笔记5

第 5 章
继承

1 类，超类和子类

父类与子类的关系是 is-a关系 is-a关系是继承的一个明显特征
关键字extends表明正在构造的新类派生于一个已存在的类　而已存在的类被称为超类 ，基类，或父类，新类被称为子类 ，派生类或孩子类

子类比超类拥有的功能更加丰富

在通过扩展超类定义子类的适合 仅需要指出子类与超类的不同之处 因此在设计类的时候 应该将通用的方法放在超类中，而将具有特殊用途的方法放在子类中 这种
将通用功能放到超类的做法 在面向对象程序设计中十分普遍

super与this的概念并不一样 super不是一个对象的引用 不能将super赋予另一个对象变量 它只是一个指示编译器调用超类方法的特殊关键字
在子类中可以增加域 增加方法或覆盖超类的方法 然而绝对不能删除基础的任何域和方法

在使用子类的时候 必须对父类进行构造器初始化

```
public A(int a) {
super(a);
}
```

这里的super语句表示 调用父类中含有a参数的构造器

使用super调用构造器的语句必须是子类构造器的第一个语句

如果子类的构造器没有显示地调用超类的构造器，则将自动地调用超类默认（没有参数）的构造器 如果超类没有不带参数的构造器 并且在子类的构造器中没有显示的调用超类的其他构造器 则java编译器错误

关键字this有俩个用途 1是引用隐式参数 2是调用该类的其他的构造器 同样 super关键字也有俩个用途 一是调用超类的方法
二是调用超类的构造器　在调用构造器的适合 这两个关键字的使用很相似 必须都是在构造器第一条语句使用 this可以传递给本类的其他构造器 super可以传递给超类的构造器

一个对象变量（列如 变量e）可以指示多种实际类型的现象被称为多态 在运行时能够自动地选择调用哪个方法的现象被称为动态绑定

1.1 继承层次

由一个公共链派生出来的所有类都被称为继承层次
在继承层次中 从某个特定类型到其祖先类的路径的继承链

通常 一个祖先类可以拥有多个子孙继承链

1.2 多态

有一个用来应该设计为继承关系的简单规则 这就是is-a原则 它表明子类的每个对象也是超类的对象
is -a的另外一种表示是置换法则　它表明程序出现的超类对象的任何地方都有用子类对象置换

在java程序设计语言中 对象变量是多态的 一个父类对象可以引用一个父类对象　也可以引用一个父类的任何一个子类的对象

```
A a = new A(1);//子类
C c = new C(2);//父类
c=a;//  成功
a=c;//出错
```

但是倒过来就出错了 不能将一个父类的引用赋给子类对象

在java中 子类数组的引用可以转换成超类数组的引用，而不需要采用强制类型转换　但是这样就有可能出错 请看下面

```
A[] as = new A[10];//子类 假如这是经理
C[] cs  = new C[10];//父类  假如这是员工
```

```
cs=as;
cs[0] = new C(2);//这里就出错了  一个C不一定是A
```

所有数组都要牢记创建它们的元素类型 并负责监督仅将类型兼容的引用存储在数组中

5.1.3

动态绑定

1 编译器查看对象的声明类型和方法名 假如需要f（Int）方法 编译器就会列出所有c类的名为f的方法和其超类中访问属性为public且名为f的方法
至此 编译器就已经获得所有可能被调用的候选方法
2 接下来 编译器将查看调用方法时提供的参数类型 如果在所有名为f的方法中存在一个与提供的参数类型完全匹配 就选择这个方法 这个过程被称为**重载解析**
注意：返回类型不是签名的一部分 因此 在覆盖方法时 一定要保证返回类型的兼容性 **允许子类将覆盖方法的返回类型定义为原返回类型的子类型**

```
/*C是父类 */
C  Ceshi() {
return new C();
}
```

```
/*A是C的子类*/
@Override
A Ceshi() {

return new A(1);
}
```

