# 集合(Collection)

**集合都在java.util包中**。集合中存放的是Object，即所有对象都能放入。**Map和Collection都是接口**，但是它们之间没关系，它们是同级的。**派生就是继承的意思**。

**广义上讲集合**：java中的集合是指java.util包下面的容器类，包括和Collection及Map相关的所有类。

**中义上讲**：一般说集合特指java集合中的Collection相关的类，不包含Map相关的类。

## 一、List中的ArrayList

### 1.ArrayList

**List是一种有序列表的集合。List集合中都有下标索引**。

ArrayList的**默认初始容量是10**，即通过new ArrayList()创建时的容量：**private** **static** **final** **int** ***DEFAULT\_CAPACITY*** = 10。底层实际上是**一个Object类型的数组**：**transient** Object[] elementData。其**查找元素相对较快**。其**元素可通过下标索引来访问**。**可存放重复元素，可存放null**。该集合可以任意添加元素。定义集合时，尽量指定集合的长度，可以减少扩容的次数，提高效率。**ArrayList是线程不安全的**。**ArrayList的扩容为1.5倍，即增长原来的0.5倍**。

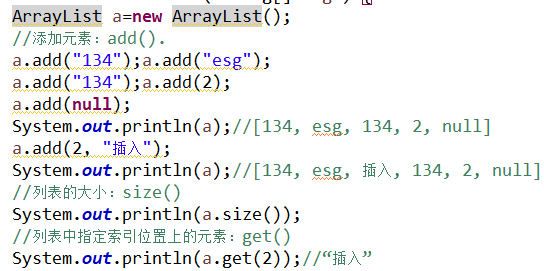
a**.add(Objetct o)**：ArrayList的方法，返回值为boolean类型，若将括号内的元素添加进列表中了，则返回值为true。a**.add(index，Object element)**：ArrayList的方法，无返回值，在指定索引位置添加元素。

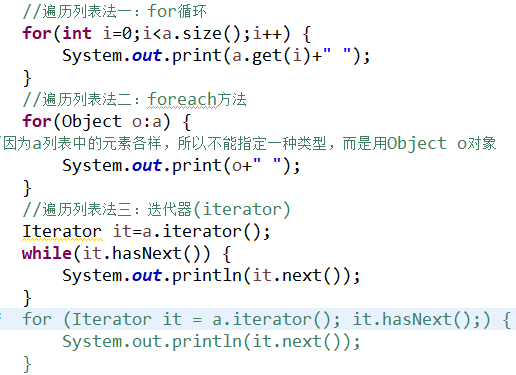
a**.size()**：返回值为int类型，返回列表的元素的个数，括号内没参数，ArrayList的方法。**注意列表中判断列表的大小是用size()方法，而不是length或length()**。

a**.get(int index)**：返回值为指定索引位置上的元素。ArrayList的方法。

a**.isEmpty()**：括号内没有参数，返回值为boolean类型，当列表为空的时候，返回的值为true。

**在集合中推荐用iterator迭代器遍历集合**。

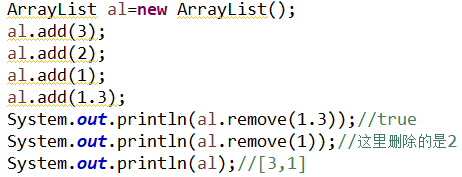


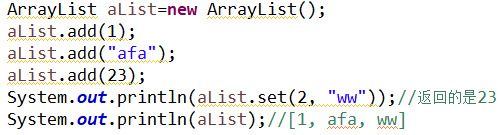


ArrayList**.remove(int index)**：删除列表中指定索引位置上的元素，返回值为这个被删除的元素。

ArrayList**.remove(Object o)**：删除括号中的参数对象，返回值为boolean类型。若删除成功则返回true，否则返回false。要特别注意这两个的区别。注意remove方法**一般是先按索引删除的**，即当括号内的参数既可表示索引位置，也可表示列表中的元素时，删除的是索引。

ArrayList**.set(int index，Object o)**：在指定索引位置上替换元素，参数2为替换的元素。该方法有一个返回值，返回的是被替换的元素。





**System.currentTimeMillis()**：括号内没参数，返回值的类型为long型，返回的是当前的毫秒数。针对底层的操作系统(underlying operating system)结果不同。

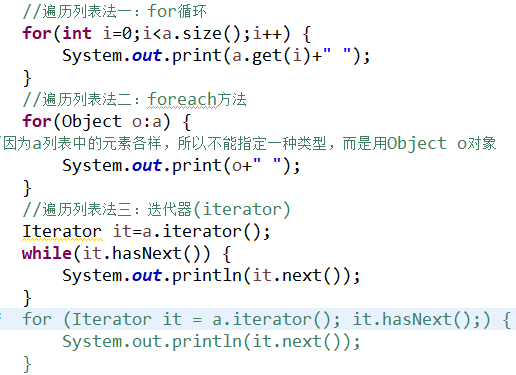
注意ArrayList的容量和长度不一样。ArrayList中的长度用size()表示，得到集合中的元素用get()方法。ArrayList的最大长度为**(231-1)-8：即整型的最大长度-8**。源码如下：

**private** **static** **final** **int** ***MAX\_ARRAY\_SIZE*** = Integer.***MAX\_VALUE*** – 8。

### 2.迭代器

[十四、迭代器](#_十四、迭代器)

ArrayList**.iterator()**：返回值为一个iterator迭代器，该方法是实现了Iterator接口后拥有的，直接点出来，而不是用new创建对象出来的。括号内没参数。ArrayList可以这个方法。一般模式可表示为： **Iterator it=a.iterator()**。it**.hasNext()**是判断指针下面是否有元素，i**t.next()**是返回当前迭代的元素指针并向下移动。注意hasNext()中N是大写，而next()中n是小写。



### 3.判断两个集合中的元素是否相同及获取不同的元素

#### ⑴两个集合中存放的是对象

当两个List集合中存放的内容是对象，如何去除集合中相同的对象。比如oldSoItems集合和newSoItems集合中都存放了SoItem对象，若SoItem的产品编号productCode相同，则表示这两个SoItem对象是同一个对像。

**法一**：**再添加两个集合用来存放相同的元素sameSoItems和sameSoItems2，不重写hashCode()和equals()方法**。遍历原来的两个集合，若两个集合中SoItem对象的productCode相同，则将这个两个SoItem分别放入准备好的集合中。分别遍历存放相同的元素的集合，将原来的集合调用remove()方法去除相同的元素。

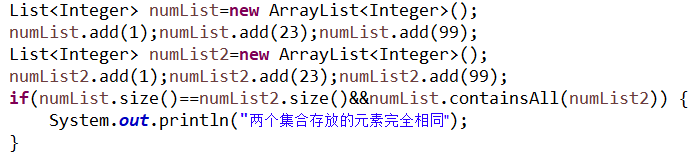
注意，当List集合中存放的是对象时，在没有重写对象的hashCode()和equals()方法时，不能只用一个集合去存放productCode相同的SoItem对象。比如下面将sameSoItems2去掉，相同的元素都存放在sameSoItems中，当遍历该集合去除newSoItems中的SoItem对象时，去除不了，因为sameSoItems集合中存放的对象一个都没存放在newSoItems集合中，newSoItems中存放的部分SoItem只是和sameSoItems中存放的SoItem对象的productCode相同而已，但两者实际上并不是相同的对象，所以newSoItems中并不能去除sameSoItems中的SoItem对象。

**法二**：**将两个集合newSoItems和oldSoItems中存放的SoItem对象重写equals()和hashCode()方法。然后用一个新集合存放相同的元素**，然后newSoItems和oldSoItems这两个集合去遍历这个存放相同元素的集合，调用remove()方法即可。



#### ⑵两个集合中存放的是基本数据类型

List有一个containsAll()方法，当一个集合中包含指定集合中的所有元素时，返回true。**boolean** containsAll(Collection<?> c);



## 二、List中的LinkedList

LinkedList的底层实际上是**双向链表**。该链表的特点是：**增加或删除操作的效率高，但是查找的效率低**。LinkedList比ArrayList多了头尾的操作。数组中元素存放的地址是连续的，而LinkedList是把不连续的元素连接起来，其底层是双向链表。

**ArrayList的查找效率高，若要不断查找数据，则用ArrayList，若要不断增加或删除数据，则用LinkedList**。

**ArrayList底层是Object类型的数组，在数组中存放的元素的地址是连续的，在访问元素时访问的是元素的地址，因为地址是连续的，所以查找效率高。每次对ArrayList进行插入、删除元素操作，都需要将数组中的元素进行向前或向后移动，同时若是插入操作，还需要判断是否需要进行扩容，因此插入和删除数据的效率低**。

**LinkedList的底层是双向链表，插入删除元素时只需要操作节点的引用即可，而不用移动元素，因此插入删除元素的效率高。LinkedList中存放的元素的地址不是连续的，当查找一个元素时，往往需要知道另一个元素的地址，因此需要向前或向后遍历，因此查询效率低**。

LinkedList中常用的方法和ArrayList中常用的方法基本一样。

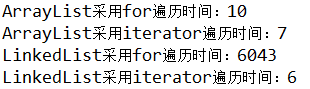
LinkedList、ArrayList中的方法到时候再补充一下，它们的底层原理再深入了解一下。

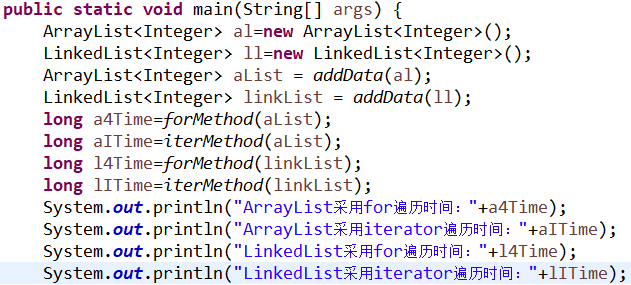
集合中各个的源码看一下

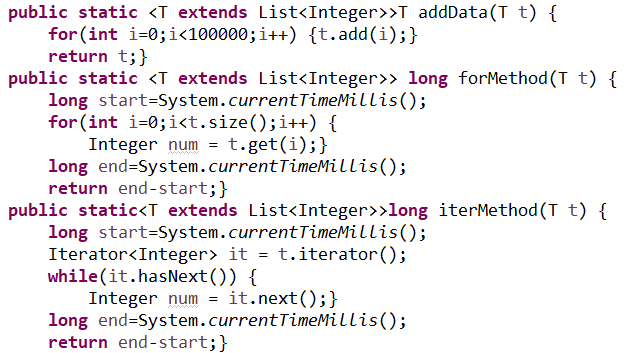
ArrayList实现了**RandomAccess**这个标记接口。这个接口的作用是判断集合是否能快速访问，也就是通过索引下标能否快速的移动到对应的元素上。目前已经知道了三个标记接口：RandomAccess、Serializable、Cloneable。

LinkedList建议采用 Iterator遍历，因为比采用for循环遍历要快很多。

**例子：**

(下面代码的输出结果，这个结果是不确定的，但是linkedList采用iterator遍历始终比for遍历要快)。





## 三、List中的Vector

Vector和Array List基本一样，唯一的区别就是**ArrayList是异步的，是线程不安全的；而Vector则是同步的，是线程安全的**。

Vector中常用的方法和ArrayList是基本一样的。**Vector默认扩容是2倍，即增长原来的1倍**。

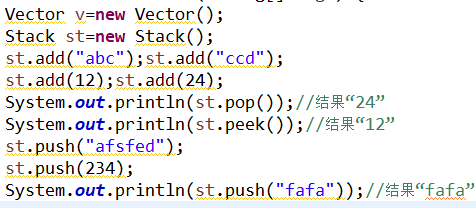
## 四、List中的Stack

**Stack是继承Vector的**。Stack中其他常用的方法同ArrayList差不多，下面是Stack中独有的方法：

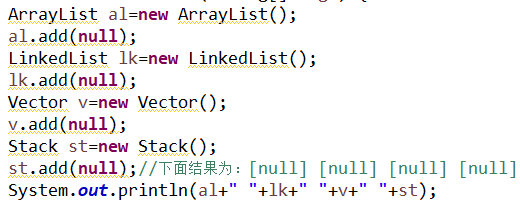
st**.peek()**：查看栈顶元素。无参数，返回值为栈顶的元素。Stack的方法。栈是先进后出的，其中的栈顶元素就是最后进栈的元素。而栈底元素则为最先进栈的元素。

st**.pop()**：无参数，删除栈顶元素，返回值被删除的元素，即返回并删除栈顶元素。Stack中的方法。

st**.push(Object item)**：向栈顶添加元素。括号内的参数为向栈顶中添加的元素，该方法的返回值为item的参数，即括号中添加的元素本身。该方法在Stack中使用同add()方法一样，不过add()方法的返回值为boolean类型。



