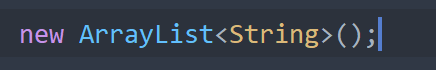
# 泛型概述

**泛型不能是基本数据类型**

**用指定泛型的类型的变量，调用变量的方法，此方法返回值才是指定泛型的。**

**有名的**

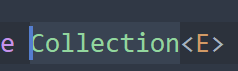
**匿名的**

**这样调用两个对象的方法，凡是方法返回值或者形参加了泛型的都会变成Integer类型。**

**有名的前面指明了泛型以后，后面会自动类型判断泛型与前面的一样，所以可以省略。jdk7**我们new一个集合如ArryList，此时是可以往集合中放任意类型的对象的，因为ArrayList底层使用的是Object[] 本质上就是可以放任意类型的对象。

那么我们可以在newArrylist的时候加一些限制，使得往里面放对象的人只能放指定类型的对象，如限制只能放实现了指定接口的类的对象，或者限定只能放某类的对象及其子类的对象。

这样就不用使得使用代码的人去注意应该只能往里面放什么类型的对象了，而是让写代码的人限制放的类型

举例Collection接口定义了泛型，那么他的所有实现类都定义泛型。

如果我们new ArrayList，指定了泛型的类型new ArrayList<String>

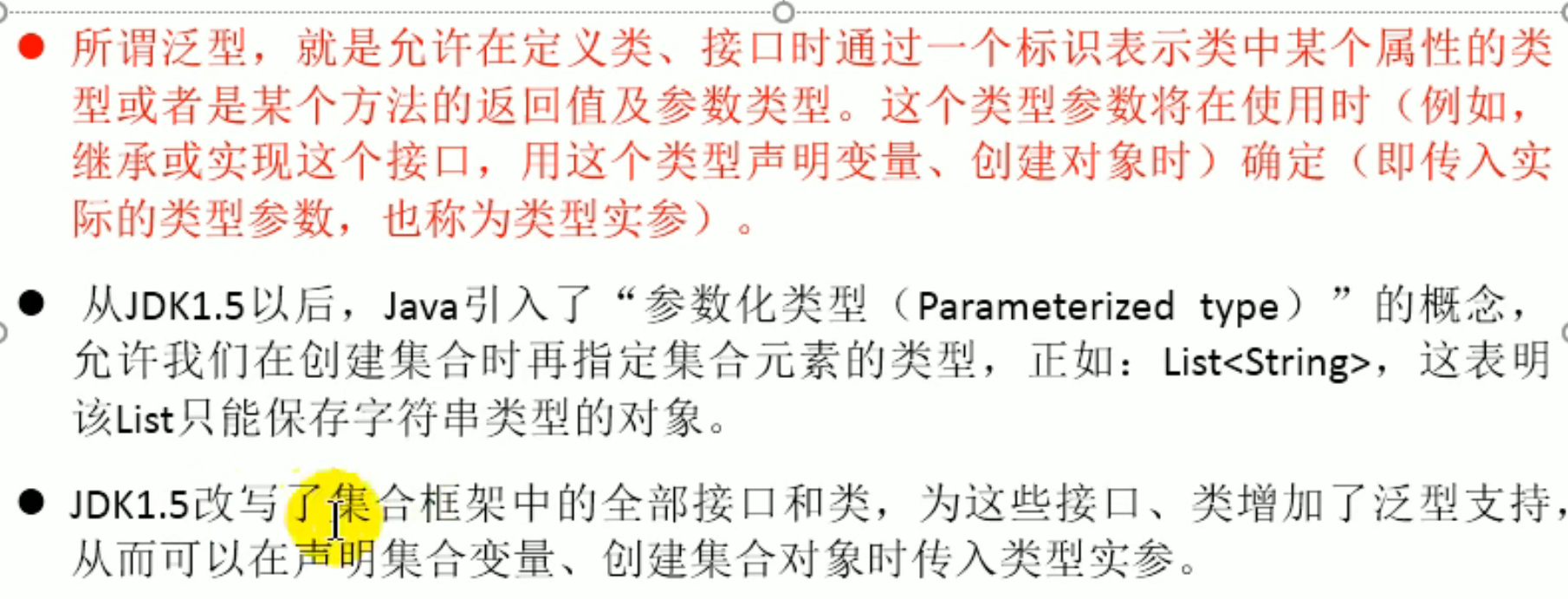


那么调用add方法，就只能添加String类型的对象。

即只要我们new类对象时候定义了泛型类型。

凡是这个类中方法形参用到泛型的，再调用的时候都只能传入指定泛型类型的。

凡是这个类中属性类型用到泛型的，给属性赋值的时候只能赋值为指定泛型类型的。



说白了new对象时只要指定了泛型的类型，后面使用对象的方法时候，传入参数的时候，如果没定义泛型，我们大意了会传入不被需要的形参。

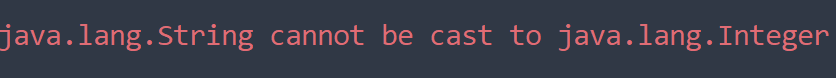
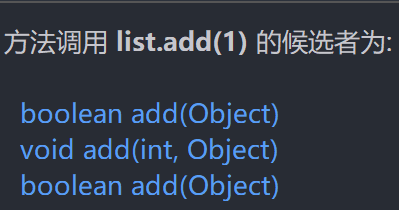
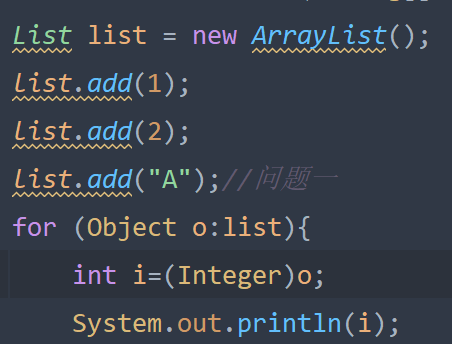
而如果定义了泛型，即使我们传错了形参类型，他也会提示我们出错了。

