TimePrinter();
```

内部类是否有用 必要和安全

内部类的语法很复杂 它与访问控制和安全性等其他语言特性的没有明显的关联

内部类是一种编译现象 与虚拟机无关 编译器将会把内部类翻译成$符合分割外部类名与内部类名的常规类文件 而虚拟机对此一无所知

内部类具有访问外部类的特权 所以与常规类相比功能要更强大

这里有一个命令 这个命令就是用反射查看相关的内部类class文件生成的代码

```
C:\Users\han\Desktop\Test\新建文件夹 (3)\JAVAC编译测试> javap -private TakingClo
ck$TimePrinter
Compiled from "InnerClassTest.java"
public class TakingClock$TimePrinter implements java.awt.event.ActionListener {
  final TakingClock this$0;
  public TakingClock$TimePrinter(TakingClock);
  public void actionPerformed(java.awt.event.ActionEvent);
}
```

大概就是这个样子 **编译器为了引用外围类 生成了一个附加的实例域this$0(名字this$0是由编译器合成的 在自己编写的代码中不能引用它)**

另外 还可以看到构造的TakingClock参数
有人会好奇 假如内部类可以翻译成名字很古怪的常规类（而虚拟机对此并不了解）内部类是如何管理那些额外的访问特权呢

```
}

C:\Users\han\Desktop\Test\新建文件夹 (3)\JAVAC编译测试> javap -private TakingClo
ck
Compiled from "InnerClassTest.java"
class TakingClock {
  private int interval;
  private boolean beep;
  public TakingClock(int, boolean);
  public void start();
  static boolean access$000(TakingClock);
}
```

可以看到如上**编译器在外围类添加静态方法access$000 它将返回作为参数传递给它的对象域beep**
if(beep)
这样做是存在安全威胁的 任何人都可以通过调用access$000 方法很容易地读取私有域beep
黑客的攻击代码需要与被攻击类放在同一个包里 因为访问权限的限制 隐秘的访问方法需要包可见性

总之 如果内部类访问了私有数据源 就有可能通过附加在外围类所在包中的其他类访问它们 但做这些事情需要高超的技巧以及强大的决心
程序员不可能无意之中就获得对类的访问权限 而必须刻意地构建或修改类文件才有可能达到这个目的
合成构造器和方法是复杂令人费解的
假设TimePrinter转换为一个内部类 在虚拟机中不存在私有类 因此编译器将会利用私有构造器生成一个包可见的类
private TalkingClock$TimePrinter(TalkingClock);
当然 没有人可以调用这个构造器 因此 存在第二个包可见构造器
TalkingClock$TimePrinter（TalkingClock，TalkingClock$1）
它将调用第一个构造器
编译器将TalkingClock类start方法中的构造器调用翻译为：
new TalkingClock$TimePrinter（this,null）

局部内部类

如果仔细上面的TakingClock代码就可以发现 TimePrinter这个类名字只在start方法中创建这个类型的对象使用过一次

这样的话 我们是可以将其定义为局部类的

```
public  void start() {

class TimePrinter implements  ActionListener {


@Override

public void actionPerformed(ActionEvent e) {

Date now = new Date();

System.out.println("At the tone ,the time is" + now);

if(TakingClock.this.beep) Toolkit.getDefaultToolkit().beep();

        }

    }

ActionListener listener =  new TimePrinter();

Timer t = new Timer(interval, listener);

t.start();

}
```

局部类是不能使用public 或private访问说明符进行声明 它的作用域被限定在声明这个局部类的块中

局部类有一个优势 即对外部世界可以完全隐藏起来 即使TakingClock类中的其他代码也不能访问它 除start方法只为 没有任何方法知道

TimePrinter类的存在

由外部方法访问final变量

与其他内部类相比较 局部类还有一个优点 它们不仅能够访问包含它们的外部类 还能访问局部变量 不过 那些局部变量必须被声明为final

```
public  void start(int interval,final  boolean beep) {
class TimePrinter implements  ActionListener {

@Override
public void actionPerformed(ActionEvent e) {
Date now = new Date();
System.out.println("At the tone ,the time is" + now);
if(beep) Toolkit.getDefaultToolkit().beep();
        }
    }
ActionListener listener =  new TimePrinter();
Timer t = new Timer(interval, listener);
t.start();
}
```

注意 这样TalkingClock类不再需要存储实例变量beep了 只是引用start方法中的beep参数变量

下面是控制流程
1 调用start方法
2 **调用内部类TimePrinter的构造器 以便初始化对象listener**
3 将listener引用传递给Timer构造器 定时器开始计时 start方法结束 此时 start方法beep参数变量不复存在
4 然后 actionPerformaed方法执行if(beep)..
为了让actionPerformed方法工作 TimerPrinter类在beep域释放之前将beep域用start方法局部变量进行备份