三种方法都可以遍历ArrayList、LinkedList、Vector、Stack。同时List中都可以存放null。

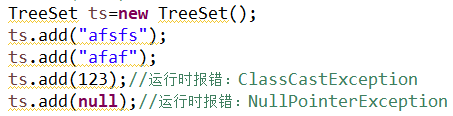


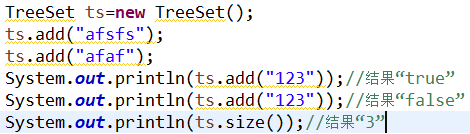
## 五、set中的TreeSet

### 1.TreeSet

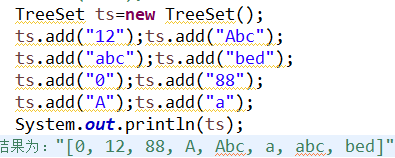
**set是一种没有重复元素的集合**。**set集合中都没有索引**。

**TreeSet的底层是由TreeMap实现的**。**TreeSet中存放的元素类型必须兼容，否则运行时会报错**。同时**TreeSet中不能存放null，否则运行时也会报错**。这两者在编译时都不会报错，只有运行时才会报错。**TreeSet也是线程不安全的**。





**注意TreeSet是有序的**，TreeSet中的元素有一套自己的排序方式。**数字是按照从小到大自然排序，字母是根据ASCII码值对应排序**，“A”为65，“a”为97。**String类型的数字和字母一起出现时，先排数字，再排字母**。



**TreeSet遍历的方式用：foreach()方法、iterator方法**。

当TreeSet中存放的是对象时，则不能直接打印输出了，否则出现ClassCastException异常，这是因为对象若没有自己默认的排序方式，则TreeSet中不好排序。若要使TreeSet中的元素为对象时进行排序可以使用比较器。

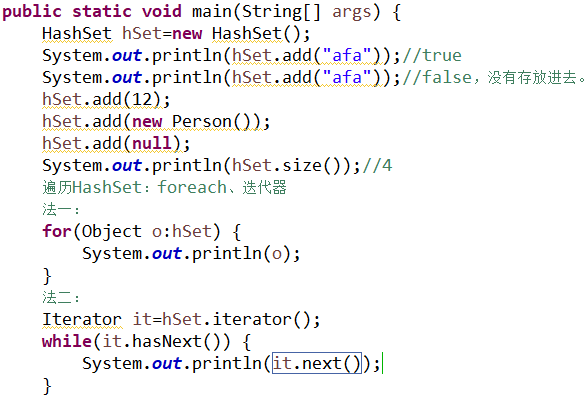
## 六、set中的HashSet

### 1.HashSet

HashSet底层实际上是**HashMap，HashSet中的各种构造方法都是调用HashMap中对应的构造方法**。**HashSet中不能存放重复的元素，当存放相同的元素时，会返回false，可以存放null**。HashSet中**可以存放数据类型不同的数据**。

HashSet中存放元素的顺序是按照HashSet集合自己的排列方式存储的。**HashSet是非线程安全的。HashSet中允许存放一个null值，这是因为HashMap中允许一个Key值为null。HashSet没有索引，所以没有List中的get()方法**。

**注意TreeSet是有序的，而HashSet是无序的，因为HashMap中的key是无序的**。



### **2.HashSet保证添加元素不重复的原理**

**⑴HashSet有两个属性**：①HashMap类型的map。②new Object()对象然后赋值给Object类型的PRESENT。源码如下：

**private** **transient** HashMap<E,Object> map

**private** **static** **final** Object ***PRESENT*** = **new** Object()

**⑵HashSet的一个构造器**：

**public** HashSet() {map = **new** HashMap<>();}

**⑶HashSet的add()方法**：

**public** **boolean** add(E e) {**return** map.put(e, ***PRESENT***)==**null**;}

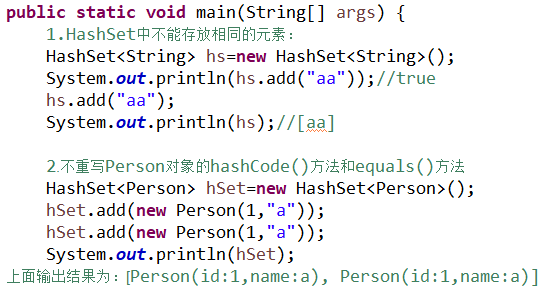
**原理分析**：**HashSet的底层是HashMap，当调用HashSet的无参构造器创建一个HashMap对象时，该构造器的代码创建的是一个HashMap对象。当调用HashSet的add()方法向集合中添加元素时，该add()方法的代码中调用了HashMap的put()方法，将要添加进集合中的元素作为put()方法中key的值，将创建一个新的Object对象作为put()方法的value值。因为HashMap中添加元素时，key值不能重复，所以HashSet中add()方法不能将重复的元素添加进HashSet集合中**。

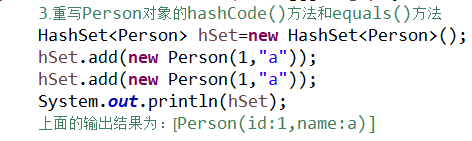
**Map中key不能重复的原因**：**先比较hash码值，再用equals()方法比较**

**①先判断hash码值是否相等，若不相等则说明是一个新元素，直接存入集合中。**

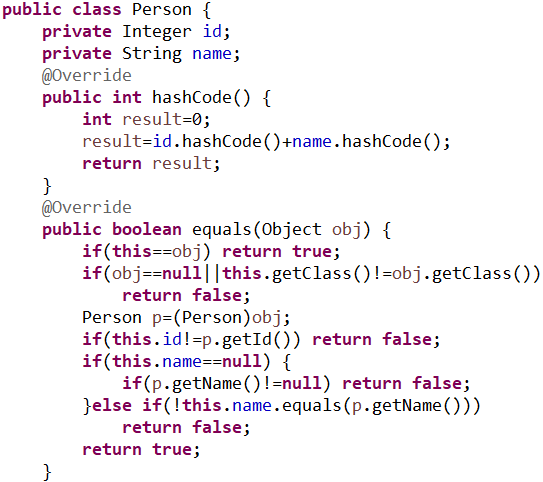
**②若hash码值相等，再用equals()方法判断是否相等，若相等，则说明元素已经存在，不存入集合中。若不相等，则说明集合中不存在该元素，直接将该元素存入集合中。**

**例子(非常重要)：**





**下面重写Person类的hashCode()和equals()方法很重要，要好好掌握，好好理解。**



**注意基本数据类型是没有hashCode()方法的，基本数据类型的包装类才有hashCode()方法，String类型的数据也有对应的hashCode()方法**。

### 3.简介hash码值和hash算法

(这个有待进一步学习)

在java中存在一种hash表结构，它通过一个算法，计算出的结果就是hash码值，这个算法叫hash算法。

## 七、比较器

**概念**：确定两个对象之间的大小关系及排列顺序称为**比较**，能实现这个比较功能的类或方法称之为**比较器**，在java中只有两种比较器。

**分类：内部比较器(Comparable接口)和外部比较器(Comparator接口)**。

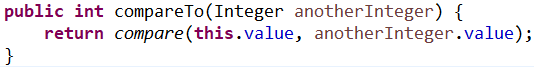
### 1.内部比较器(Comparable)

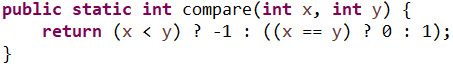
就是**把比较器写在类的内部。将要比较的类实现Comparable接口，重写compareTo()方法，该类就有了内部比较器**。

**public** **int** compareTo(Object o)：**该方法的返回值为0、负整数、正整数。若为0则表示此对象与目标对象相等。若为负数，则表示此对象小于目标对象。若为整数则此对象大于目标对象**。

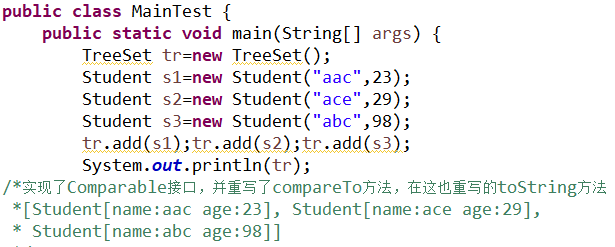
**重写compareTo方法时，有如下规则：若本类的属性-参数的属性(或者本类的属性在前，参数属性在后)，则是从小到大排序；若参数的属性-本类的属性(或者参数属性在前，本类的属性在后)则是从大到小排序。**

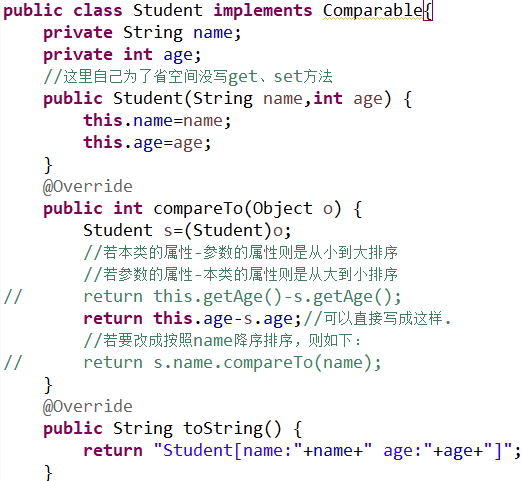
对于基本类型的封装类和String类型，它们都实现了内部比较器，即都实现了Comparable接口。所以它们都可以直接进行排序。如Integer是按从小到大的顺序进行排序的，其对应的源码如下。如果要将Integer按绝对值大小进行排序，则要自定义一个外部比较器了。





**实现Comparable接口的例子：**





### 2.外部比较器(Comparator)

就是**在类的外面重新定义一个比较器类，然后该比较器类实现Comparator接口并重写compare()方法。或者创建Comparator的匿名内部类**。

**int** compare(Object o1, Object o2)：该方法同compareTo()方法一样。**返回值为0、负整数、正整数。若o1>o2则返回正整数，若o1<o2则返回负整数，若o1=o2则返回0。**

注意自定义外部比较器时，可以采用匿名内部类的方式，注意这里new 的是一个**Comparator()(**而不是实现Comparable接口)，并且重写的是**compare()**方法，而不是compareTo()方法。同时，**若比较对象的String类型的数据时，return 中的返回值用compareTo()方法写**，而不是compare()方法写。

：这样写是错的。

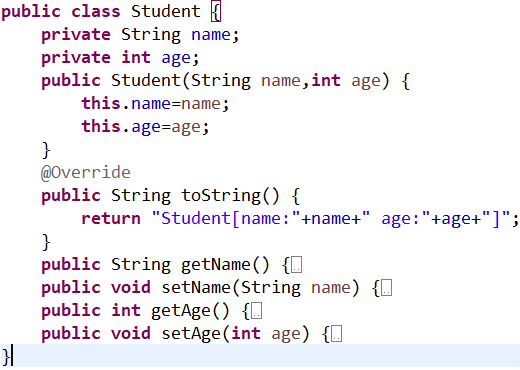
：这样写也是错的。

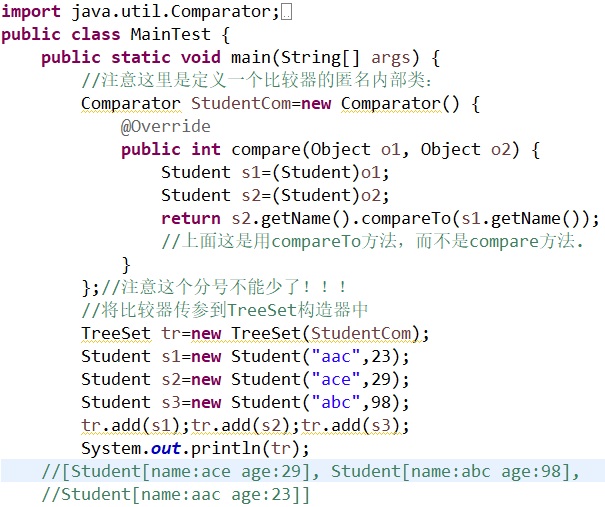
：若name是private修饰的，在自定义比较器时，这样写会报错，但是通过实现Comparable接口，这样写则不会报错。

**外部比较器的例1：匿名内部类的方式**

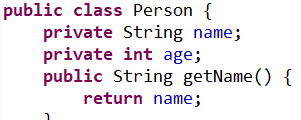


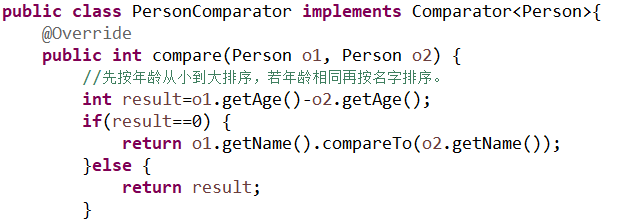
**例2：**

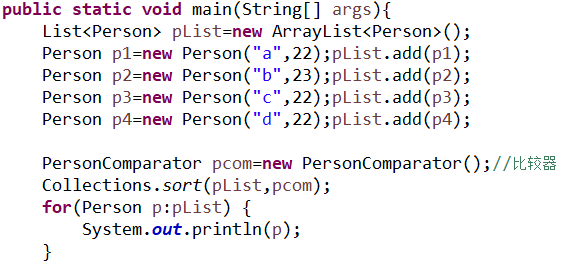




**例3：实现Comparator接口的方式**







### 3.小结

比较器的两种方法：**①对象的类实现Comparable接口，重写compareTo()方法**。**②自定义：new Comparator()匿名内部类，重写compare()方法，或者自定义一个比较器类，该类实现Comparator接口**。