就是起到一个限制提示作用。

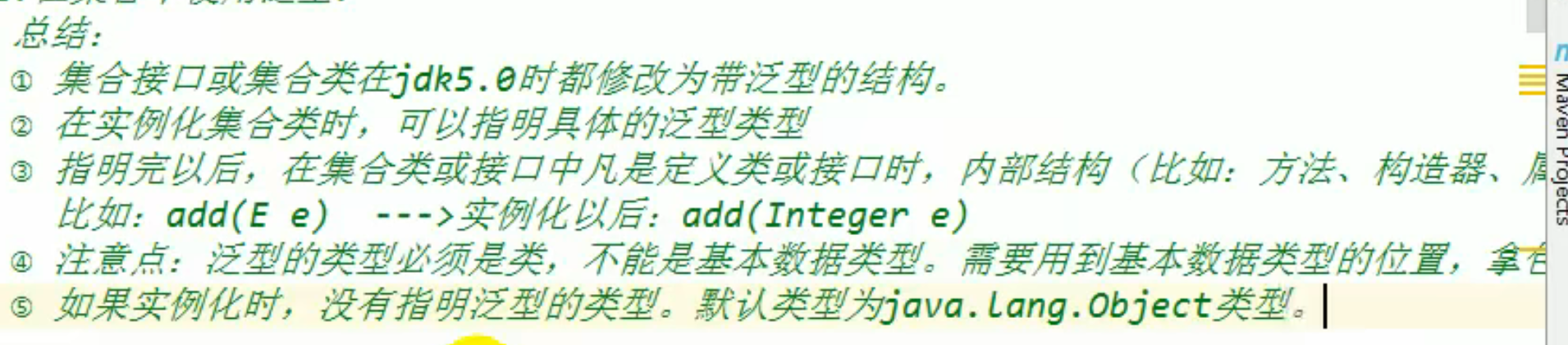
举例：学生成绩添加到集合中。

问题一：本来都是要添加Interger类型的成绩。但是手误添加了一个字符串类型的。

问题二：遍历的时候要将集合中Object强转为Interger，再用int接收成绩值，但是到了字符串就会强转报错。



# 泛型使用举例

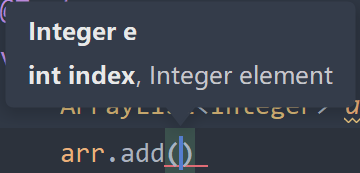


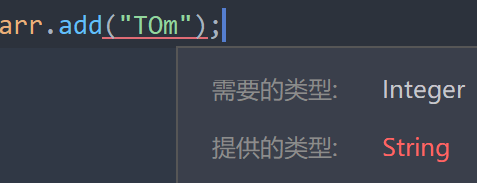
## ArrayList

new 使用了泛型的类的对象的时候，格式如下。

XX<指定泛型类型> xx=new XX <指定泛型类型>（）；



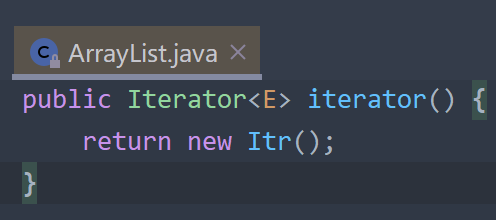
此时调用add就会发现只能传入Integer类型的对象了。



遍历的时候，就直接可以用Integer类型变量接收值，而不是Object类型变量o了。

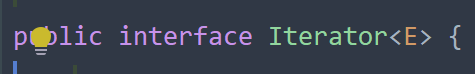
 

返回iterator对象的时候



返回了一个带有泛型的Iterator接口类型的iterator对象。

前提是Iterator接口定义为泛型的。

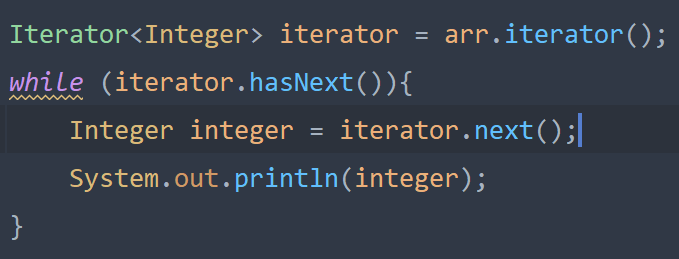




当我们调用方法iterator()返回泛型Iterator接口类型的对象时候，我们需要用Iterator泛型的变量接收。这样使用此变量调用的next方法返回值才会是泛型的。

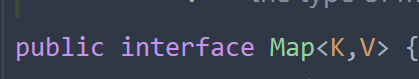
这样连串起来。

1. 因为Iterator接口也定义了泛型。
2. 所以ArrayList创建类型为其泛型的iterator对象。
3. 使用Iterator<指定泛型>的变量调用iterator中的next方法
