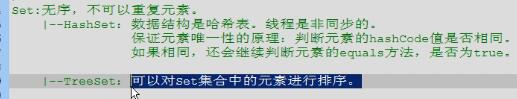
**P183 集合框架（TreeSet）**

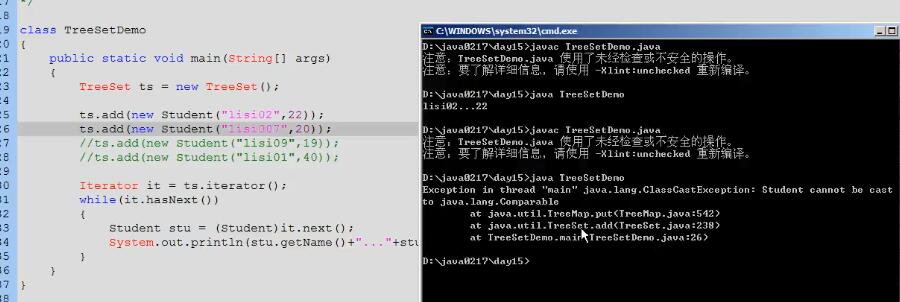


**P184 集合框架（TreeSet存储自定义对象）**

需求：往TreeSet集合中存储自定义对象学生，想按照学生的年龄进行排序。

当主要条件相同时，按照次要条件。

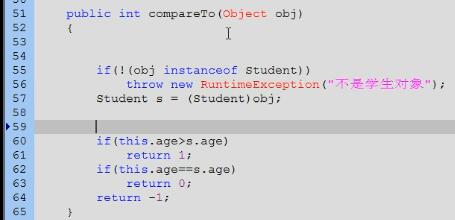
当我们往TreeSet中存入两个元素后进行编译，发现报错了。



为什么会报错：因为你将元素存入TreeSet中，TreeSet会自动帮你排序。但是自定义对象不具备比较性，没办法进行排序。所以，java对这些对象提供了Comparable接口，即要想你的元素具备比较性，那么你必须实现这个接口。

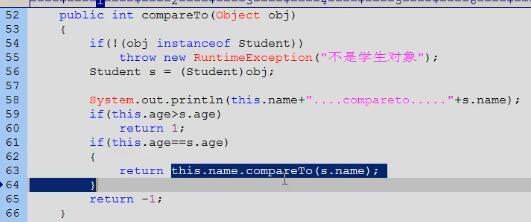
简单总结，TreeSet对元素排序的底层实现靠的就是元素的compareTo方法。所以往TreeSet里面存的元素必须具备比较性，而比较性通过Comparable接口来实现。

Comparable接口只有一个抽象方法compareTo，返回一个int，大于0大于，等于0等于，小于0小于。所以只要让自定义类实现该接口的compareTo方法即可。



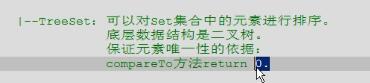
但是这么一来又有个问题，TreeSet对元素排序的底层实现靠的就是元素的compareTo方法，那么compareTo方法返回值为0的元素，TreeSet就会认为是重复元素，不存储进来。这一现实情况相违背：年龄相同不一定是同一个人，一般我们认为姓名年龄都相同才是同一个人，不用存入到Set中。

同时我们发现，String类型已经实现过Comparable接口。

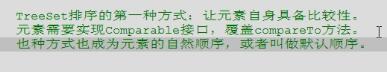


**P185 集合框架（二叉树）**

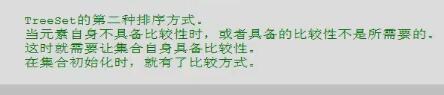
通过上面的例子我们发现，TreeSet集合框架，它每存入一个元素，都会于其他元素进行比较然后排序。如果是按顺序比较，那么会是十分低效的。所以TreeSet比较排序的底层实现结构是二叉树。



同样的，判断TreeSet是否包含某个元素，或者删除某个元素也是依赖compareTo方法实现：调用判断和删除方法，本身要传入一个元素，然后传入的元素依照二叉树的顺序调用compareTo去比较TreeSet里面的元素。如果返回0说明找到了，判断返回true删除执行操作；如果返回正数或负数，判断返回false删除不执行操作。