## 八、Map中的HashMap

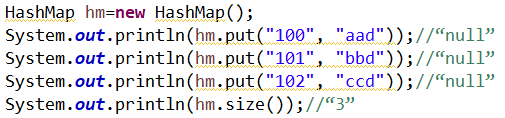
Map中的元素是以键值对(key-value)的形式存在的，**一个键只能对应一个值**， **Map中键不能重复，但是值可以**。**HashMap中键和值都可以存放null。**HashMap在原理上使用了hash表来存放数据，如果元素满了，在想要使用的节点上延伸出来一个单项链表，如果链表的长度超过8，就转化成红黑树。

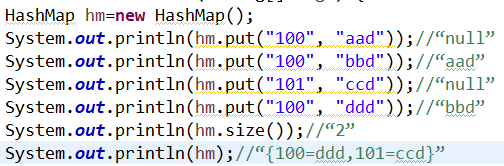
hm**.put(Object key,Object value)**：HashMap中添加元素不能用add()方法，而是用put()方法。括号内的参数为传入的键和值。要特别注意该方法的返回值。

hm.**containsKey(Object key)**：判断HashMap中是否包含指定的key，若包含，则返回true，若不包含则返回false。

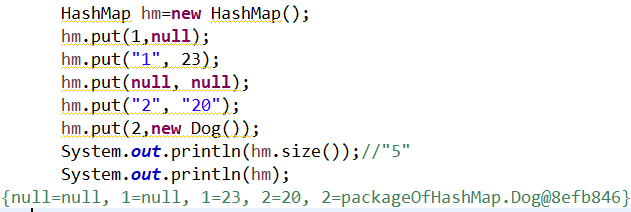
hm**.containsValue(Object value)**：判断HashMap中是否包含这个value值。若包含则返回true，不包含则返回false。

认真对比下面的输出结果可以发现：**当put()括号内的key没有相同的时候，返回值都是null，而且都能添加到HashMap中**。**而当put()括号内的key有相同的时候，该key第一次出现时put方法的返回值为null，而后面出现同样的key时，put()方法的返回值为上一次出现的key所对应的value，而且只有最后一次出现该key所对应的value才能被添加到HashMap中**。





**在HashMap中可以存放任意数据类型的数，且也有一套自己的元素排列顺序**。**HashMap中存放的key、value的数据类型都可以不一样**。

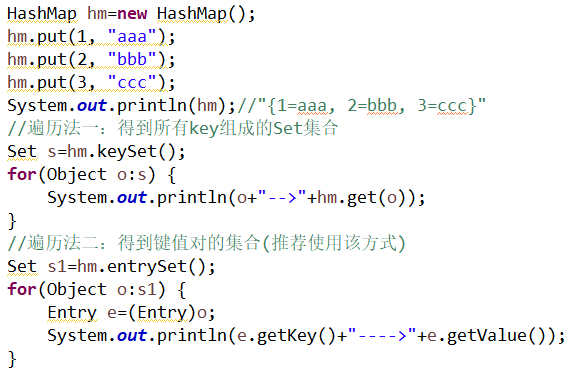


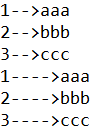
hm**.get(Object key)**：返回值为key所对应的值。如果括号内的参数key在HashMap中没有，返回值则为null。

hm**.keySet()**：括号内无参数，**返回值为该HashMap中所有key所组成的Set集合类型**。

hm**.entrySet()**：括号内无参数，**返回值为该HashMap中的key、value所组成的Set集合**。

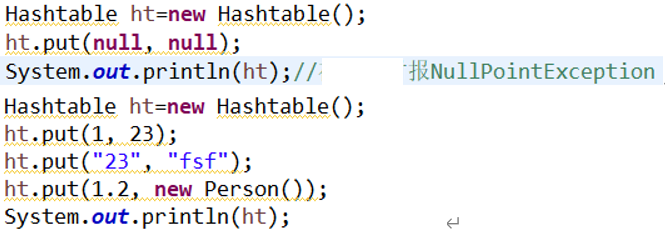
遍历HashMap的方法，法一：hm.keySet()；法二：hm.entrySet()





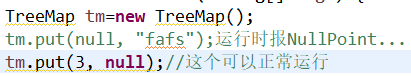
## 九、Map中的Hashtable

**注意Hashtable中的t是小写的**。其用法和HashMap 的用法差不多。**Hashtable是线程安全的，而HashMap则是线程不安全的**。注意**Hashtable的键和值都不能为null**。其遍历的方法也是keySet()方法和entrySet()方法。**存放的数据类型可以是的不兼容**。Hashtable的扩容是0.75倍。



## 十、Map中的TreeMap

TreeMap中的key不能为null，但是value可以为null。HashMap、Hashtable、TreeMap中的put()方法是一样的作用效果，详细看HashMap中的介绍。



**TreeMap中存放的元素必须是可兼容类型的，否则运行时会报错**。同时**TreeMap中元素的顺序是按照key的自然顺序排列的，数字从小到大，字母按ASCII码值排序**。当TreeMap中存放了对象时，其排序有两种方法，法一：实现Comparable接口，重写compareTo()方法；法二：自定义一个比较器，new Comparator，重写compare()方法。详细看TreeSet中的介绍。

## 十一、总结

**1.List：是有序列表的集合，List集合中都有下标索引。可存放重复元素、不同类型的元素，可存放null。其中：ArrayList和LinkedList都是线程不安全的，Vector和Stack都是线程安全的。**

**2.Set：是无序不可重复的集合，都没有下标索引。**

**⑴HashSet：可以存放null，存放的数据类型可以不兼容，线程不安全的，底层HashMap。**

**⑵TreeSet：存放的元素类型必须兼容，不能存放null，线程不安全的，底层TreeMap。**

**3.Map：key-value(Key不能重复，value可以重复)**

**HashMap：键和值都可以存放null。存放的数据类型可以不兼容，线程不安全的。**

**Hashtable：键和值都不能为null，存放的数据类型可以不兼容，线程安全的。**

**TreeMap：键key不能为null，值value可以为null。存放的数据类型必须是可兼容的，线程不安全的。**

**注意List、set中添加元素都是用add()方法(除了Stack用push()方法)，Map中添加元素用put(key，value)方法。**

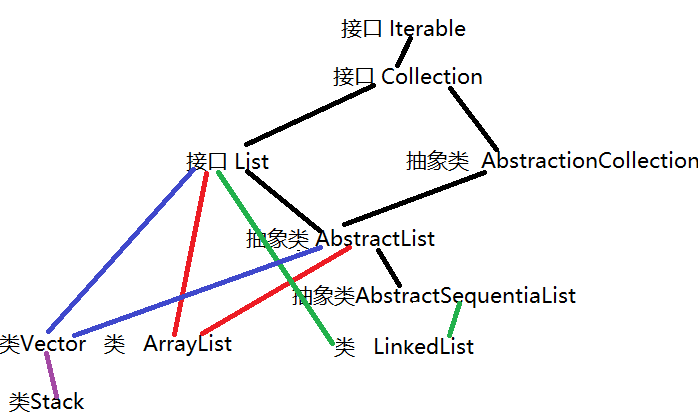
**ArraList和LinkedList的区别：两者都实现了List接口，都属于有序列表集合，都是线程不安全的。其中ArrayList的底层是Object类型的数组，其查找效率高；而LinkedList的底层是双向链表，其增删操作的效率高。**

**ArrayList和Vector的区别：ArrayList是线程不安全的，它的方法没有被synchronized关键字修饰。Vector是线程安全的，它的方法有被synchronized关键字修饰。建议在单线程访问时使用ArrayList，效率更高；多线程访问使用Vector，因为它是线程安全的。在扩容方面，ArrayList默认是扩容1.5倍。Vector默认扩容为2倍。ArrayList和Vector都实现了List接口，都是有序列表集合，都可存放重复的元素，都可存放null。**

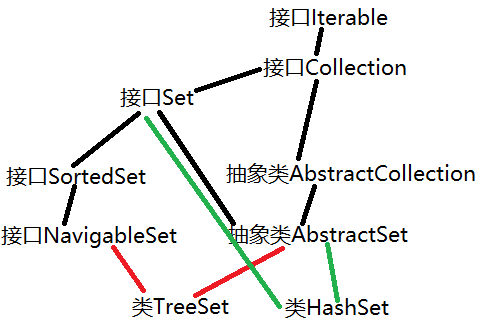
**HashMap和Hashtable的区别**：**都实现了Map接口，都属于Map类型的。其中HashMap的key、value都可以为null，而Hashtable的key、value都不能为null。HashMap是线程不安全的，而Hashtable是线程安全的。因此在单线程时建议使用HashMap效率更高，在多线程时使用Hashtable**。

**集合中的继承关系图：**

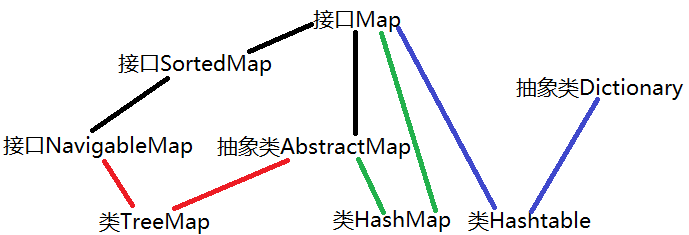
**①List**



**②Set**



**③Map**



## 十二、泛型

泛型的英文Generic，generic的意思是通用的，泛型可以理解为广泛适用的类型，可以称为参数化类型、类型参数，或者说将类型当作参数传递给类或者方法。

泛型提高了代码的可读性，可以直接看出代码要操作的数据类型，同时可以减少类型的强制转换。

### 1.泛型的类型参数

在Java中，经常用T、E、K、V等单个大写字母的形式来表示泛型的类型参数。

**①T：代表一般的任何类**

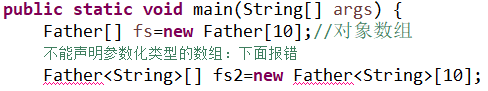
**②E：代表 Element 的意思，或者 Exception 异常的意思。**

**③K：代表key的意思。**

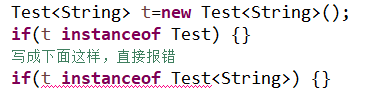
**④V：代表value的意思，一般和K配合使用**。

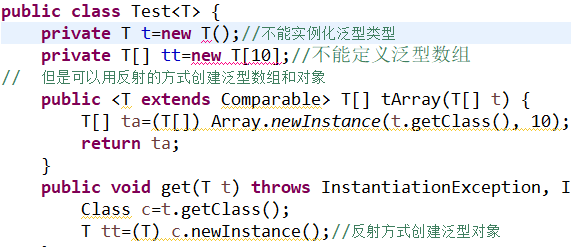
**⑤？：通配符，表任意类型**。

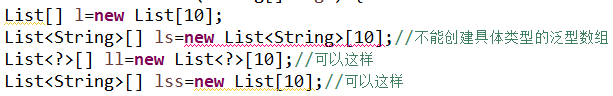
**java中的泛型类型参数不能是基本数据类型，可以是基本数据类型对应的包装类**。不能在try...catch()中catch括号中使用泛型变量。不能声明参数化类型的数组。



不能实例化泛型、实例化泛型数组，也不可以创建具体泛型类型的数组，但是可以用反射的方式创建泛型数组和对象。不能对确切的泛型类型使用instanceof操作。**但可以使用通配符创建泛型数组**。

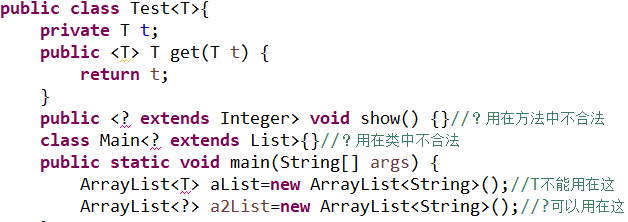




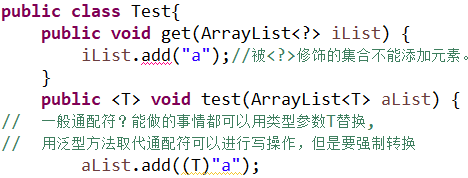


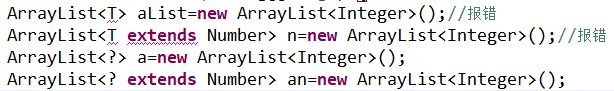
**⑴T和？的区别**

**①**T用于泛型类和泛型方法的定义，不能用于调用代码。？用于泛型方法的调用代码和类型参数。T往往是一个确定的类型，而？往往是一个不确定的类型。









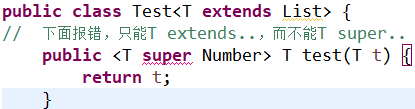
使用泛型参数T可以保证下面两个List中的元素类型是一致的。

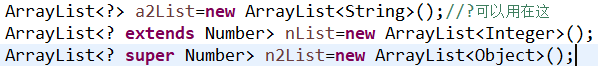


使用通配符？则不能保证下面两个List中的元素类型一致。



**②**T只有extends一种限定方式，如<T extends List>是合法的，而<T super List>是不合法的。而？有extends和super两种限定方式。





**③**T可用于多重限定，如<T extens A&B>，而？不能用于多重限定。

### 2.泛型的通配符

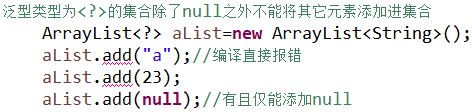
**⑴？**：无边界的通配符(Unbounded Wildcards)，**<?>**，也称无限定通配符。表参数化类型可以是**任何类**。**该<？>泛型相当于是其它所有泛型的父类**。