3 如果是private方法，static方法，final方法（有关修饰符的含义将在下一节讲解）或者构造器 那么编译器可以准确的知道应该调用哪个方法 我们将这种调用方式称为静态绑定

4 当程序运行时 并且采用动态绑定调用方法时 虚拟机一定调用与x所引用对象的实际类型最合适的那个类的方法
每次调用方法都要进行搜索 时间开销大 因此 虚拟机预先为每个类创建了一个**方法表** 其中列出了所有方法的签名和实际调用的方法 这样一来，在真正调用方法的适合，虚拟机仅查找这个表就行了

警告：在覆盖一个方法的适合 子类方法不能低于超类方法的可见性 特别是 如果超类的方法是public 子类方法一定要声明为public 如果 在声明子类方法的时候 遗漏了public修饰符 此时 编译器将会把它们解释为试图提供更严格的访问权限

5.1.4 阻止继承 final类和方法

如果可能希望阻止入门利用某个类定义子类 不允许扩展的类称为final类 如果在定义类的时候使用final修饰符就表示这个类是final类

同样 类中的方法也可以声明为final 如果这样做 子类就不能覆盖这个方法 （final类的所有方法都是final方法）
域也可以被声明为final 对于final域来说 构造对象之后就不允许改变它们的值了 不过 如果一个类声明为final 只有其中的方法自动地成为final 而不包括域

将方法或类声明为final的主要目的 确保它们不会在子类中改变语义
在早些的java中 有些程序员为了**避免动态绑定带来的系统开销而使用final关键字**
如果一个方法没有被覆盖并且很短 编译器就能够对它进行优化处理 这个过程被称为内联
列如：内联调用e.getName()将被替换为访问e.name域 这是一项很有意义的改进 这是由于cpu在处理方法时 使用的分支转移会扰乱预取指令的策略
所以 这被视为不受欢迎的 然而 如果getName在另外一个类中被覆盖了 那么编译器就无法知道覆盖的代码将会做什么操作 因此也就不能对它们进行内联处理了

如果虚拟机加载了另一个子类 而在这个子类中包含了对内联的方法的覆盖 那么优化器将取消对覆盖方法的内联 这个过程很慢 但却很少发生

强制类型转换

Manager boos=(Manager)staff[0];
强制类型转换的格式如上

将一个值存入变量时 编译器将检查是否允许该操作
但 将一个超类的引用赋给一个子类变量 必须进行类型转换 这样才能够通过运行时检查

应该养成这样良好的程序设计习惯 在进行类型转换之前 先看一下是否能够成功地转换 这个过程可以用instanceof运算符

```
if(a instanceof  C){
System.out.println("成功");
}
```

一般只有在需要子类的特殊方法时 才会用强制类型转换 在一般情况下 尽量不要使用强制类型转换 和instanceof运算符
只能在继承层次使用强制转换
 在超类转换为子类前 应该使用instanceof来判断

抽象类

抽象类 我有自己的一些理解 person student employee 三个类 其中person毫无疑问是父类 拥有着普遍的都有的东西
这个person里面也同样有这描述自己是什么的一个方法 但是注意 person是无法描述自己是什么的 在这里无法进行这个方法的工作
在它的子类中到是可以做到这一点 于是 这便是在person这个祖先类中抽象的一个概念 这也是抽象类的由来 我们在这里定义了这么一个抽象方法 这个抽象方法只是定义了这个方法 但这个方法的具体实现是在子类中 这样就能够解决了这个问题 也符合面向对象的思想

抽象类的具体关键字是abstract
为了**提高程序的清晰度 包含一个或多个抽象类方法的本身也是必须要声明为抽象的**
除了抽象方法外 抽象类还可以包含具体数据和具体方法
扩展抽象类有俩种选择 一种是在抽象类中定义部分抽象类方法或不定义抽象方法
这样就必须将子类也标记为抽象类 另一种是在抽象类定义全部的抽象方法 这样子类就不是抽象的了