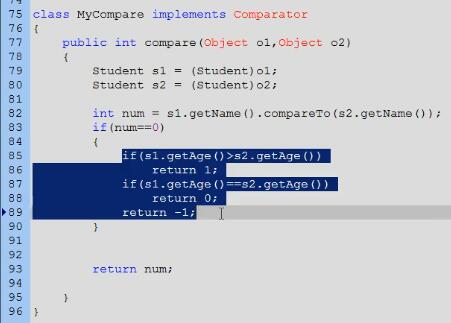


**P186 集合框架（实现Comparator方式排序）**



通过构造函数实现

而且通过比较器的权限还比较大：意思是自定义的类按年龄排序，比较器按姓名排序。最终的排序结果是按照年龄。



有个问题，接口本身全是抽象方法的集合，也就没有写好了主要的代码，暴露出一个简单的方法出来供你操作的说法。那么，你实现了Comparator接口的子类真的就只有小小的一个compare方法，它又是怎么实现这么大的功能呢？

你要这么想，该接口是并不是凭空产生，插在哪个类上都能实现的。它仅仅是由TreeSet暴露出来，只能插在TreeSet上。所以TreeSet当然在内部进行了完善合理的封装。

**P187 集合框架（TreeSet练习）**

需求：让TreeSet根据输入字符串的长短进行排序

String是final，所以不能继承String来重写compareTo方法。只能使用方法二。从这也可以看出，在实际的编程当中，方法二更常用一些。



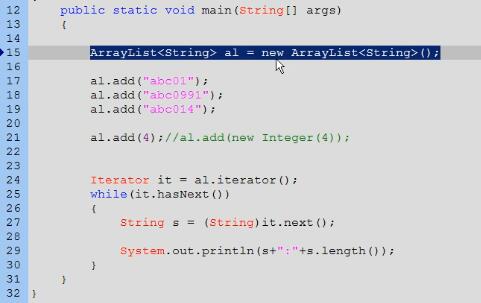
注意java语法规则，像9，10，11行，每创建一个局部变量，都要记得申明变量的类型

**P188 集合框架（泛型概述）**

JDK1.5版本以后，Integer具有自动装箱和拆箱功能。所以能实现以下操作：

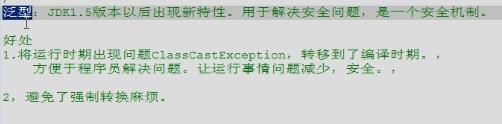
-ArrayList al = new Array List();

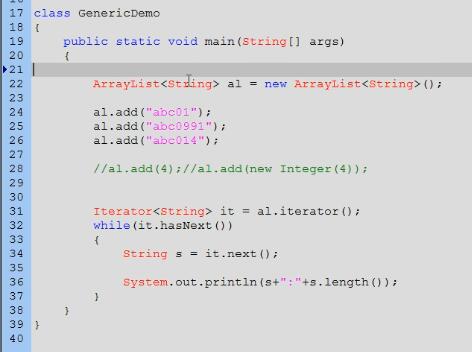
-al.add(4)// 等同于al.add(new Integer(4));因为al中只能储存对象



如果没有第15行的泛型写法，那么在编译时不会报错，在运行时会报错，存在安全隐患。

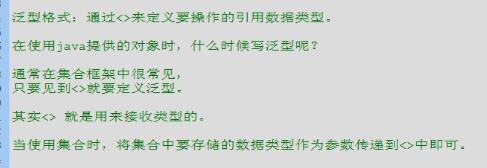
所以类似于数组，我们对ArrayList集合框架所存储的元素的类型也进行事先的规定。同时如果事先规定好存入ArrayList中的元素类型，那么迭代器的输出类型也可以不再是Object类。