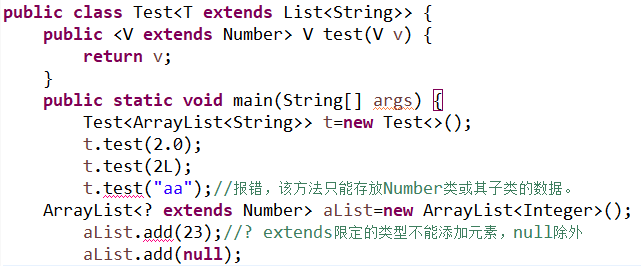
**⑵<? extends T>**：**表类型的上界，表参数化类型可以是T类或者T的子类**。

**⑶<? super T>**：**表类型的下界，表参数化类型可以是T类或者T的父类，直至Object类**。

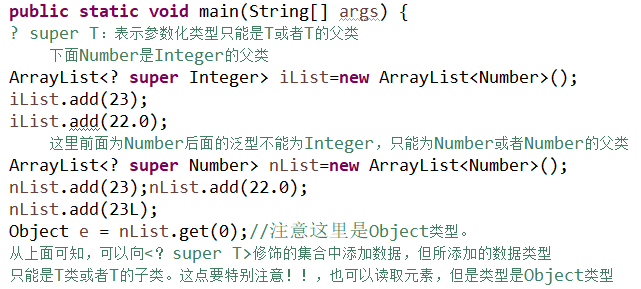
可以为一个泛型指定上界或者下界，但不能同时指定上下界。

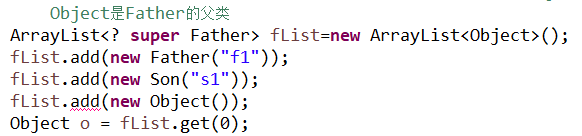
**被？声明的集合，除了null之外，不能往该集合中添加其它任何类型的元素，只能作为生产者(被迭代)**。**即集合不能调用add()或put()方法向集合中添加元素**。**null是任意引用类型的实例**。**同样，被<? extends T>限定的泛型所修饰的集合，也不能往集合中添加元素，null除外，只能读取元素**。





**注意被<? super T>所声明的集合，可以向该集合中添加元素，但所添加的元素类型只能是T类和T的子类。并且从集合中取出的元素类型为Object类型**。**同时要注意**，**添加元素只能为T类和T的子类要和参数化类型只能是T类或者T的父类区别开来**。



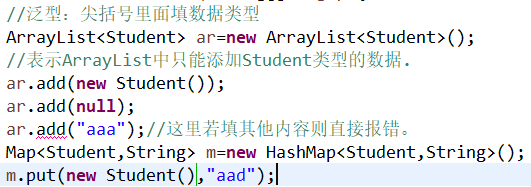


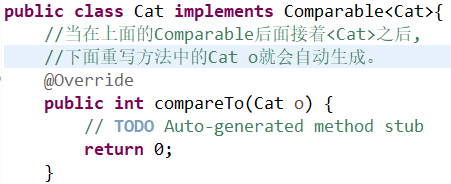
**⑷PECS法则**

生产者(Producer)使用extends，消费者(Consumer)使用super。若需要读取T类型中的元素，需要声明成<? extends T>类型，如List<? extends Number>，则不能往集合中添加元素。若要添加T类型的元素，需要声明成<? super T>类型，如List<? super Number>，可以往集合中添加Number类或者Number的子类的元素，取元素时类型为Object。若需要同时添加和和取元素，则不使用通配符？。

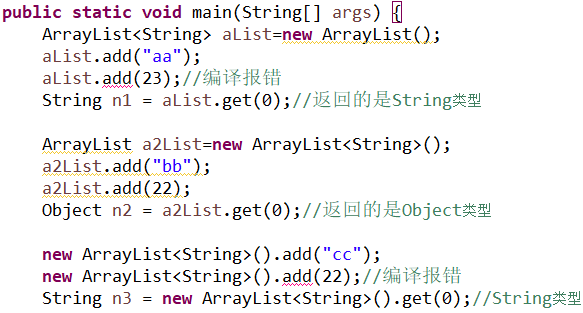
### 3.泛型的应用

#### ⑴泛型用于集合中



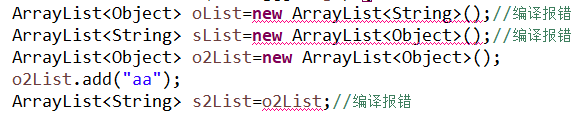


java中的编译器是通过先检查代码中类型，然后再进行泛型擦除，然后再进行编译的。**这里的类型检查针对的是引用，和引用所指向的对象无关，即不会对对象的泛型进行检查是否匹配**。



**注意泛型中参数化类型是没有继承关系的**，如Son类继承Father类，但是下面的泛型写法是错误的。注意要和<? extends T>和<? super T>的区别开来。

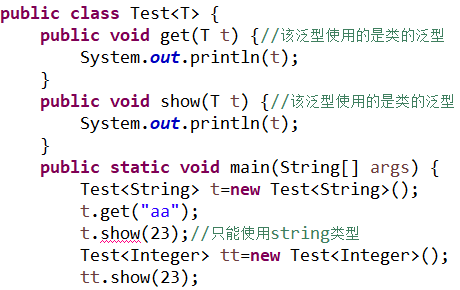




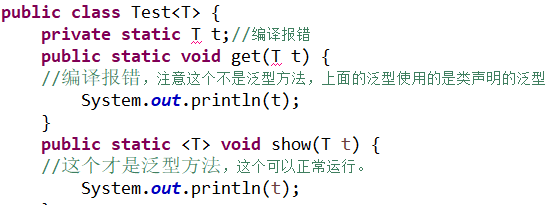
#### ⑵泛型类

**泛型类定义的泛型在整个类中有效**。**若泛型类中的方法使用了该泛型类定义的泛型，一旦泛型类创建实例对象明确了泛型类型，则方法中的泛型类型也就确定了**。

**泛型类的格式**：**在类名后面加一个<类型参数>**。泛型类可以接收多个类型参数。如public class Test<T,V>{}。



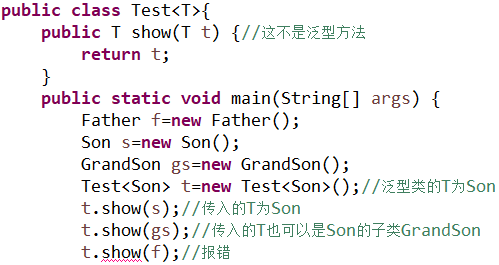
**注意泛型类中的静态方法和静态变量不能使用泛型类所声明的泛型类型参数。但是静态方法可以使用泛型方法自己声明的泛型类型参数**。

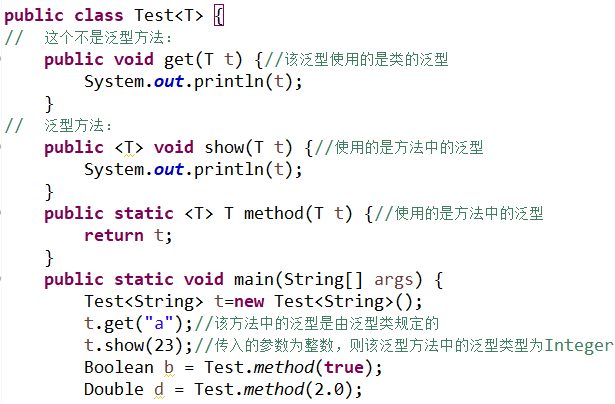


#### ⑶泛型方法

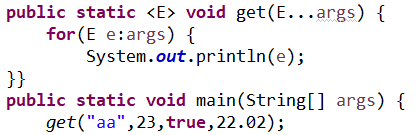
定义泛型方法可以让不同的方法操作不同的数据类型，且数据类型是不固定的。泛型方法的类型参数只能在它所修饰的泛型方法中使用。

**泛型方法的格式**：**权限修饰符 (static、final) <类型参数> 返回值类型 方法名(形参)**。如public <T> void get(T t){}，<T>中的T称为**类型参数**，而方法中的参数T称为**参数化类型**。**返回值类型也可以是泛型**。泛型方法中的泛型类型是在方法调用的时候传入的参数是什么类型该泛型类型就是什么类型。**上面的<类型参数>相当于声明该方法是泛型方法。泛型方法可以是对象的普通方法也可以是类的静态方法**。

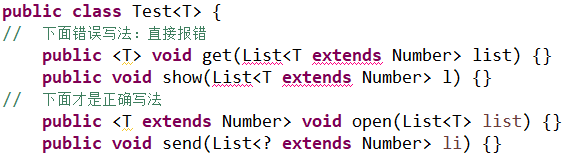




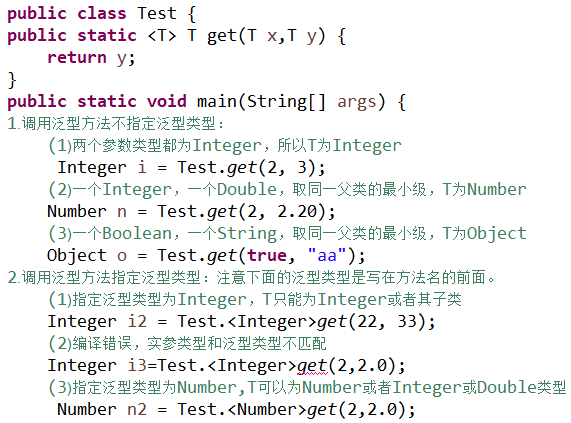
**泛型方法中的可变参数：**



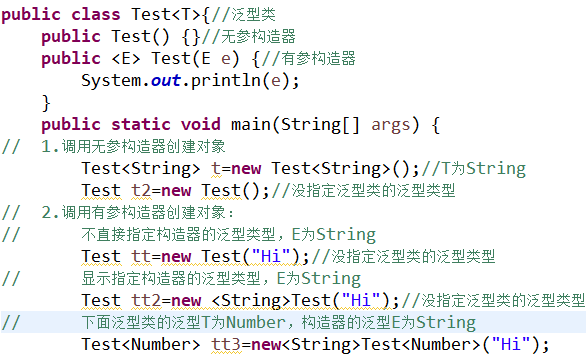
在泛型方法中添加泛型的上边界限制时，必须给<T>添加上边界限制，而不是给方法中的参数T添加上边界。

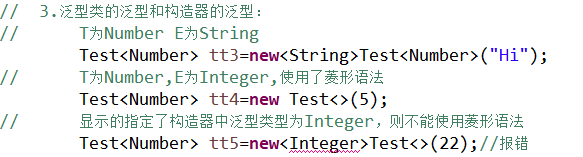


**在调用泛型方法时，可以指定泛型也可以不指定泛型**。**若不指定泛型的情况，泛型变量的类型为该方法中的几种类型的同一个父类的最小级，直到Object。若指定泛型，则该方法中的泛型变量的类型必须是该泛型实例类型或者其子类**。



**也可以在构造器中声明类型参数，其用法和泛型方法一样**。在调用构造器创建对象时，允许使用一对尖括号来代表泛型类型，该语法称为**“菱形”语法**。但若显示的指定了泛型构造器中类型参数的实际类型，则不可以使用“菱形”语法。





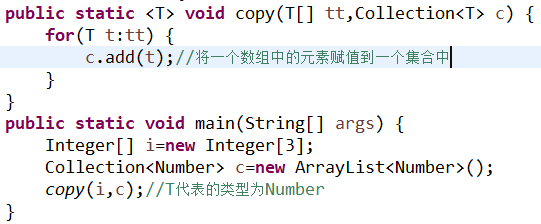
下面这三个个方法的效果是一样的。一般采用第一个方法和第三个更好，因为第二个方法的类型参数E仅在List<E>中使用了一次，其它的类型参数不依赖于它，可以直接用通配符？代替。