抽象类不能被实例化
但可以创建一个具体子类的对象 需要注意
可以定义一个抽象类的对象变量 但是它只能引用非抽象子类的对象

受保护方法

 最好将类中的域标记为private 而方法标记为public 任何声明为private的内容对其他类都是不可见的
人们希望某些方法允许被子类访问 或允许子类的方法访问超类的某个域 为此 需要将这些方法或域声明为protected
在实际应用中 需谨慎的使用protectd属性 假如 实现一个父类 底下有许多子类 一旦修改了父类的protectd方法
 也要将下面的子类方法全部修改 违背了oop提倡的数据封装原则
受保护的方法更具有实际意义 如果需要限制某个方法的使用 就可以将它**声明为protectd 这表明子类得到信任**
可以正确的使用这个方法 而其他类不行

java控制可见性的4个访问修饰符
1 仅仅对本类可见--private
2 对所有类可见---public
3 对本包和所有子类可见 ---protectd
4 对本包可见--默认 不需要修饰符

Object--所有类的基类

Object类型的变量只能作为各种值的通用持久者 要想对其中的内容进行具体的操作 还需要清除对象的原始类型 并进行相应的类型转换
在java中 只有基本类型不是对象 比如int Boolean。。但是数组类型不是基本类型 不管是对象数组 还是基本类型的数组都是扩展与Object

1 equals方法

Objec类中的equals方法用于检测一个对象是否等于另外一个对象
如果俩个对象具有相同的引用 它们一定是相等的
但是 对于大部分类来说 并不是这么来判断的 而是来判断类的状态是否相等 比如学生A 类与学生B类的年龄 姓名之类的是否一样
这里有个方法 叫getClass 这个方法是反射里的内容 这个方法返回一个对象的所属的类 在这个方法中 用这个方法来比较的话只有相同的类才可能相等
 其实在java内部提供了一个方法 叫Objects.equals方法 这个方法可以用来判断null值 防止null值的出现导致的出错

在子类中定义equals方法时 首先调用超类的equals 如果检测失败 对象就不可能相等 如果超类中的域都相等 就需要比较子类中的实例域

相等测试与继承

java语言规范要求equals具有下面的特性
1 自反性

2 对称性
3 传递性
4 一致性
5 对任何非空引用x ,x.equals(null)返回false
这些规则从而避免了类库实现者在数据结构中定义一个元素时还要考虑调用x.equals(y)，还是要调用y.equals(x)的问题
有俩种方法进行比较 一种是instanceof 一种是getClass
需要说明的是instanceof测试并不是完美无瑕的 类与类存在继承的关系 如果子类与父类进行比较 子类是父类的 但是子类自己的特殊信息怎么办？
但是 getClass方法进行比较的话 这样有可能就不符合置换原则
这本书在这里进行了一些经验的总结：
如果子类能够拥有自己的相等概念 则对称性需求将强制采用getClass进行检测
如果由父类决定相等的概念，那么就可以使用instanceof进行检测 这样可以在不同子类的对象之间进行相等的比较

hashcode方法

散列码是由对象导出的一个整形值 散列码是没有规律的
比如String类是如下计算散列码的
int hash=0；
for(int i=0;i<length();i++)
hash=31 *hash+charAt(i);

如果重新定义散列码 就必须重新定义equals方法
hascode方法应该返回一个整形数值 并合理的组合实例域的散列码
以便能够让不同的对象产生的散列码更加均匀
在JAVA7中还作了一些改进
首先 最好使用null安全的方法object.hashCode 如果其参数为null 这个方法会返回0 否则返回对参数调用hashcode的结果