4. next方法返回值定义为泛型的
5. 返回值为指定泛型的。

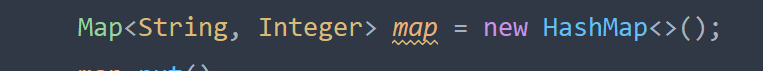
 

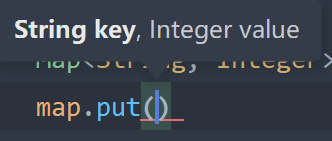
## Map

### 泛型嵌套



我们声明Map变量的时候，可以传入指定两个泛型，字面意思就是一个key的一个value的。





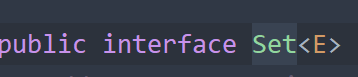
我们想返回entry集合时



因为每个Entry对象都传入了泛型，而Entry构成的set集合也必须使用泛型修饰一下。

这样调用set变量的时候其内部的Entry才是泛型的。

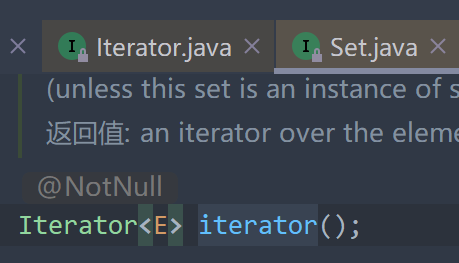
那么怎么修饰呢，如果我们只简单Set<String,Integer>这是不对的。因为Set接口没定义这种两个的泛型。



所以实际上我们得先将E变成Entry，再用<String,Integer>修饰Entry

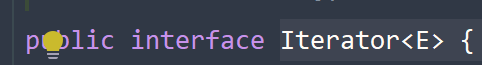
即Set<Entry<String,Integer>> set=map.entry();

此时集合种存储的就是String key，Integer Value的Entry了。



调用set.iterator() 返回的对象也需要用泛型修饰一下类型来接收，因为此时Set的泛型为Entry，所以他方法返回的iterator泛型也是Entry

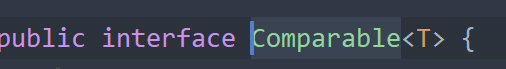
反正实际上iterator此时操作的就是Entry。



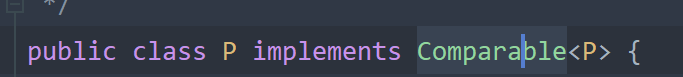
Iterator<Entry<String,Integer>> iterator=set.iterator();