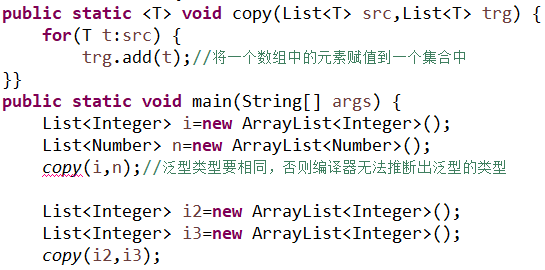




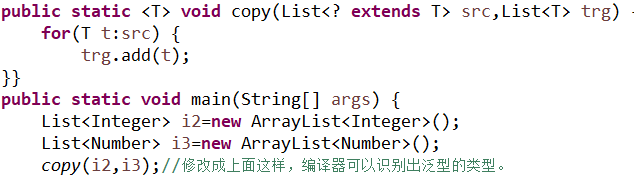
**例子：**



将上面例子修改如下：



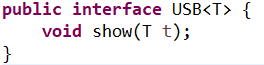
将上面例子修改如下：

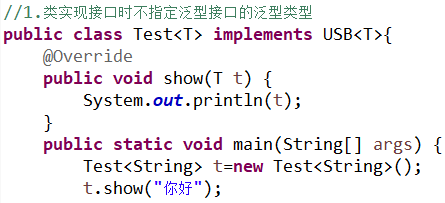


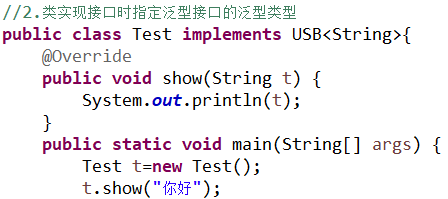
#### ⑷泛型接口

**泛型接口的格式**：**在接口名称后面加一个<类型参数>**。

当类去实现这个泛型接口时，可以指定泛型接口的泛型类型，也可以不指定泛型接口的泛型类型，若不指定泛型类型，则类也要添加一个泛型，否则报错。



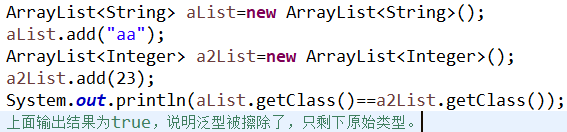


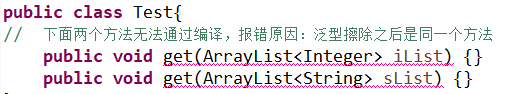


### 4.类型擦除

**java中的泛型是伪泛型。在编译期间，所有的泛型信息都会被擦除掉**。Java中的泛型基本上都是在编译器这个层次实现的。**在生成的Java字节码中是不包含泛型中的类型信息的**。**使用泛型的时候加上的类型参数，会在编译器编译的时候擦除掉，这个过程就称为类型擦除(type erasure)或者泛型擦除**。

**泛型擦除的例子：**

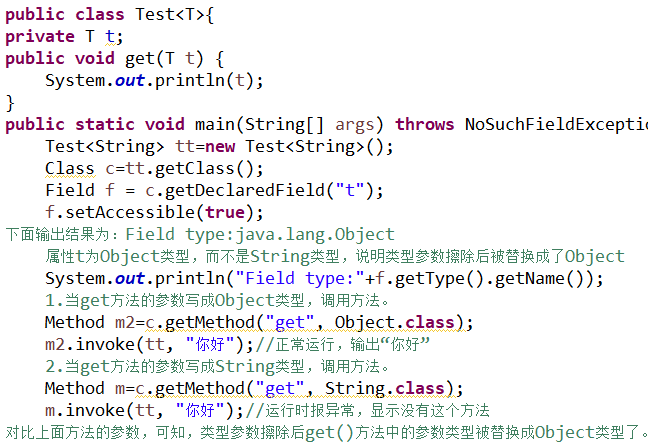


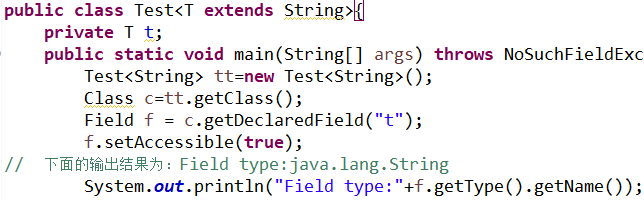


**原始类型(raw type)**：擦除了泛型信息，最后在字节码中的类型变量的真正类型。

**当泛型被擦除了，无限定的类型变量用Object替换，**如<T>替换成Object**。如果类型变量有限定，则原始类型就用第一个边界的类型变量替换，**如<T extends Comparable&Serializable>替换成Comparable类型。下面的例子很好的说明了泛型被擦除时，类型变量被替换的情况。

**例子：**





### 5.解决泛型擦除的限制

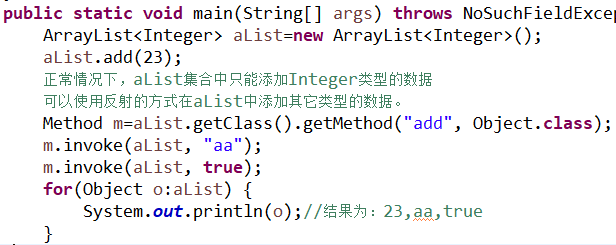
利用类型擦除的原理，用反射的手段可以绕过正常开发中编译器不允许的操作限制。

ArrayList类的源码和它的add()方法的源码：

**public** **class** ArrayList<E> **extends** AbstractList<E>**implements** List<E>.....

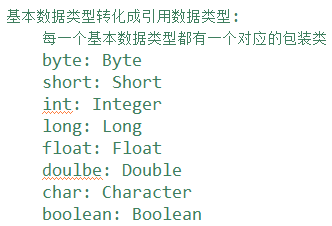
**public** **boolean** add(E e){....}

当ArrayList中的泛型擦除后，类型参数E被替换成了Object类型，此时ArrayList的方法就相当于add(Object o)。可以利用反射，绕过编译器去调用这个add()方法。



## 十三、基本数据类型的包装类(Wrapper Class)

### 1.包装类



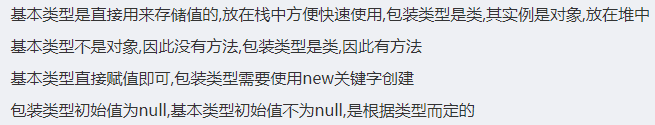
java中的数据类型int、double等不是对象，无法通过向上转型获取到Object提供的方法，而像String却可以，因为String是一个对象。为了弥补这个缺陷，java提供了包装类。包装类都为于java.lang包中。

在基本数据类型对应的包装类中，可以分为两类：**对象型包装类**(不继承任何其他类，而是Object的直接子类)、**数值型包装类**(继承于Number类)。只有Character和Boolean是Object的直接子类，而Byte、Short、Integer、Long、Double、Float则是Number的直接子类。





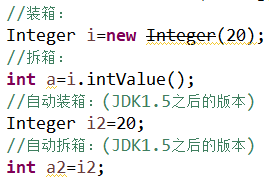
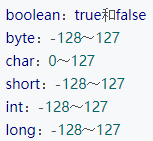


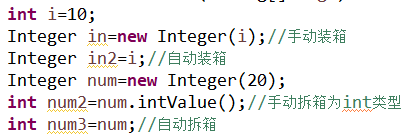


### 2.装箱和拆箱

装箱：将基本数据类型转换成包装类，默认调用**对应包装类的静态valueOf(...)方法**。可分为**手动装箱**和**自动装箱**。

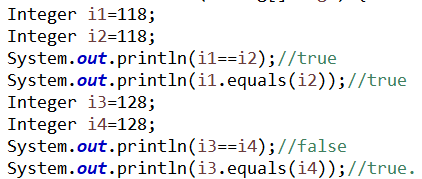
拆箱：将包装类转换成基本数据类型，默认调用的是对应包装类的**xxxValue()**，如intValue()。可分为**手动拆箱**和**自动拆箱**。

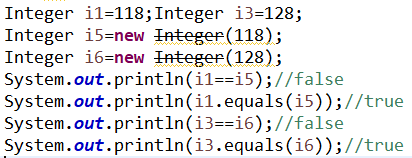
 

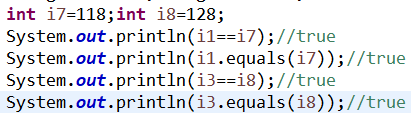


各个包装类的缓冲范围如右上。Integer类型的缓冲范围是**-128~127**。**对于上述缓冲范围，在自动装箱的过程中，在调用valueOf()方法时规定，只有不超出缓冲范围，自动装箱都不会创建对象，而是直接在类似常量池中取数据，若超出了范围，则自动创建对象，即自动用new方法在堆中创建。**

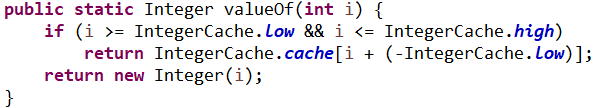
**当整型Integer的数据取值的范围在-128~127之间时，自动装箱则不会创建对象，而是类似String类型，从常量池中取数据，下面的i1和i2是同一个对象，因此用==比较和equals()方法比较的结果都正确。而超出了这个范围，就会再创建新对象，因此i3和i4是不同的对象，用==相比的结果为false，而用equals()方法相比，Integer类似String都重写了equals()方法，比较的结果为true。注意，对于手动装箱，即采用new进行包装类创建时，和String类型用new创建新对象一样，都是在堆中开辟一个新空间，然后创建对象。同时，当Integer包装类的值和int的值进行比较时，不管是使用==比较还是使用equals()方法比较，结果都为true。**

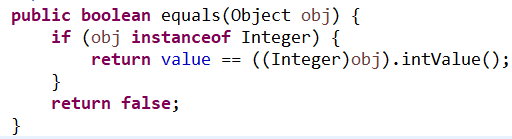






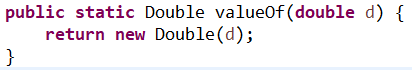
**Integer类的valueOf(int i)方法和equals()方法源码如下：**

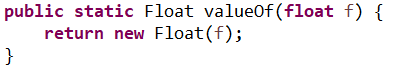




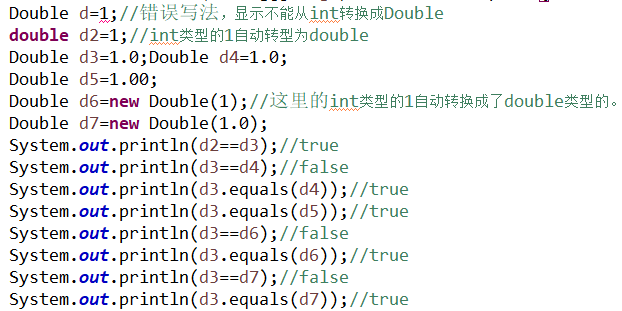


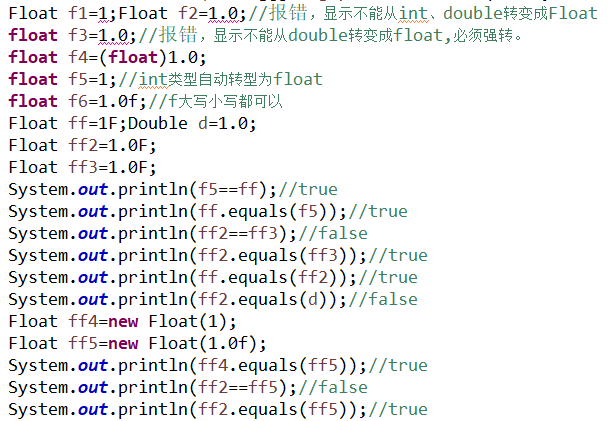
**Double和Float没有缓冲范围**。**在自动装箱过程，调用Double类和Float类的valueOf()方法都是直接创建一个对象**。**因此，Double类和Float类的数据比较都不能用==号比较，只能用equals()方法比较。**其对应的源码如下：





**例子：**



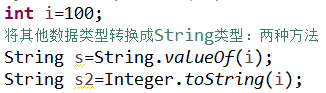


**包装类的编码规范**：**所有的相同类型的包装类对象值之间的比较全部用equals()方法**。基本数据类型的包装类都重写了equals()方法。

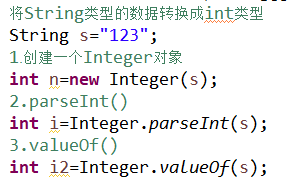
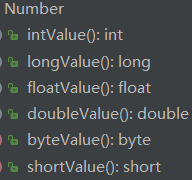
### 3.String和基本数据类型相互转换

**将其他数据转换成String类型：①使用String.valueOf(基本数据类型)方法；②其他数据类型的包装类.toString()方法；③用一个空字符串加上基本数据类型，得到的就是基本数据类型对应的字符串。**

**将String类型转换成基本数据类型：①根据字符串创建一个Integer对象，然后自动拆箱。②调用包装类的parseXxx (String s)。③调用包装类的valueOf(String s)方法转换成包装类，然后自动拆箱成基本数据类型。**





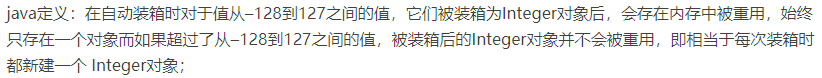
 

### 4.int和Integer的区别

一、int是基本类型，直接存数值；而Integer引用数据类型

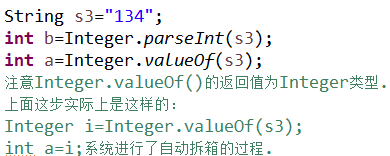
二、Int的声明不需要实例化，且变量声明后的初始值为0；Integer的是一个类，初始值为null，需要进行实例化，才能对变量数据进行处理。