还有更好的做法 需要多个散列值时 可以调用Objects.hash并提供多个参数 这个方法会对各个参数调用Objects.hasCode ,并组合这些值
一定要确定 Equals和HashCode的定义必须一致 如果x.equals(y)返回true 那么x.hashCode()就必须与y.hashCode()具有相同的值 例如
如果定义的Employee.equals比较雇员的ID 那么hashCode方法就需要散列ID 而不是雇员的姓名和存储地址
如果是数组类型：可以用静态的一个方法 Arrays.hashCode方法计算一个散列码 这个散列码由数组元素的散列码组成

在这本书这里 还涉及了一个关于编写一个完美的equals方法的建议 其实现在的IDE里都自动生成了模板 不用自己手动写了 到时候有兴趣的话就看书吧
提示 对于数组类型的域 可以使用静态的Arrays.equals方法检测相应的数组元素是否相等

toString方法

toString方法用于返回表示对象值的字符串
toString的格式 是类的名字然后是一对方括号括起来的数值
随处可见toString的主要原因是 至于对象与一个字符串通过操作符"+"连接起来 java编译器就会自动地调用toString方法 以便获得这个对象的字符串描述

提示 在调用x.toString 的地方可以用""+x代替 这里的x就是toString 与toString不同的是 如果x是基本类型 也能够正确执行

如果x是一个对象

调用
System.out.println(x)
println方法就可以直接调用x.toString 并输出得到的字符串
Object类定义了toString方法 用来打印输出对象所属的类名和散列码

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

这样就会直接输出
java.io.PrintStream@677327b6 这是因为 PrintSteam类没有覆盖toString
警告:在数组中 数组继承了Objec的toString方法 但是数组类型按照旧的格式打印 这个格式很奇怪 修正的方法是使用静态的Arrays.toString 才能打印出与其他类toString格式一样的代码
toString同时是一个很有用的调试工具
建议自定义类都加一个toString 这样不仅对自己好 对其他使用这个类的程序员也好

泛型数组列表

这里主要介绍的是Arraylist这个动态数组

ArrayList是一个采用类型参数的泛型类
在JAVA5之前是没有泛型类的 Arraylist保存的是Object的元素 它是自适应大小的集合 如果使用古老版本的JAVA的话 就需要将后面的<>去掉
在老版本中 程序员使用Vector类实现动态数组
这里有个方法 如果已经清楚或能够估计除数组可能存储的元素数量 就可以在填充数组之前调用ensureCapactiy方法

```
staff.ensureCapacity(100);
```

这个方法返回一个包含100个对象的内部数组 然后调用100次add 而不用重新分配空间

一旦确定数组列表的大小不再发生变化 **就可以调用trimToSize方法 这个方法将存储区域的大小调整为当前元素所需要的存储空间数目 垃圾回收器将回收多余的存储空间**
一旦 整理了数组列表的大小 **添加新元素就需要花时间再次移动存储块 所以应该在确认不会添加任何元素时 再调用trimToSize**
这是个优化的不错的方法

```
staff.trimToSize();
```

访问数组列表元素

ArrayList不是JAVA程序设计语言的一部分
介绍了一些方法 add set get size

类型化与原始数组列表的兼容性

可能更愿意使用类型参数来增加安全性
下面这是一个方法 我们可以将一个带泛型的arraylist传入其中 但是这样是没有检测的 这样的调用并不是很安全 因为添加到这个方法的元素不一定是泛型限制的那个类

```
void Test(ArrayList list) {
```

```
ArrayList<Employee> arrayList = new ArrayList<>();
list = arrayList;//警告
```

比如上面这样的情况 就会出现警告 被告之转换有误 使用类型转换也还会出现这个警告
这就是**java不尽人意的参数化类型的限制所带来的结果 编译器在对类型转换进行检查之后 如果没有发现违反规则的现象 就将所有的类型化数组列表转换为原始Arraylist对象**
在程序运行时 所有的数组列表都是一样的 虚拟机中的类没有类型参数