## 自然排序

1. 让要被排序的类实现Compable接口，Comparable接口定义了泛型<>

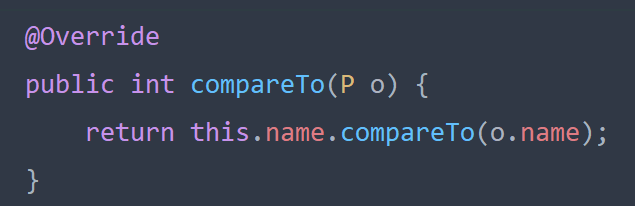


我们不能定义类的泛型，而是去定义接口的泛型，compareTo方法是实现来的，它是在接口中定义的，而不是在我们P类中定义的，我们只是实现，重写。

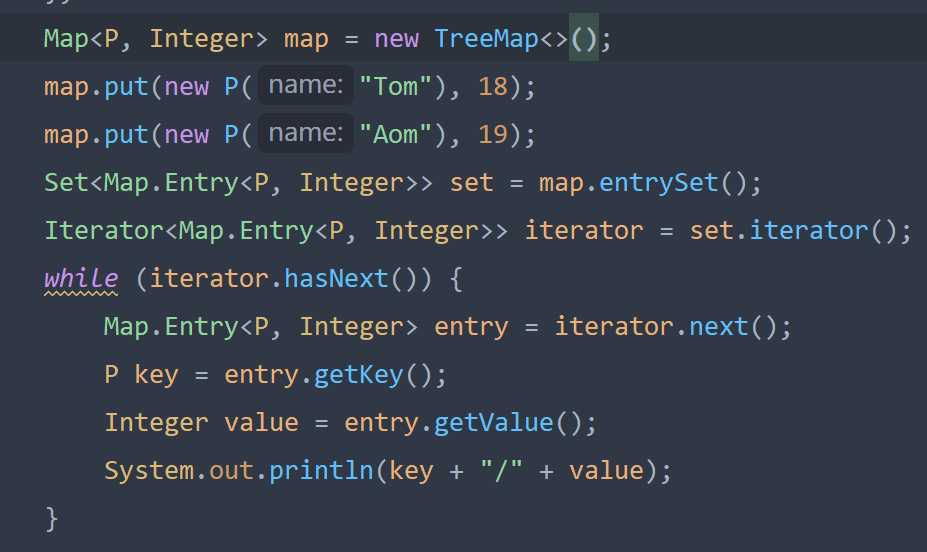
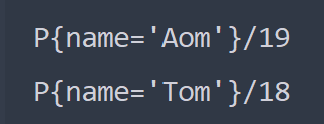


1. 重写compareTo方法此时该方法自动将形参变为接口指定的泛型P

不用我们再手动去判断是不是P类型的了。



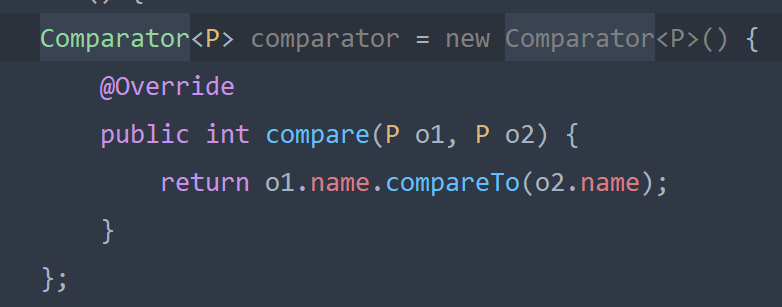
1. new TreeMap开始存储，也可以用泛型

## 定制排序



new Comaprator实现类时指定泛型，重写其compare方法时自动将形参定为指定泛型类型的。同样不需要我们在方法体中再去判断传入的两个形参是不是P类型的了。