三、Integer类是int的包装类，实际开发中Integer被看成一个对象，可以进行数据转换等操作。



**Integer.valueOf(String s)：将String类型转换成Integer类型，返回值为Integer类型的数据**。记忆，String.valueOf(xxx)返回的是String类型的数据。

**Integer.parseInt(String s)：将String类型转换成int类型，返回为int类型的数据。**



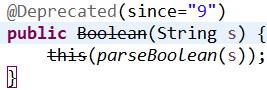
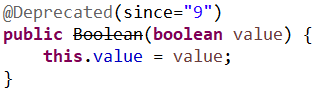
### 5.字符串转Boolean

字符串转数值类型时，字符串只能包含数字，否则会抛出NumberFormatException异常。而**字符串转boolean时，parseBoolean()方法会将“true”转换成true，而将非“true”类型的都转换成false**。

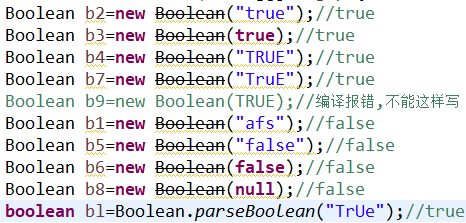
包装类的好处：可以在对象中定义更多的功能方法操作该数据。

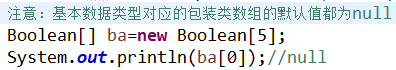
注意这些包装类**都被final修饰**，表明这些包装类都不能被继承。

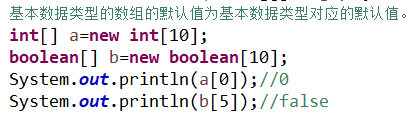
八种基本数据类型对应的包装类(除了char的包装类Character只有一个构造器之外，该构造器中的形参为对应数据类型的值)都有两个构造器，一**个构造器中的形参为对应数据类型的值，一个构造器中的形参为String类型**。没有无参构造器。



**boolean类型的包装类Boolean，当创建一个Boolean对象时**，**若构造器的形参是String类型的，只要传入的String的实参为true(忽略大小写，即TRUE、TruE等也可以)，其结果就为true。其他情况的结果都为false，包括实参为null结果也是false。同样，Boolean.parseBoolean(String s)的方法也是一样的，当该String类型忽略大小写时用equals()方法比较和”true”相等时得到的都是true，其他情况都是false。**







**注意所有的包装类以及String类型都是被final修饰的，都是不允许被继承的。**

注意：基本数据类型中其他任何数据类型都不能和boolean类型转换，即使强转也不行，编译会报错。



## 十四、迭代器

[2.迭代器](#_2.迭代器)

### 1.Iterator

Iterator(迭代器)是一个接口，它包含三个方法：hasNext()、next()、remove()。其源码如下：

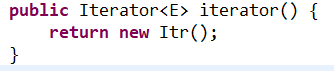
**public** **interface** Iterator<E>{...}：接口

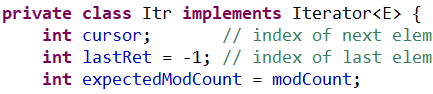
①**boolean** hasNext()：判断迭代的指针下面是否还有元素，若有则返回true。

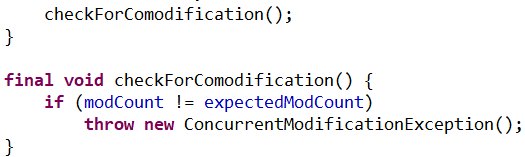
②E next()：返回当前迭代的元素，同时迭代的指针向后移动。

③**default** **void** remove(){....}：删除迭代器中返回的元素。

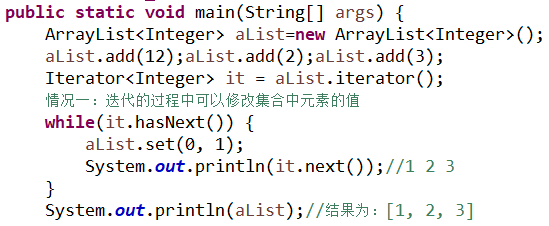
**注意在使用Iterator遍历集合中的元素时，不可以采用集合的add()和remove()方法向集合中添加或删除元素。因为ArrayList中有一个变量modCount是用来记录集合的修改次数，初始值为0。每当集合调用自己的add()或remove()方法，该modCount的值就+1。在采用迭代器遍历集合时，modCount的值首先赋值给一个expectedModCount变量，在迭代遍历的过程中只要调用了集合自身的add()或remove()方法，则modCount的值会改变。同时在迭代遍历过程中，会调用ArrayList集合中的一个checkForComodification()方法，该方法是用来判断遍历过程中集合是否被修改的，只要modCount的值不等于expectedModCount的值，则该方法就会抛出一个异常。**部分源码如下：

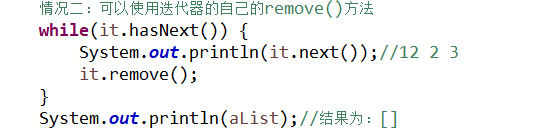


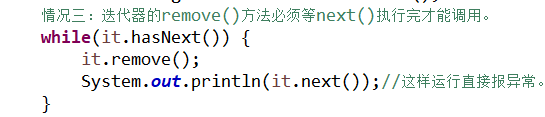


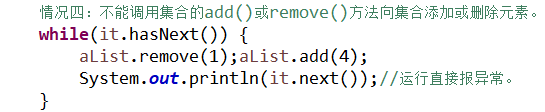


**注意在使用迭代器遍历集合的过程中，是可以修改集合中元素的值，也可以使用Iterator的remove()方法删除元素。但不能使用集合的remove()方法。在使用迭代器的remove()方法时，要注意该remove()方法必须等next()执行完才能调用，否则报错**。如迭代过程中要删除第一个元素，不能直接remove()，要先调用一个next()才可以。









### 2.Iterable

Iterable也是一个接口。实现了这个接口的集合对象是可以迭代的。部分源码如下：

**public** **interface** Iterable<T>{...}

Iterator<T> iterator();

### 3.遍历方法的选择

采用ArrayList对随机访问比较快，而for循环中的get()方法，采用的即是随机访问的方法，因此在ArrayList里，for循环较快。

采用LinkedList则是顺序访问比较快，iterator中的next()方法，采用的即是顺序访问的方法，因此在LinkedList里，使用iterator较快。(这个的正确性有待考虑)

**for循环适合访问顺序结构，可以根据下标快速获取指定元素。Iterator 则适合访问链式结构，因为迭代器是通过next()等来定位的，可以访问没有顺序的集合。**

### 4.Fail-Fast(快速失败)机制

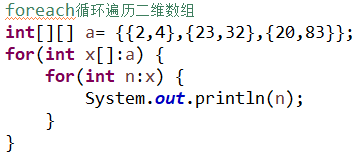
fail-fast机制是java集合中的一种错误机制。

在遍历集合的过程中，除非通过迭代器自身的remove()对列表结构进行修改，否则在其他线程中以任何形式对列表进行修改，迭代器马上会抛出ConcurrentModificationException异常。**该机制目的就是为了防止多线程下迭代的不安全操作**。(有空时候深入学习一下)

### 5.Iterator和ListIterator的区别

Iterator可以遍历Set和List，而ListIterator只能遍历List。Iterator遍历只能向前遍历，而ListIterator可以在两个方向遍历。ListIterator也是一个接口，并继承Iterator接口。ListIterator中还有额外的一些功能，如向List中添加元素，获得上下元素的索引位置等方法。(有空学习深入学习一下ListIterator)

### 6.forEach循环



## 十五、Collections

Collections类中包含了各种有关集合操作的静态方法，Collections类中包含的方法全部都是静态方法。

**Collections.sort(List<T> list)**：使用该方法对集合进行排序，要注意两点：①集合中的元素的类型要是同一类的，即被泛型定义好的。②该元素类型的类必须继承了Comparable内部比较器。

[**Collections**](eclipse-javadoc:%E2%98%82=testPro/E:%5C/JavaRunEnviroment%5C/lib%5C/rt.jar%3Cjava.util(Collections.class%E2%98%83Collections)**.sort(**[**List**](eclipse-javadoc:%E2%98%82=testPro/E:%5C/JavaRunEnviroment%5C/lib%5C/rt.jar%3Cjava.util(Collections.class%E2%98%83Collections~sort~Ljava.util.List%5C%3CTT;%3E;~Ljava.util.Comparator%5C%3C-TT;%3E;%E2%98%82java.util.List)**<**[**T**](eclipse-javadoc:%E2%98%82=testPro/E:%5C/JavaRunEnviroment%5C/lib%5C/rt.jar%3Cjava.util(Collections.class%E2%98%83Collections~sort~Ljava.util.List%5C%3CTT;%3E;~Ljava.util.Comparator%5C%3C-TT;%3E;%E2%98%82T)**> list,** [**Comparator**](eclipse-javadoc:%E2%98%82=testPro/E:%5C/JavaRunEnviroment%5C/lib%5C/rt.jar%3Cjava.util(Collections.class%E2%98%83Collections~sort~Ljava.util.List%5C%3CTT;%3E;~Ljava.util.Comparator%5C%3C-TT;%3E;%E2%98%82java.util.Comparator)**<? super** [**T**](eclipse-javadoc:%E2%98%82=testPro/E:%5C/JavaRunEnviroment%5C/lib%5C/rt.jar%3Cjava.util(Collections.class%E2%98%83Collections~sort~Ljava.util.List%5C%3CTT;%3E;~Ljava.util.Comparator%5C%3C-TT;%3E;%E2%98%82T)**> c)**：一般定义了一个外部比较器，然后要对集合中的元素按照比较器中定义的方式排序时，采用这种方法。

注意Collections.sort()方法只能对List集合进行排序，从上面的形参中就可以看出。不能用来对Set中元素排序。

## 十六、克隆Cloneable

**标记接口(marker-interface)**：没有定义任何方法，是一个空接口，如Cloneable接口和Serializable接口。标记接口的作用是告诉JRE这些接口的实现类是否具有某个功能，如是否支持克隆、是否支持序列化等。

### 1. clone()方法

java中的clone功能，它允许在堆中克隆出一块和原对象一样的对象，并给这个新对象的内存地址赋值给一个新的引用。Java 中 一个类要实现clone功能 必须实现 Cloneable接口，否则在调用 clone() 时会报 **CloneNotSupportedException** 异常。

java中所有的类都默认继承java.lang.Object类，Object类中有一个clone()方法，这个方法将返回一个Object对象的拷贝。**注意两点**：**①原对象拷贝出来的是一个新对象，而不是对象引用，即clone()方法复制的对象和原来的对象是同时独立存在的；②拷贝出来的新对象与new创建的新对象的区别：前者已经包含了一些原来对象的信息，new创建的对象只包含初始化的信息**。注意：一个类只是重写了Object类的clone()方法而没有实现Cloneable接口，则会抛出异常。下面是Object类的clone()方法的源码：

**protected** **native** Object clone() **throws** CloneNotSupportedException

**为什么采用Object类的clone()方法克隆一个对象，而不是采用new创建一个新对象，并将原对象的内容赋值给新对象？**答：**Object类的clone()方法是一个native修饰的方法，它可以直接操作内存中的二进制流，所以native方法的运行效率一般都是远高于java中的非native方法**。

**注意Object类的clone()方法是被protected修饰的，一个类实现Cloneable接口并重写Object的clone()方法之后，为了让其它类能调用这个类的clone()方法，需要将这个类的clone()的属性改为public**。

源码中关于Object类的clone()方法有如下的注解：对于任何对象Object x：

**①x.clone() != x will be true**

**这说明克隆出来的对象和原对象不是同一个对象，保证克隆出来的对象将有单独的内存地址分配**。

**②x.clone().getClass() == x.getClass() will be true, but these are not absolute requirements**.

**这说明原对象和克隆出来的对象具有相同的类的类型，但这不是强制要求**。

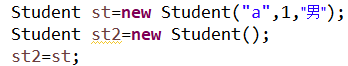
**③x.clone().equals(x) will be true, this is not an absolute requirement**.