可以用一个注解来标注这个变量能够接受类型转换 如下：

```
@SuppressWarnings("unchecked")
void Test(ArrayList list) {
ArrayList<Employee> arrayList = new ArrayList<>();
list = arrayList;
}
```

对象包装器与自动装箱

int Float Long Double ... 每个类都有包装器 比如int有Integer 这个包装器类 对象包装器是不变的 即一旦构造了包装器 就不允许更改包装器中的值 同时
对象包装器类还是final 因此 不能定义它们的子类

假如定义写个整形数组列表 而尖括号中的类型参数不允许为基本类型 比如

```
ArrayList<int> list1 = new ArrayList<>();//error
```

这样就是不行的 必须要用包装器

```
ArrayList<Integer> list1 = new ArrayList<>();
```

由于每个值分别包装在对象中，所以Arraylist<Integer>的效率远远低于int[]数组 因此 应该用它构造小型集合 其原因是此时程序员操作的方便性要比执行效率更加重要

JAVA se5 的另一个改进之处是更加**便于**添加或获得数组元素
比如 list.add(3)
自动转换为
list.add(Integer.valueof(3));
这叫**自动装箱**

当将一个Integer 对象赋给int值时 将会自动地拆箱
甚至在算术表达式也能自动的进行拆箱装箱

Interger n=3；
n++；
编译器将自动地插入一条对象拆箱的指令，然后进行自增计算 最后再将结果装箱
在很多情况下 **容易误以为 基本类型与它们的对象包装器是一样的**

它们的相等性不同 ==运算符可以应用于对象包装器对象 只不过检测的是对象是否指向同一个存储区域 一般不会相等成立 java实现却又可能让它成立 如果将经常出现的值包装到同一个对象中 这种比较有可能成立 这种不确定的结果并不是我们所希望的 解决这个问题的办法是在**俩个包装器对象比较时调用equals方法**

自动装箱规范要求boolean byte char<=127 介于128-127之间的short和int被包装到固定的对象中

装箱和拆箱是**编译器**认可的 而不是虚拟机 编译器在生成类的字节码的适合 插入必要的方法调用 而虚拟机只是执行这些字节码

使用数值对象包装器还有另一个好处 java设计者发现 可以将某些基本方法放置在包装器中
比如经常使用的 将字符串转换为整形的方法
Interger.parseInt(s);
Java方法都是**值传递** 是不可能编写一个下面的这样的能够增加整形参数值的java方法

```
void ceshi2(int x){
x=3*x;
}
```

换成Interger会如何

问题在于Integer对象是**不可变** 包含在包装器中的内容不会改变 不能使用这些包装器类创建修改数值参数的方法
如果想编写一个修改数值的方法 就需要使用在org.omg.CORBA包中定义的持有者类型 **包装IntHolder，BooleanHolder等，每个持有者类型都包含一个公有域值**
通过它可以访问存储在其中的值 在这里将进行传入的参数改变的目的

```
void triple(IntHolder x) {
x.value = 3 * x.value;
}
```

参数数量可变的方法

以下面的这个输出方法为例

```
public PrintStream printf(String format, Object ... args) {
return format(format, args);
}
```

这里的Object的参数的后面的...代表的就是可变 可以接受任意数量的object对象
这里的对printf的实现者来说object...参数类型和object[]是完全一样的 因为可以自动进行装箱拆箱 转换
编译器对printf的每次调用进行转换 以便将参数绑定到数组上 并在必要的适合进行自动装箱
注意： **允许将一个数组传递给可变参数方法的最后一个参数** 列如

```
System.out.printf("%d %s", new Object[]{new Integer(1),"widgets"});
```

所以可以将已经存在最后一个参数是数组的方法重新定义为可变参数的方法 而不会破坏内部任何已经存在的代码

```
public static void main(String... args) {
```