```
C:\Users\han\Desktop\Test\新建文件夹 (3)\JAVAC编译测试>javap -private TakingCloc
k$1TimePrinter.class
Compiled from "InnerClassTest.java"
class TakingClock$1TimePrinter implements java.awt.event.ActionListener {
  final boolean val$beep;
  final TakingClock this$0;
  TakingClock$1TimePrinter();
  public void actionPerformed(java.awt.event.ActionEvent);
}
```

注意构造器的boolean参数和val$beep实例变量 当创建一个对象的时候 beep就会被传递给构造器 并存储在val$beep域中 编译器必须检测对局部变量的访问 为每一个变量建立相应的数据源 并将局部变量拷贝到构造器中 以便将这些数据域初始化为局部变量的副本

从程序员的角度看 局部变量的访问非常容易 ，它减少了需要显示编写的实例域，从而使得内部类更加简单
前面提到过 局部类的方法只可以应用定义为final的局部变量 鉴于此情况 在列举的示例中 将beep参数声明为final 对它进行初始化后不能够再进行修改 因此 就是的局部变量与局部类内建立的拷贝保持一致

注释：final可以用于创建局部变量 实例变量 静态变量 只需要赋值一次 以后再也不能修改它的值 在定义final变量时候 不比进行初始化
定义时没有初始化的final变量通常被称为空final变量
但是有的时候final修饰不太方便 比如
需要一个可变的值 但是final了 就变成了常量 这就不可变 不能进行记录一些事情了
这个时候我们可以有一种小技巧
将本来final int counter ;
counter++ ; //error
使用Integer也是不行的 因为Integer也是可变的
这样的变量转换为
final int[] counter =new int[1];
counter[0]++;
这样final一个counter数组 仅仅表示不可以有让它引用另一种数组 数组中的数据元素可以自由的更改

匿名内部类

将局部内部类的使用再深入一步 假如只创建这个类的一个对象 就不必命名 这种类被称为匿名内部类

```
public  void start(int interval,final  boolean beep) {

ActionListener listener =  new ActionListener() {
@Override
public void actionPerformed(ActionEvent e) {
Date now = new Date();
System.out.println("At the tone ,the time is" + now);
if(beep) Toolkit.getDefaultToolkit().beep();
```

```
        }
    };
Timer t = new Timer(interval, listener);
t.start();
}
```

它的含义是 ：创建一个实现ActionListener接口的类的新对象 需要实现的方法actionPerformed定义在括号{}内
关于匿名内部类的具体代码格式如下：
superClass (construction parameters) {
inner class methods and data
}
这个superclass不仅仅可以是ActionListener这样的接口 于是内部类就要实现这个接口 也可以是一个类
于是内部类就要扩展它
匿名类没有类名 所以**匿名类不能有构造器 取而代之的是 将构造器参数传递给超类构造器** 尤其是在内部类实现接口的时候 不能有任何构造参数
不仅如此 还要提供下面这样提供一组括号
new InterfaceType(){
methods and data
}
这样匿名类的好处在哪呢 这样能够节省一些录入的时间 但是节省这单时间会让人陷入混乱的java代码中
双括号初始化
如果你想传递一个数组列表到一个方法 并且希望它作为一个匿名列表 那该如何添加元素呢
methodName (new ArrayList<String>(){{add("A") ; add("B")}}) 这就叫双括号初始化

```
public static void main(String[] args) {

Test test = new Test();
test.ceshi(new ArrayList<String>(){{add("a");add("b");}});
}


void  ceshi(ArrayList<String> arrayList){
System.out.println(arrayList.get(0));
System.out.println(arrayList.get(1));
}
```

警告：注意匿名类的equals方法需要当心
equals方法其中一种的推荐的比较方法就是如下
if(getClass() !=other.getClass())return falsel
但是对匿名子类做这个测试时会失败

提示： this在对静态方法中使用的时候需要注意 静态方法没有this 当我们想getclass这个本身这个类是时候就很麻烦
可以这么做 new Object[]{}.getClass().getEnclosingClass() 在这里 new Object[]{}会建立Object的一个匿名子类的一个匿名对象
getEnclosingClass 则得到其外围类 也就是包含这个静态方法的类

静态内部类

有的时候 使用内部类只是为了把一个类隐藏在一个类的内部 并不需要内部类引用外围类对象 为此 可以将内部类声明为static 取消产生的引用
比如如下所示

```
ArrAlg.pair p =  ArrAlg.minmax(d);
System.out.println("min =" + p.getFirst());
System.out.println("max =" + p.getSecond());
```

只有内部类才能声明为static 静态内部类的对象除了没有对生成它的外围类对象的引用特权外 与其他所有内部类完全一样
在我们的示例中 必须使用静态类 这是因为 静态方法导致的
注释：在内部类不需要使用外部类的时候 应该要使用静态内部类 这接口中声明内部类的时候 是自动生成为static 和public类的