# 自定义泛型

jdk为我们提供了很多带有泛型修饰的接口，类等

## 泛型类

### 一定义

如果我们想要在以后给这某个属性赋值的时候，不知道赋值什么类型的值。

可能我们会想到为什么不将属性定义为Object的呢？这样赋什么类型的值不都可以吗？

但是如果此属性出现在此类的很多方法中，比如get方法。

日后我们想赋值某个类型的值了，赋值完以后，调用get，方法返回值时Object的，我们还需要强转为某个类型的。

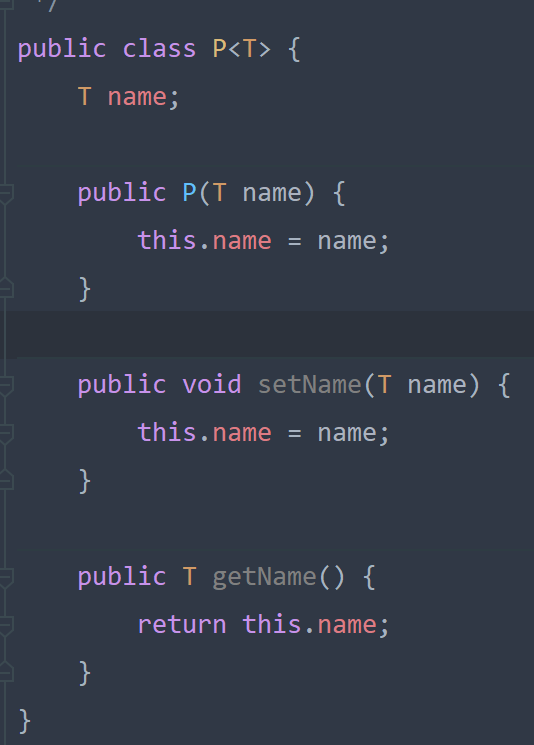
如果我们用了泛型去定义，牵一发而动全身。

一旦给属性赋值了某种类型，get方法返回值自动变为某类型的，不再是Object的了。就不需要我们再去强转

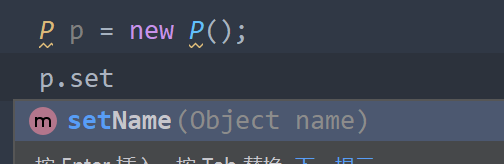
大可将泛型看作是一种修饰，我们声明变量时会这样写：XXX类型的 xxx变量名

这个XXX可以替换为T，就是泛型修饰。

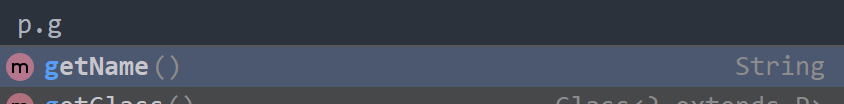
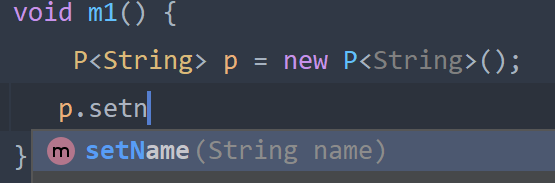
大前提：必须在声明类的时候使用泛型，这样实例化此类对象的时候，填入具体的泛型是啥，然后调用对象的方法属性的时候泛型就会生效。



如果没有使用泛型

默认为Object修饰的name

使用了的话set，get方法返回的就是指定的类型的name



### 二继承性

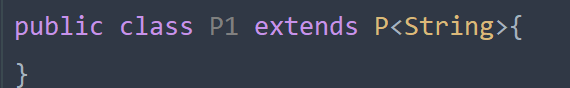
在集合中，接口List是带有泛型的，而ArrayList实现类实现了List接口，在声明ArrayLsit类的时候，开发者也加上了泛型E。

作为子类，如果父类是E，T等泛型的，子类继承父类的时候，可以指明泛型的具体类别。此时子类就不再是泛型类。

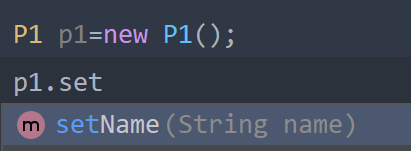
也可以跟父类一样还是用E，T，还可以不加泛型，默认被泛型修饰的属性方法等为Object类型的。

因为ArrayList这种集合类存储的数据不一定是什么类型的，所以开发者也将ArrayList声明为E泛型的。

**子类不再是泛型类：**

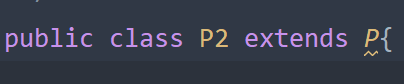


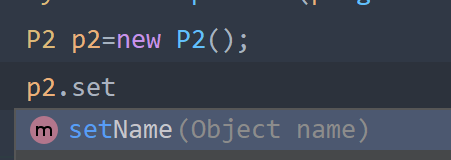
在继承时明确将父类的泛型指明为String类型的，P1中的name属性就是String类型的。