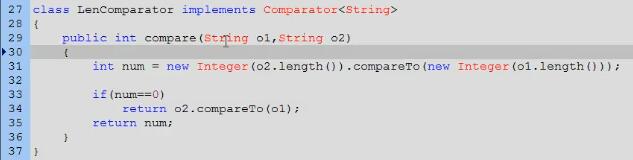


与原先的不同主要在22行和31行

**P189 集合框架（泛型使用）**



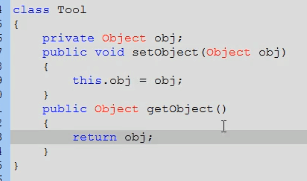
重新定义比较器



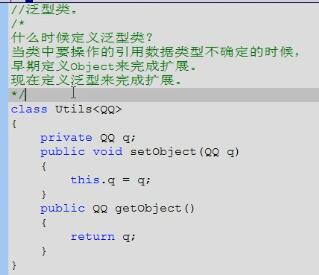
**P190 集合框架（泛型类）**

像ArrayList，TreeSet这些类，它们是如何实现泛型的？

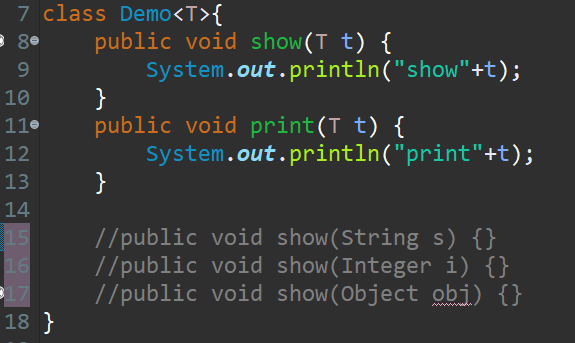
泛型出现以前是这么定义的



泛型出现以后是这么定义的



差别

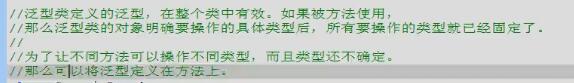


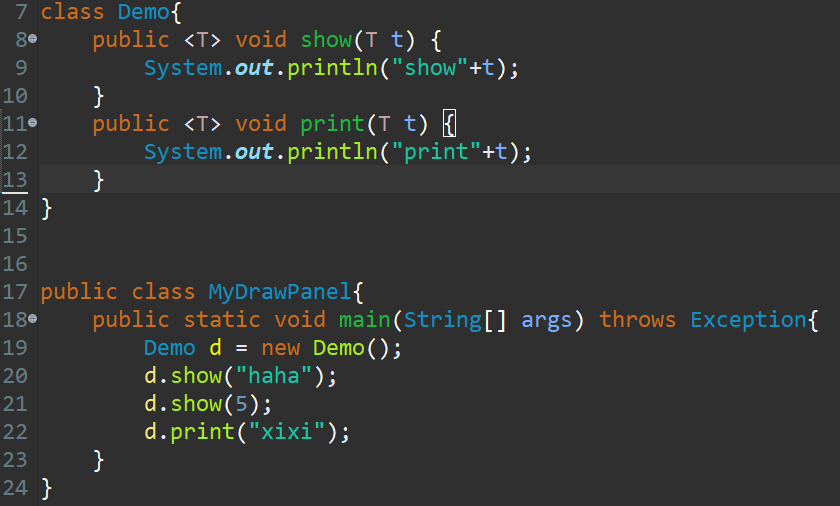
在没有泛型之前，比如要实现第8行的show方法，要么采用15行+16行，要么采用17行+后面强制转换。

泛型的一点局限性：T的引用类型在Demo创建的时候就应该定义好，后面调用方法时就无法传入定义好的引用类型以外的类型。

**P191 集合框架（泛型方法）**

泛型运用在方法上



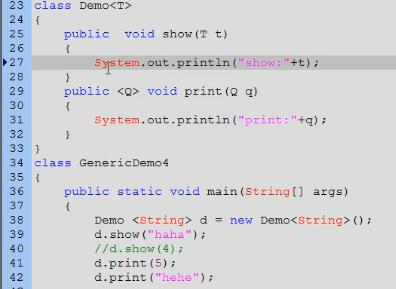


第8行和第11行，虽然都是T，但是不意味着两个方法的传入引用类型是一样的。因为两个T都是局部变量，只在自己的方法体范围内生效。

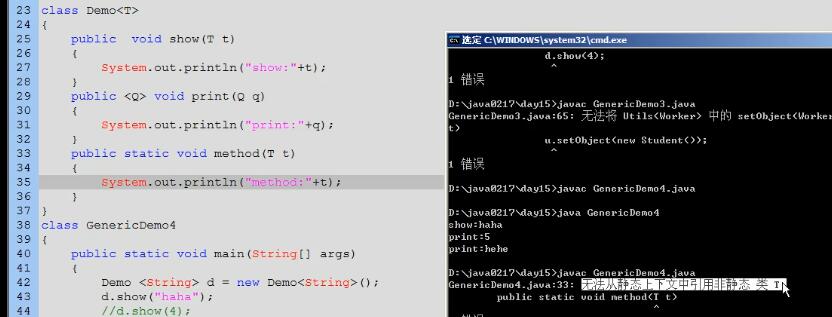
问题：看20行和21行，为什么既能传入String类型，又能传入Integer类型？