```
public class StaticInnerClassTest {

public static void main(String[] args) {
double[] d = new double[20];
for (int i = 0; i <d.length ; i++) d[i] = 100 * Math.random();
ArrAlg.pair p =  ArrAlg.minmax(d);
System.out.println("min =" + p.getFirst());
System.out.println("max =" + p.getSecond());
    }
}
class ArrAlg
{
```

```
public  static  class  pair{
private  double first;
private double second;

public pair(double first, double second) {
this.first = first;
this.second = second;
        }

public double getFirst() {
return first;
        }

public double getSecond() {
return second;
        }
    }
public  static  pair minmax(double[] values)
    {
double min=Double.MAX_VALUE;
double max = Double.MIN_VALUE;
for (double v : values) {

if(min >v)min=v;
if(max <v)max=v;
        }
return new pair(min, max);
    }

}
```

代理

利用代理可以在运行时创建一个实现了一组给定接口的类　这种功能只有在编译时无法确定需要实现哪个接口才有必要使用

假设有一个表示接口的Class对象（有可能只包含一个接口）它的确切类型在编译时无法知道 这确实有些难度 要想构造一个实现这些接口的类 就需要使用newInstance方法或反射找出这个类的构造器 但是 不能实例化一个接口 需要在程序处于运行状态时定义一个新类

代理机制是一种很好的解决方案 代理类可以在运行时创建全新的类 这样的代理类能够实现指定的接口

它具有下列方法:

1指定接口所需要所需要的全部方法

2 Object类中的全部方法 列如 toString ，equals等

不能在运行时定义这些方法的新代码 而是要提供一个**调用处理器** 调用处理器是实现了InvocationHandler接口的类对象　而这个接口中只有一个方法

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
```

无论何时调用代理对象的方法 调用处理器的invoke方法都会被调用 并向其传递Method对象和原始的调用参数 调用处理器必须给出处理调用的方式

关于使用Proxy类的newProxyInstance方法 这个方法有三个参数

1 一个类加载器

2 一个Class对象数组 每个元素都是要实现的接口

3 一个调用处理器

```
import java.lang.Comparable;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.Arrays;
import java.util.Random;

/**
```

```
 * Created by han on 2016/8/15.
 */
public class ProxyTest {

public static void main(String[] args) {
Object[] elements = new Object[1000];

for (int i = 0; i <elements.length ; i++) {
Integer value=i+1;
InvocationHandler handler = new TraceHandler(value);
Object proxy = Proxy.newProxyInstance(null, new Class[]{Comparable.class}, handler);
elements[i] = proxy;
        }

Integer key = new Random().nextInt(elements.length)+1;

int result = Arrays.binarySearch(elements, key);
// 到这里运行binarySearch方法JDK动态代理才会动态的去创建代理类
//  因为binarySearch是按照这样子调用的  if（element[i].compareTo(key)）这个compareTo方法调用了invoke方法

if(result >=0) System.out.println("=========="+elements[result]);
    }
}
class  TraceHandler implements  InvocationHandler
{

private  Object target;

public TraceHandler(Object t) {
this.target = t;
    }

@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
System.out.print(target);
System.out.print("." + method.getName() + "(");

if (args != null) {

for (int i = 0; i < args.length; i++) {
System.out.print(args[i]);
if(i<args.length -1) System.out.print(", ");
            }
        }
System.out.println(")");

return method.invoke(target, args);
    }
}
```

注意的是 toString不是属于这个接口的对象 也是被代理的

代理类的特性

代理类是在程序运行过程中创建的 然而 一旦被创建 就变成了常规类 与虚拟机中如何其他类没有什么区别
所有的代理类都扩展于Proxy类 **一个代理类只有一个实例域 ---调用处理器** 它定义在Proxy的超类中
为了履行代理对象的职责 所需要的任何附加数据都必须存储在调用处理器中
所有的代理类都覆盖了Object类中的方法toString ，equals，hashCode 与所有的代理方法一样 这些方法仅仅只是调用了invoke
没有定义代理类的名字 Sun虚拟机中的Proxy类将生成一个以字符串$Proxy开头的类名
对于**特定的类加载器和预设的一组接口来说 只能有一个代理类** 也就是说 如果使用同一个类加载器和接口数组调用俩次
newProxyInstance方法的话
那么只能得到同一个类的俩个对象 也可以利用getProxyClass方法获得这个类：Class proxy =Proxy.getProxyClass(null,interfaces);
代理类一定是public和final **如果代理类实现的所有接口都是public 代理类就不属于某个特定的包 否则 所有非公有的接口都必须属于同一包 同时 代理类也属于这个包**
可以通过调用Proxy类中的isProxyClass方法检测一个特定的Class对象是否代表一个代理类