此时因为在继承时指明了类别，P1子类就不再是泛型类了。

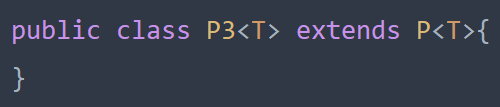
或者继承时什么也不指定，默认为Obejct，此时也不再是泛型。

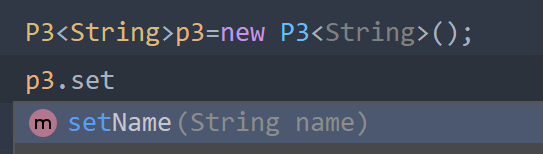




**子类还是泛型：**

不指明父类的泛型类型，就可以将子类声明为泛型的。

****

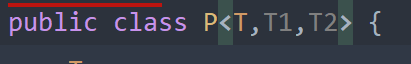
****

或者子类既有自己的泛型，还继承了父类的泛型

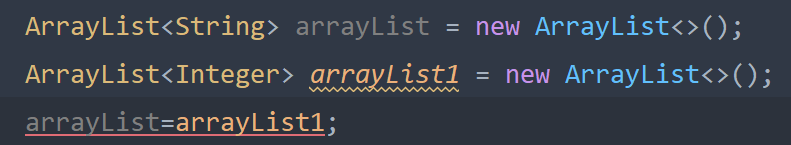


### 三其他注意点

**如果有多个泛型，声明时可以<T1 ，T2， T3> 中间用逗号隔开。**



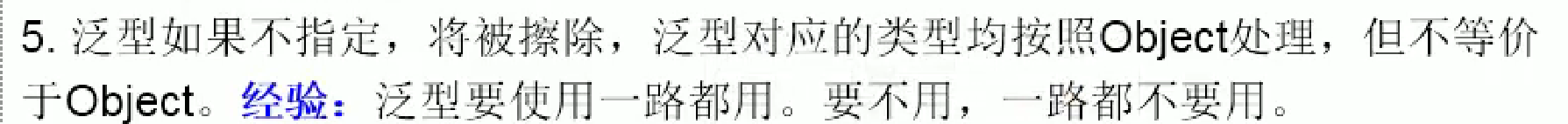
**如果同一个类实例化两个对象，且将泛型指明了不同类型，就不能将一个变量的地址赋值给另一个变量。**

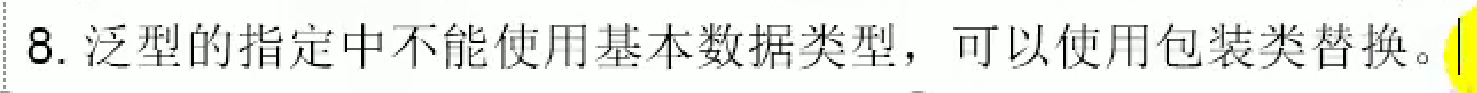


arr指向的是只能放String数组的一块地址。

arr1指向的是只能放Integer数组的一块地址。

如果第三句不报错的话，调用arr然后add一个String类型的对象，就会放到限制类型只能为Integer类型的数组中。就不合理了。





**不能在静态方法或者属性前使用泛型，因为类还没有加载，泛型也没有被加载，静态方法是一开始就加载的，那么他前面的泛型没意义。**

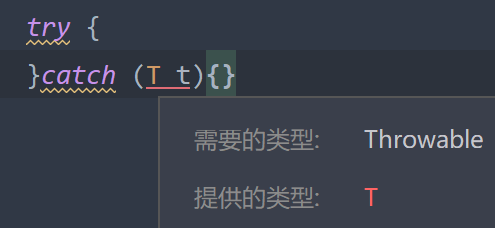
**复习：java中不能定义静态类，只能定义内部类为静态类。**





平时我们自定义异常类时都继承Exception，但是Exception也没定义泛型，所以我们自定的异常类也不能使用泛型。

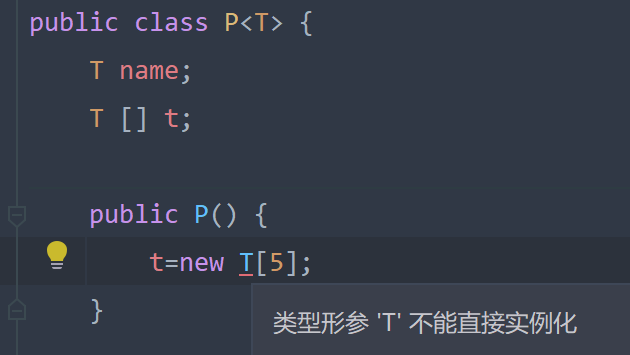




**new数组必须指明类型不能直接定义为泛型**

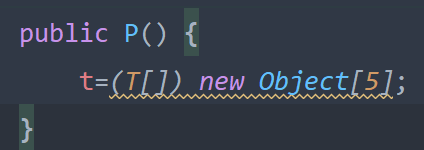
new 泛型类的对象时候，通常省略后面的<xx>，是因为jdk1.8会类型推断，不用我们再填后面的泛型。