比如这样

枚举类

下面的 就是枚举类的一个列子

```
public enum Size {

SMALL,MEDIUM,LARGE,EXTRA_LARGE;

}
```

所有的枚举类型都是enum的子类 其中最有用的一个是toString方法 而且在比较俩个枚举类型的值时 直接 ==就可以了 不用equals

如果需要的话 **也是可以在枚举中添加一些构造器 方法 和域的 当然 构造器 只是在构造枚举常量的适合被调用**
每个枚举类型都有一个**静态的values方法** 它返回一个包含全部枚举值的数组
ordinal方法返回enum声明中的枚举常量的位置 位置从0开始计数 列如

```
Size.MEDIUM.ordinal();
```

反射

反射库提供了一个非常丰富且精心设计的工具集 以便写出能动态操作java代码的程序 这项功能大量的被应用于JavaBean 它是Java组件的体系结构
特别是在设计或允许中添加新类时，能够快速地应用开发工具动态的查询新添加类的能力
能够分析类能力的程序称为反射
反射可用用来
1 在运行时分析类的能力
2 在运行时查看对象
3 实现通用的数组操作代码
4 利用Method对象

Class类

在程序运行期间 Java运行时系统始终为所有的对象维护一个被称为运行时的类型标记 这个信息跟踪每个对象的所属的类 虚拟机利用运行时类型信息选择相应的方法执行
如同用一个Emplyee对象表示一个特定的雇员属性一样， 一个Class对象将表示一个特定类的属性 最常用的Class方法是getName 这个方法返回类的名字
如果类在包里 那么包名也会作为类名的一部分

```
C c = new C();
Class c2 = c.getClass();
```

如果类名保存在字符串 并可在运行时改变就可以使用这个方法 这个方法只有在这个字符串是类名或接口才能够执行 否则 就抛出一个异常

```
Class.forName("com.inherit.C");
```

无论何时使用这个方法 都应该添加一个异常处理器

```
try {
Class.forName("com.inherit.C");
} catch (ClassNotFoundException e) {
e.printStackTrace();
}
```

有个技巧 在启动时 包含main方法的类被加载 它会加载所有需要的类 这些被加载的类又要加载它们需要的类 以此类推 对一个大型的应用程序来说 这个启动时十分缓慢的 可以使用下面一个方法 使得有一种启动比较快的错觉 首先确保包含main方法的类没有显示的引用其他的类 然后 显示一个启动画面 然后 通过调用 Class.forname手工的加载其他的类

获得Class类对象的第三种办法

```
Class<C> c4 = C.class;
```

注意 一个Class对象实际上表示的是一个类型 而这个类型未必是一种类 比如 int 不是类 但 int.Class 就是Class对象了

警告：由于历史原因 数组的getName方法会返回一个很奇怪的名字

虚拟机为每个类型管理一个Class对象 因此 可以利用 ==运算符实现俩个类对象的比较的操作
比如

```
if(C.class==  Class.forName("com.inherit.C")) System.out.println("yes");
```

还有一个很有用的方法 newInstance()用来快速地创建一个类的实例
如下：

```
Class<C> c4 = C.class;
c4.newInstance();
```

创建了一个与c4具有相同类型的实例 newInstance方法调用默认的构造器（没有参数的构造器）初始化新创建的对象 如果这个类没有默认的构造器 就会抛出一个异常
将forname与newInstance配合起来使用 可以根据存储在字符串中的类名创建一个对象

```
Object sClass= Class.forName("com.inherit.C").newInstance();
```

如果需要以这种方式按名称创建的类的构造器提供参数 就不要使用上面的那条语句 **而必须使用Constructor类中newInstance方法**

捕捉异常

异常是很重要的
这里没什么好讲的 在第11章有详细的讲解

主要是分为已检查的异常 和未检测的异常

利用反射分析类的能力

反射最重要的内容 检测类的结构
在java.lang.reflect包中有三个类 分别是Field Method Constructor分别用俩描述类的域 方法 和构造器
这三 个类都有一个getname的方法 返回项目的名称
field类有一个getType方法 用来描述所属类型的Class对象 Method 和Consreuctor类有能够报告参数类型的方法 ,Method类还有一个报告返回类型的方法
这三个类还有一个getModifiers的方法 它将返回一个整形数值 用不用的位开关来描述public static这样的修饰符使用情况 另外 是可以用reflec包中的Modifier类的静态方法分析getModifiers返回的整型数值
同时 我们还基于利用Modifiers.toString 方法将修饰符打印出来