**这说明原对象和克隆出来的对象使用equals()比较结果是相等的，但这不是强制要求。默认情况下是不相等的，除非这个对象的类重写了equals()方法，没有重写equals()方法比较的是两者的内存地址，由于原对象和克隆出来的新对象的内容地址不同，所以equals()的结果只能是false**。

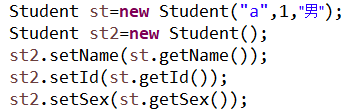
### 2.克隆对象的方式

**⑴方法一**：采用直接赋值的方式，效率低。这种严格来说是复制。

①这种方式其实是将原对象引用指向的地址赋值给一个新对象的引用，原引用和新的引用实际上指向的是同一个对象。当新的引用改变对象的属性值，原对象的属性值也会改变。



②下面这种方式效率低。



**⑵方法二**：类实现Cloneable接口，重写Object类的clone()方法，将clone()方法的访问权限由protected修改为public。当调用该类的对象的clone()方法时，实际上调用的是clone()方法中的super.clone()方法来得到需要的复制对象。**这种方式可分为深克隆和浅克隆**。

**⑶方法三**：类实现Serializable接口，通过序列化和反序列化的方式，将对象写入流中，写到流中的对象是原始对象的一个拷贝，而原始对象仍然存于内存中，再从流中将对象读出来。**这是深克隆**。**注意对象中的引用类型的成员变量对应的类也要实现序列化接口**。

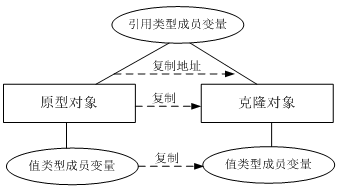
### 3.克隆的分类

**好好理解深克隆、浅克隆的例子**

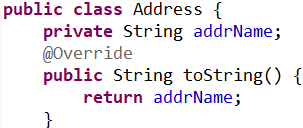
#### ⑴浅克隆(Shallow Clone)

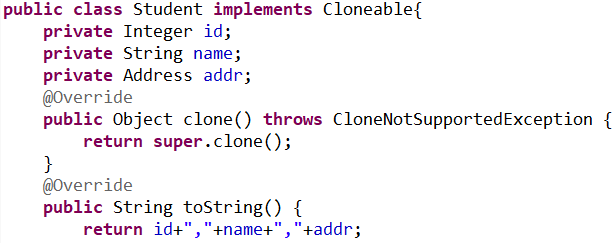
**在浅克隆中，若原始对象的成员变量是基本数据类型，则直接复制一份给克隆对象；若原始对象的成员变量是引用类型，则将引用对象的地址复制一份给克隆对象，即原始对象和克隆对象的成员变量指向相同的内存地址**。

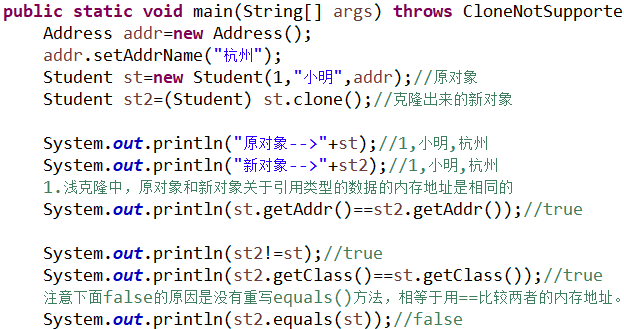
简单理解：浅克隆中，当对象被复制时，相当于创建一个同类型的新对象，该新对象的成员变量中，基本数据类型的变量的值和原对象的值相等，而引用类型的变量和原对象的变量指向相同的内存地址。

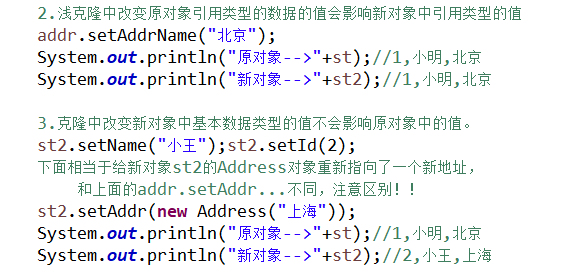


**浅克隆的例子：**



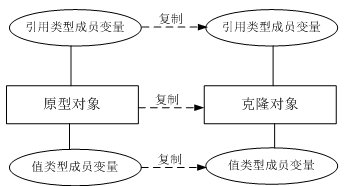






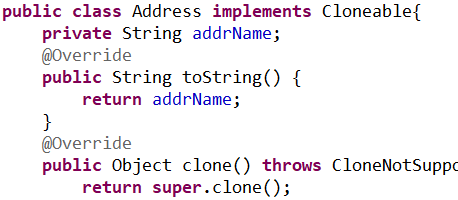
#### ⑵深克隆(Deep Clone)

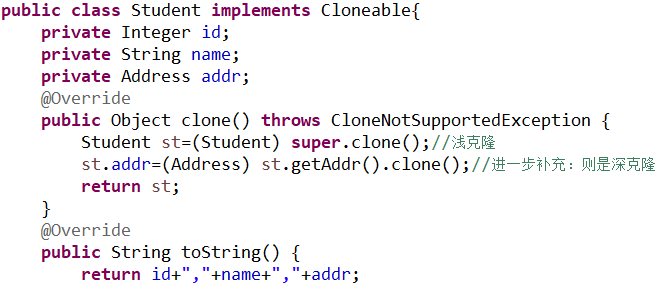
**在深克隆中，无论原始对象的成员变量是基本数据类型还是引用类型，都将直接复制一份给克隆对象，深克隆将原始对象的所有引用对象也复制一份给克隆对象**。即**不管修改原对象的任何类型的成员变量的值都不会影响克隆出来的新对象**。

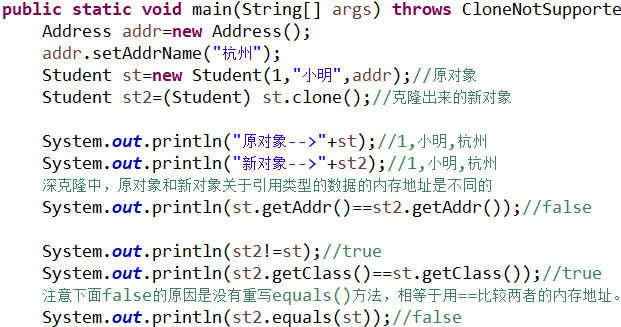


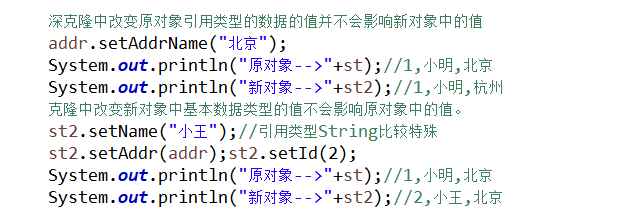
不管是深克隆还是浅克隆都可以采用实现Cloneable接口，重写Object类的clone()的方法实现。但是，当一个类的成员变量中的引用类型里面还包含很多引用类型，或者说内层引用类型的类里面又包含引用类型，这时采用clone()方法实现深克隆就会很麻烦，这时可以采用序列化的方式实现深克隆。

**深克隆采用clone()的例子：**

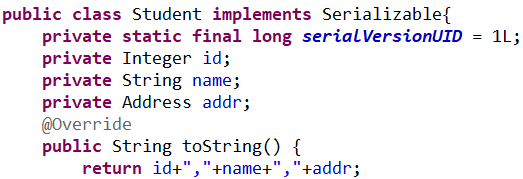


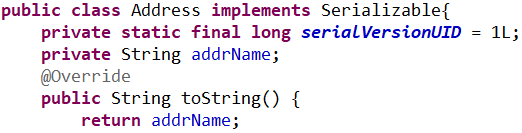


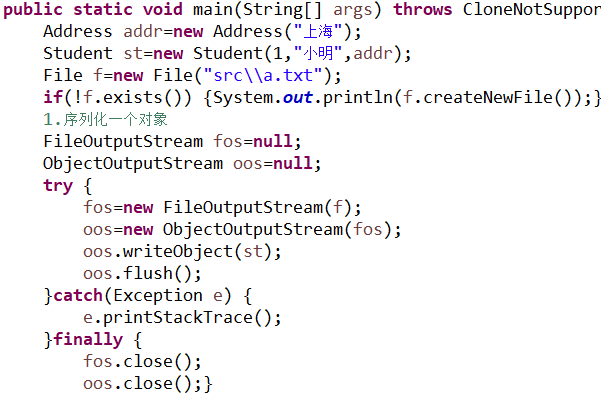


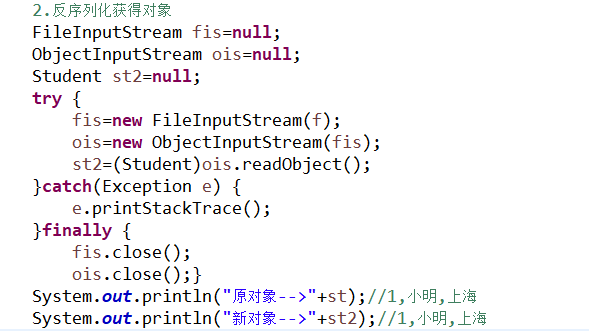


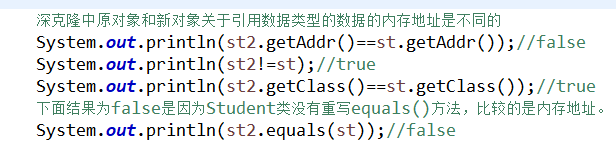
**深克隆采用序列化的例子**：

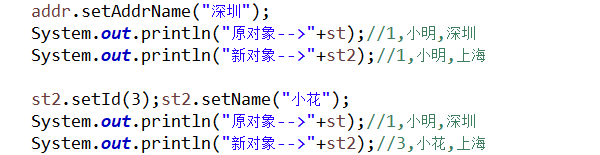






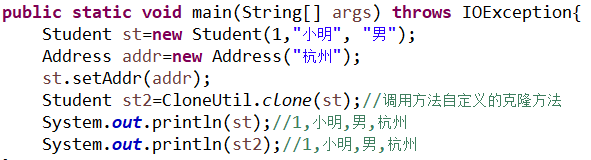


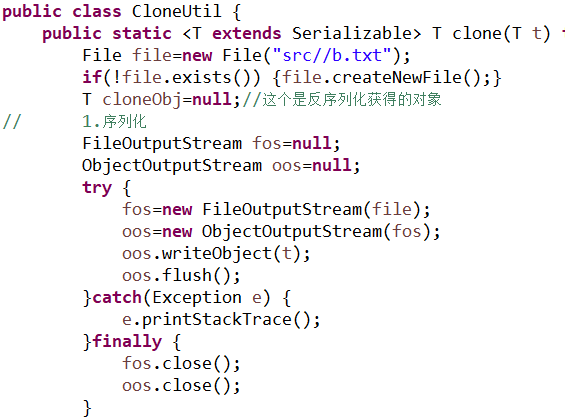


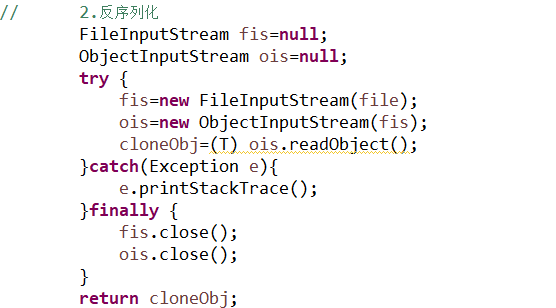


**优化深克隆采用序列化的例子(重点掌握)：**

将上面的例子进行改进，将序列化和反序列化的操作封装为一个方法。然后在测试中调用这个自己定义的clone()方法对对象进行克隆。







## 十七、Serializable接口

将一个类的对象实现序列化，要将该类实现Serializable接口或者Externalizable接口，**注意transient关键字在Externalizable中不起作用**。Serializable接口也是一个标记接口，这个接口中没有定义任何成员方法和成员变量。**一个类只有实现了Serializable接口，它的对象才能被序列化和反序列化**。java提供了一种保存对象状态的机制，那就是序列化。**当试图对一个对象进行序列化时，若遇到一个没有实现java.io.Serialization接口的对象时，将抛出NotSerializationException异常**。注意在序列化的过程中，若对象的成员变量有引用数据类型，则该引用数据类型对应的类也要实现序列化接口，否则抛出异常。

平时创建出来的对象都存在于JVM中的堆（heap）内存中，只有JVM处于运行状态的时候，这些对象才可能存在。一旦JVM停止，这些对象也就随之消失。

**序列化**：**将对象转化为字节序列的过程称为对象的序列化**。

**反序列化**：**将字节序列恢复为对象的过程称为对象的反序列化。**

**什么情况下需要序列化：①当需要把对象的状态信息通过网络进行传输；②需要把内存中的对象状态保存到一个文件中或者数据库中的时候**。**③通过RMI(远程方法调用)传输对象时**。

**注意序列化保存的是对象的状态，只能保存对象的非静态成员属性，并且保存的只是属性的值，对于属性的任何修饰符都不能保存。不能保存任何的成员方法和静态的成员变量**。

**被static或者transient修饰的成员变量的数据不能被序列化，因为static修饰的数据是属于类的而不是对象的，transient修饰的属性数据代表对象的临时数据。**

为什么需要序列化和反序列化？java中，当两个进程进行通信时，可以相互发送各种数据，如图片、视频、音频等，这些数据以二进制序列的方式在网上传送，当进程中传输的是java对象时，这就需要将java对象进行序列化和反序列化了。