如果直接new一个数组，我们想要这个数组是泛型。

这样就会报错。

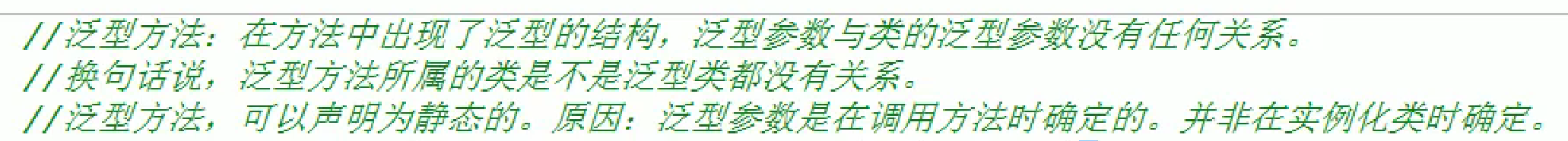
因为在直接实例化数组的时候，必须指明其类型

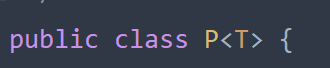
我们可以先声明一个Object类型的数组，然后强转为泛型的。

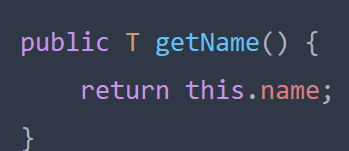


此时实例化P对象指明具体类型的时候，对象的t数组实际上就是指明的类型的数组。

## 泛型方法







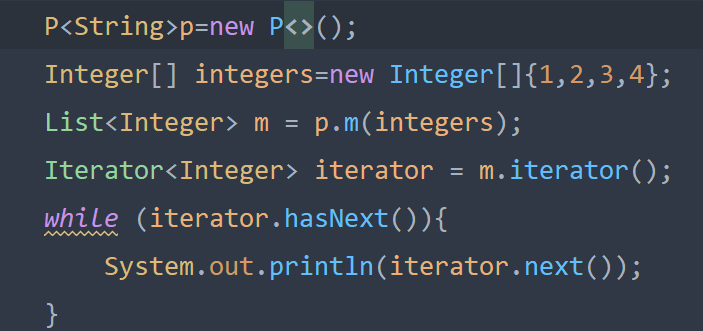
像我们定义的泛型类中的方法**不叫泛型方法**。

泛型方法就必须是不受类的泛型影响的泛型。即类的泛型是T，方法的泛型是E，两者不受影响。

并且方法脱离了此类，在别的类中定义一模一样的泛型方法也不会报错。因为他的泛型是不受类的泛型影响的。

格式如下，必须在前面加泛型，因为不加的话编译器会认为E是某个类。也相当于声明泛型类的时候那样，规定整个方法所使用的泛型。

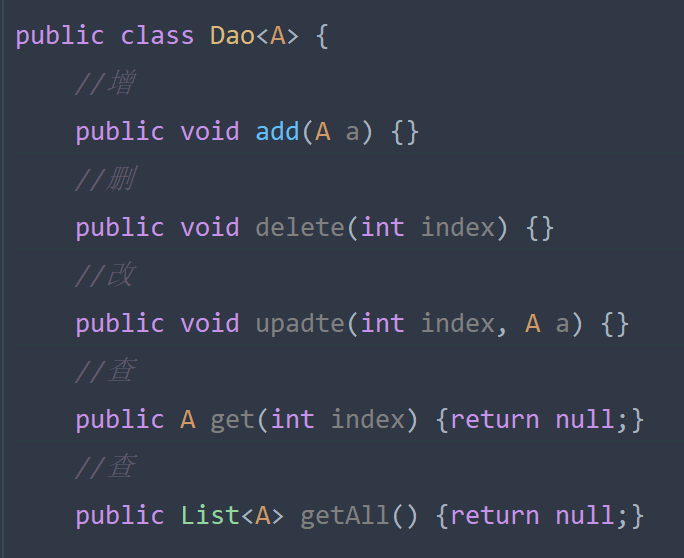


## 泛型应用场景

假设数据库中有Student表和Animals表，那么在java中就有两个类Student和Animals。

我们可以将操作数据库中表的通用操作封装到Dao类中。比如增产查改。但是不写方法体。并且将Dao定义为泛型类。增删查改方法都用泛型修饰一下。



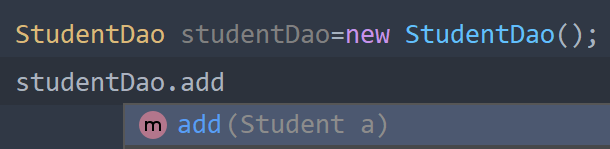
然后定义StudentDao/AnimalsDao子类继承Dao，并指明继承的具体类型。