Class类中的getFields getMothod getConstructor方法将分别返回类的public域 方法 和构造器数组 其中是包含父类公有成员
Class的getDeclareFields，
getDeclareMethods和getDeclaredConstructors方法将分别返回类中声明的全部域 方法和构造器 其中包含私有和受保护成员 但是不包含超类的成员

```
System.out.println("反射的内容");
System.out.println("=======================================================");
C c=new C();
Class c1=C.class;

Class c3 = c.getClass();
try {
if(C.class==  Class.forName("com.inherit.C")) System.out.println("yes");
    } catch (ClassNotFoundException e) {
e.printStackTrace();
    }
try {
String s = "com.inherit.C";

Class sClass = Class.forName("com.inherit.C");
Object SnewInstanceClass= sClass.newInstance();
Method[] method = null;

System.out.println(sClass.getSuperclass().getName());
int x = 0;
System.out.println("======================");
for (Method m : sClass.getDeclaredMethods()) {
m.invoke(SnewInstanceClass);
System.out.println(Modifier.isPublic(m.getModifiers()));
System.out.println(Modifier.toString(m.getModifiers()));
System.out.println(m.getName());
Class returnType = m.getReturnType();
System.out.println(returnType.getName()+" "+m.getName()+" ");
System.out.println(1);


/*类型参数*/
Parameter[] parameters = m.getParameters();
for (Parameter p : parameters) {
System.out.println(p.getName());
           }

        }
    }catch(ClassNotFoundException e){
e.printStackTrace();
    } catch (InvocationTargetException e) {
e.printStackTrace();
    } catch (IllegalAccessException e) {
e.printStackTrace();
    } catch (InstantiationException e) {
e.printStackTrace();
    }
```

```
}
```

在运行时使用反射分析对象

利用反射机制可以在编译时还不清楚的对象域

查看对象域的关键方法是Field类的get方法 如果f是一个field类型的对象 obj是包含f域的类对象 f.get(obj)将返回一个对象 其值为obj域的当前值
但是 这是存在问题的 如果我们要返回的obj是一个私有域的话 get的就会抛出一个IllegalAccessException异常 只有利用get方法才能得到可访问域的值 除非拥有访问权限 反射机制的默认行为受到java访问控制 然而 如果一个java程序没有受到安全管理器的控制 就可以覆盖setAccessible方法
setAccessible是AccessObject类的方法 这个类是field和Method和Constructor的共同的超类 这个特性是为了调试 持久存储 和相似机制提供的

```
Person[] persons = new Person[2];
persons[0] = new Employee("h",50000,1989,10,1);
persons[1] = new Student("M", "Computer ");
```

```
Class hraay=persons[0].getClass();
Field[] f=hraay.getDeclaredFields();
//AccessibleObject.setAccessible(f,true);//设置关闭安全管理器 私有的也可以被查询到
AccessibleObject.setAccessible(f, true);
for (Field field:f) {
field.setAccessible(true);
System.out.println(field.getName());

Object v = field.get(persons[0]);
System.out.println(v);
 }
```

注意 这里的域的类型需要注意 假如get是返回一个string类型的变量 而我们用的是getDouble 这就会出错 所以 尽量使用get方法 这个方法可以自动进行转换 并且
反射机制会自动的将这个域打包到相应的对象包装器中