**java序列化的好处**：

**①**可以实现数据的持久化，永久性保存对象，将对象的字节序列保存到本地文件中或数据库中。

**②**通过序列化将对象以字节流的形式在网络中传递或接收。

**③**通过序列化在进程中传递对象。

**并非所有对象都可以序列化，原因如下**：

**①**安全方面的原因：如一个对象中被private修饰的属性，在序列化进行传输过程，这个对象的private等权限是不被保护的。

**②**资源分配方面的原因：如Socket、Thread类，若可序列化，也无法对它们进行重新的资源分配。

### 1.serialVersionUID

**序列化运行时使用一个称为 serialVersionUID 的版本号与每个可序列化类相关联，该序列号在反序列化过程中用于验证序列化对象的发送者和接收者是否为该对象加载了与序列化兼容的类。如果接收者加载的该对象的类的 serialVersionUID 与对应的发送者的类的版本号不同，则反序列化将会抛出InvalidClassException**。**可以自己给可序列化类声明一个具体的serialVersionUID属性，该属性必须是被static和final修饰的long类型的数据，最好也是被private修饰**。若可序列化类未显式声明 serialVersionUID，则序列化运行时将基于该类的各个方面(类名、成员方法、成员变量等)计算该类的默认 serialVersionUID 值。**强烈建议所有可序列化类都显示的声明serialVersionUID的值，若没有特俗要求，最好使用默认值1L，这样可以确保反序列化成功**。因为计算默认的 serialversionuid对类的详细信息具有较高的敏感性，根据编译器实现的不同可能千差万别，从而可能导致反序列化失败。

当可序列化类的serialVersionUID的值是根据类的信息如类名、成员方法、成员变量等计算出来时，将类的实例对象序列化之后，若再改变类中的信息，如给类声明一个新的成员变量，然后再进行反序列化，这时会出现序列化时对象的serialVersionUID的值和反序列化时serialVersionUID的值不一样，从而导致反序列化抛出InvalidClassException异常。

在Eclipse中生成的序列化ID有两种：

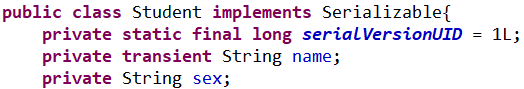
**①固定为1L。默认值。**

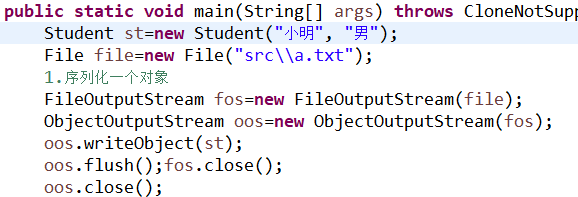
**②根据类名、接口名、成员方法、成员变量等生成一个随机的不重复的long类型的数据**。

**private static final long *serialVersionUID* = 1L；**

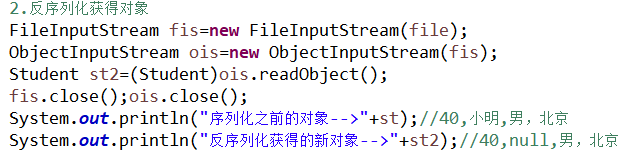
当一个类的对象通过网络进行传输时，若同一个类的serialVersionUID不一致，则反序列化就不能正常进行。

**测试1：**



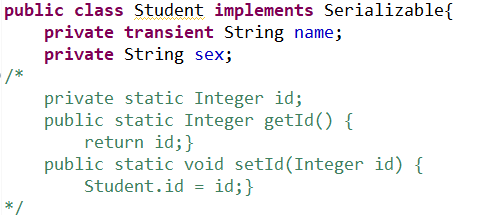


直接运行上面的代码，将对象实例序列化，之后将Student类中的serialVersionUID的值改为2L，然后运行下面的代码，结果抛出java.io.InvalidClassException异常。



**测试2：**

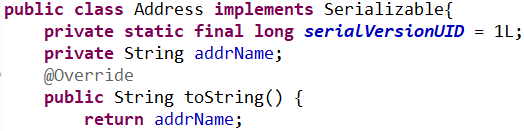
将上面的Student类改成下面这种形式，这个没有将serialVersionUID显示的声明出来，系统默认根据类名、属性、方法等计算出一个serialVersionUID的值出来。同样按照测试1中方式序列化对象。之后，将Student类中的注释部分的内容取消注释，然后执行测试1中的反序列化过程，结果抛出上面同样的异常，导致反序列化失败。这是因为反序列化时，由于Student类添加了属性，这导致系统计算出的serialVersionUID的值和原来的值不一样，从而抛出异常。

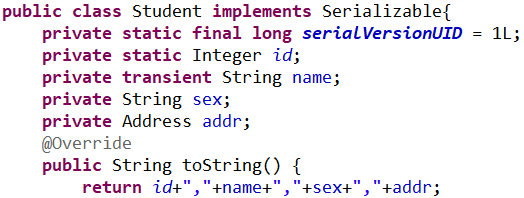


### 2.transient关键字

**作用**：阻止对象中用此关键字修饰的成员变量的序列化。即**当对象被反序列化时，被transient关键字修饰的属性的值不会被恢复，而是该属性对应的数据类型默认的值**。

**例子：**





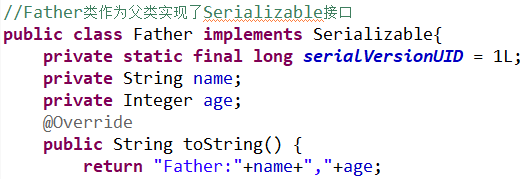


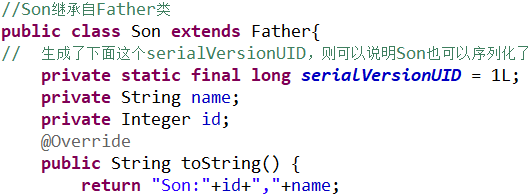


分析上面的输出结果：Student类中id属性被static修饰，name属性被transient修饰。序列化之前name的值为“小明”，序列化之后name的值为null，说明被name属性的值没有被反序列化得到，从而说明被transient修饰的属性不能序列化。在序列化之前，id的值为20，若被static修饰的id属性能被序列化，那么在输出“序列化之前的对象”这一行的输出结果中id的值应该为20，而不是40，40的值是在序列化完Student类的对象实例之后设置的，从这可说明被static修饰的属性也不能序列化。

### 3.序列化中的继承

**当一个父类实现序列化，其子类自动实现序列化，不需要显式实现Serializable接口**。





**一个子类实现了 Serializable 接口，它的父类都没有实现 Serializable 接口，要想将父类对象也序列化，就需要让父类也实现Serializable 接口。(下面的好像有点问题，自己敲代码时报异常)如果父类不实现 Serializable接口的话，就需要有默认的无参的构造函数**。这是因为创建java 对象的时候需要先有父对象，才有子对象，反序列化也不例外。**在反序列化时，为了构造父对象，只能调用父类的无参构造函数作为默认的父对象。因此当我们取父对象的变量值时，它的值是调用父类无参构造函数后的值**。在这种情况下，在序列化时根据需要在父类无参构造函数中对变量进行初始化，否则的话，父类变量值都是默认声明的值，如 int 型的默认是 0，string 型的默认是 null。

## 十八、Externalizable接口

Externalizable是一个继承了Serializable接口的接口，它包含两个方法，接口中的普通方法默认是public abstract修饰的。源码如下：

**public** **interface** Externalizable **extends** java.io.Serializable{...}

**void** writeExternal(ObjectOutput out) **throws** IOException;

**void** readExternal(ObjectInput in) **throws** IOException, ClassNotFoundException;

当一个类采用实现Externalizable接口实现序列化时，要重写writeExternal()和readExternal()方法，writeExternal()方法在对象被序列化时调用，readExternal()则在反序列化时调用。在writeExternal()方法中指定哪些属性进行序列化，readExternal()则读取序列化的属性值。

特别注意：**在使用Externalizable进行序列化的时候，在读取对象时，会调用被序列化类的无参构造器去创建一个新的对象，然后再将反序列化时调用readExternal()方法获得的属性的值分别填充到新对象中。所以，实现Externalizable接口的类必须要提供一个public的无参的构造器(访问权限必须为public)**，否则在反序列化获得对象时会抛出java.io.InvalidClassException: no valid constructor异常。

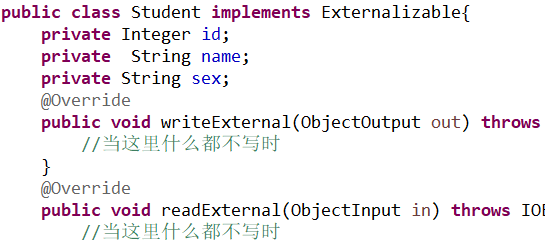
### 1.使用Serializable和Externalizable的区别：

**①Serializable序列化时不会调用默认的构造器，而Externalizable序列化时会调用默认构造器的。**

**②实现Externalizable接口的类，重写的writeExternal()和readExternal()方法中可以具体指定哪些属性进行序列化。而实现Serializable接口的类，除了被static、transient修饰的属性外，其它属性都可以被序列化。**

**③transient关键字只能在Serializable中使用，在Externalizable中使用虽然不报错，但是不会起任何作用。Externalizable中也可以将被transient修饰的属性进行序列化**。

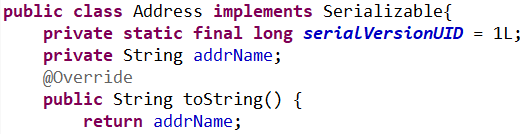
**注意若一个类同时实现了Externalizable和Serializable接口，则使用的是Externalizable进行序列化的**。

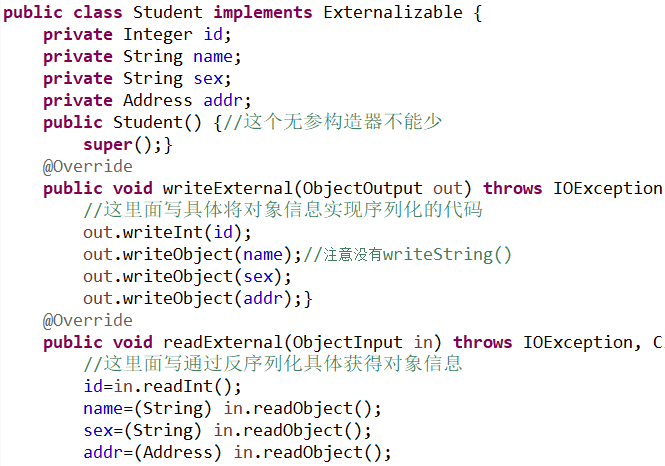


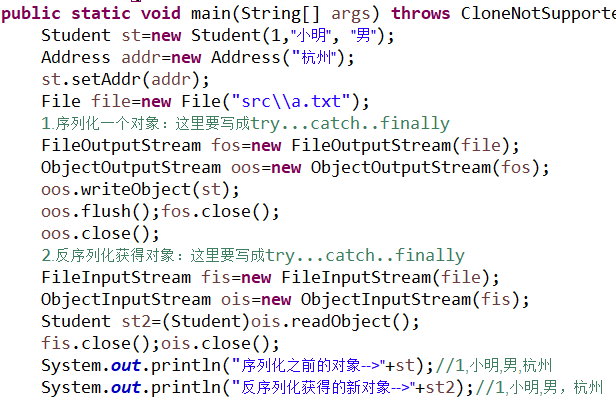
执行序列化和反序列化，得到下面的输出结果，可以发现发序列化并没有获得序列化的对象。

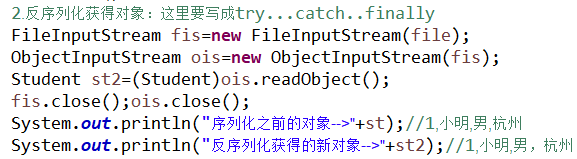


**例子：**

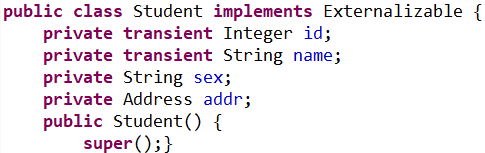


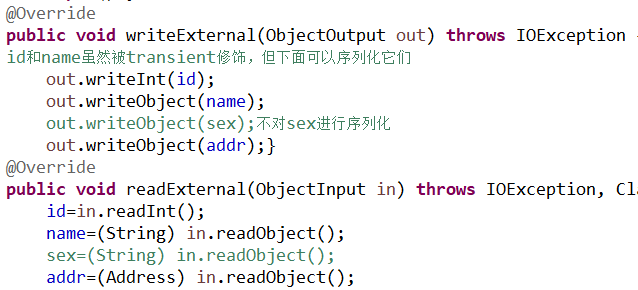






将上面的代码稍微改变一下，得到如下：





进行序列化和反序列化后，测试结果如下：说明实现了Externalizable接口的类，在对象进行序列化时，即使对象的属性被transient修饰了，也能将改属性序列化。



### 2.使用Serializable和Externalizable的优缺点

**⑴使用Serializable**

①优点：java虚拟机提供支持，易于实现。

②缺点：占用空间较大，由于额外的开销导致速度变比较慢。

**⑵使用Externalizable**

①优点：开销较少(由程序员自己决定存储什么)、可能的速度提升。

②缺点：虚拟机不提供任何帮助，需要存储什么内容的任务落到开发者身上了。