同样是因为T是局部变量，在运行完20行后，原本为String的T释放掉了，读到21行，T又变成了Integer。

这么做有什么好处呢？与泛型类搭配使用，达到41行效果。



**P192 集合框架（静态方法泛型）**

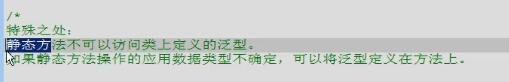


当泛型类中有静态方法时会报错。为什么？

因为静态方法method用到了T，而T要在创建对象后才能明确。举个例子：

Demo<String> d = new Demo<String>();

要在创建对象后才知道T的类型是String

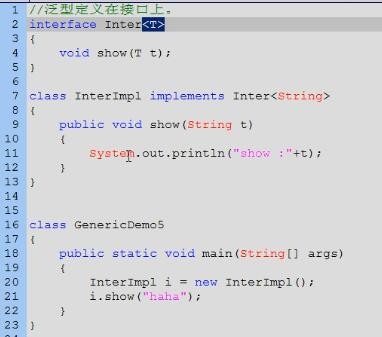


所以静态方法必须要把泛型弄在方法上



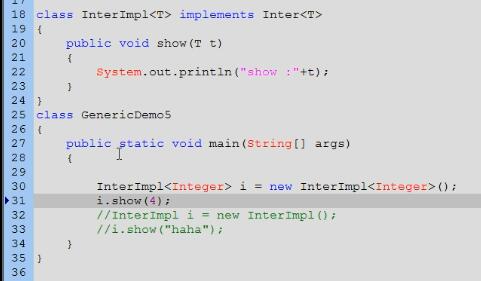
**P193 集合框架（泛型接口）**

知道引用数据类型，这么定义



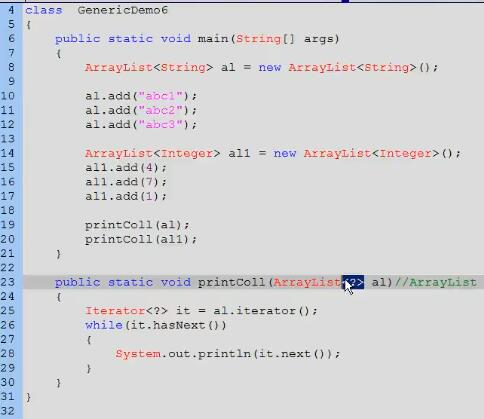
不知道引用数据类型，留给使用的人确定，这么定义

将上面7-23的内容替换成



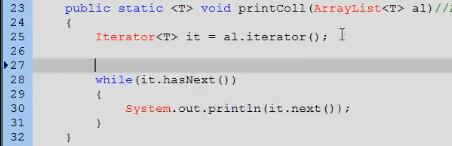
**P194 集合框架（泛型限定）**

需求：我们建立了两个ArrayList，一个存放String引用类型，一个存放Integer引用类型。然后想遍历这两个不同引用类型的ArrayList。

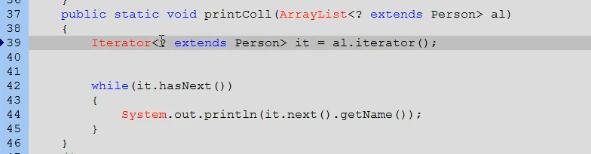


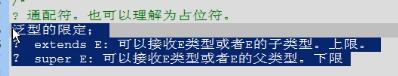
用占位符？有好有坏，好处是提高了拓展性，坏处是不能调用引用数据类型的特有方法。跟多态一样

当然，也可以替换成泛型方法



问题：我既不想printColl方法只接受一个引用数据类型，但是也不想什么类型都接受。比如我只想接受一个自定义的Person类及其子类，即泛型限定，怎么办？只需该23，25两行。





**P195 集合框架（泛型限定2）**

//