另外还有一个set方法 这个是设置值的

另外可以利用这个写出一个通用的toString方法

使用反射编写泛型数组代码

java.lang.reflect包中的Array类允许动态的创建数组 比如 Array类的copyOf方法

```
public static Object[] badCopyOf(Object[] a, int newLength) // not useful
{
Object[] newArray = new Object[newLength];
System.arraycopy(a, 0, newArray, 0, Math.min(a.length, newLength));
return newArray;
}
```

但是 在实际使用中就会出现 一个问题 这段代码返回的数组类型是对象数组类型 这样就会出错
一个对象数组不能转换为其子类的数组 这样做会抛出异常的 一个子类数组临时转换为对象数组 然后再转换回来是可以的 但是一开始就是对象数组的是不能进行转换的
我们就要用到**Array的一个静态方法 这就是newInstance** 这个方法有俩个参数 一个是数组的元素类型 一个是数组的长度
为了能够实际运行 需要获得新数组的长度和元素类型
可以通过调用Array.getLength（）获得数组的长度 也可以通过Array类的静态方法getLength方法的返回值得到任意数组的长度

而得到新数组的类型 需要如下工作

1 首先获得a数组的类对象
2 确认它是一个数组
3 使用Class类的getComponenType方法确认数组对象的类型

就是获得原来的对象 然后进行填装 在这个步骤中 需要确认 需要获取类型 需要用Array的newInstance方法构建新类型

```
public static Object goodCopyOf(Object a, int newLength) {

Class cl = a.getClass();
if(!cl.isArray()) return null;
Class componenType = cl.getComponentType();
int length = Array.getLength(a);
Object newArray=Array.newInstance(componenType,newLength);
System.arraycopy(a, 0, newArray, 0,Math.min(length, newLength));
return newArray;

}
```

注意 应该将goodCopyOf方法的参数声明为Object类型 而不要声明为Object[] 整形数组类型int[]可以转换为Objec 但是不能转换为Object[]

调用任意方法

这里就是大概提到了一个方法 这个方法是method的invoke方法 这个与Field的get方法查看对象域的过程类似
invoke方法的签名是
Object invoke(Object,Object...args)
这个方法很重要 有俩个参数 第一个参数是隐式参数 就是当前要调用的方法类的对象实例
如果是静态方法 第一个参数是被忽略的 就是可以被设置为null

如果返回类型是基本类型 invoke方法会返回其包装器类型

如果调用方法提供了一个错误的参数 invoke方法就会报出异常
另外invoke方法的参数和返回值必须是Object类型的 这就意味必须进行多次的类型转换 这样做将会使编译器错过检测代码的机会 因此
等到测试阶段才会发现这些错误 找到并改正将会更加困难 不仅如此 使用反射获得方法指针的代码要比仅仅直接调用方法明显要慢一点
建议 在必要的适合才使用Method对象 而最好使用接口和内部类 建议java开发者不要使用Method对象的回调功能 使用接口进行回调会使得代码执行速度更快 更易于维护

1 将公共操作和域放在超类

2 不要使用受保护的域
也就是proteced这个proteced域有主要俩个缺点
2.1 子类集合是无限的 每一个子类都可以直接访问proteced的实例域 从而破坏封装性
2.2 在java设计语言中 同一个包中的所有类都可以访问proteced域 而不管它是否为这个类的子类
不过 protected方法对于指示那些不提供一般用途而应在子类中重新定义的方法很有用
3 使用继承实现 is-a关系

4 除非所有继承的方法都有意义 否则不要使用继承

5 在覆盖方法时 不要改变预期的行为
置换原则不仅应用于语法 而且可以应用于行为
在覆盖子类中的方法时 不要偏离最初的设计想法

6 使用多态 而非类型信息

使用多态方法或接口编写的代码比使用对多种类型进行检测的代码更加易于扩展和维护

7 不要过多的使用反射

反射是很脆弱的 即编译器很难帮助人们发现程序中的错误 因此只有在运行时才发现错误并导致异常