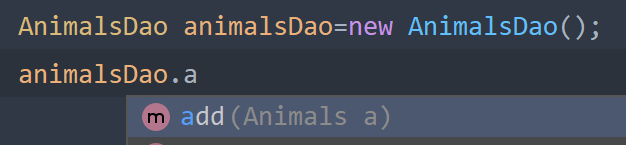




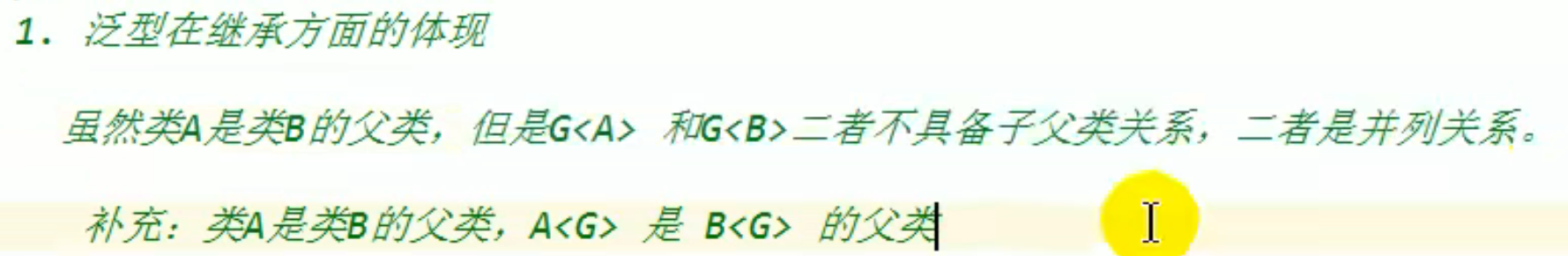
然后重写方法体等。

在测试中，我们实例化子类对象，调用增删查改方法，发现不再是Object了，而是我们指定的类型Student类，Animals类





# 泛型在继承方面的体现



我们知道，如果类A是类B的父类。假设他们的对象是a，b

那么a=b这样赋值地址值是可以的。多态性。

同样A[]a和B[]b

a=b也是可以的。多态性。

但是在集合中

ArrayList<Object> arr1=null;

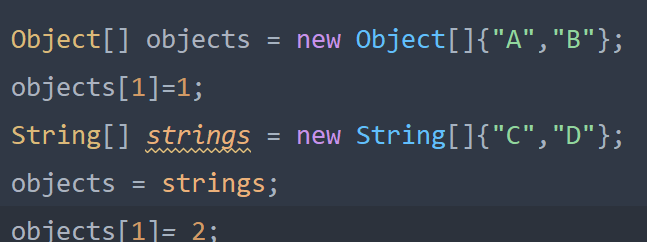
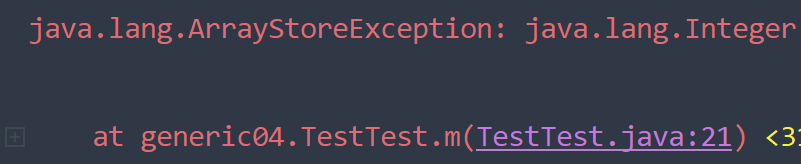
ArrayList<String>arr2=null;

arr1=arr2就是错的。

为什么错误呢，反证法，如果正确了，arr1=arr2 。

现在通过arr1向底层的Object数组中添加了一个Integer类型的对象，因为arr1=arr2，所以实际上是添加到arr2的数组中去了，但是我们在声明的时候明确指出arr2只储存String类型的对象，这就矛盾了。

数组在定义的时候开发者并不认为我们经常直接对数组本身做数据修改。

但是对于泛型来说，他就是规定什么类型的存在，规定了就必须在编译层面考虑到储存数据类型的问题。所以从根源上杜绝数组出现的异常问题（如上）。

如果是

A<Srting> a

B<String> b

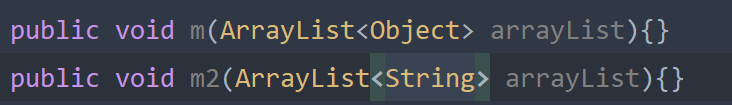
a=b是可以的。这就像



两者的泛型都指定为String，就可以直接按照多态性来赋值地址值了，子类地址给父类引用。

对于集合来讲，底层的数组都是一样的，子集合的地址当然可以赋值给接口的引用。

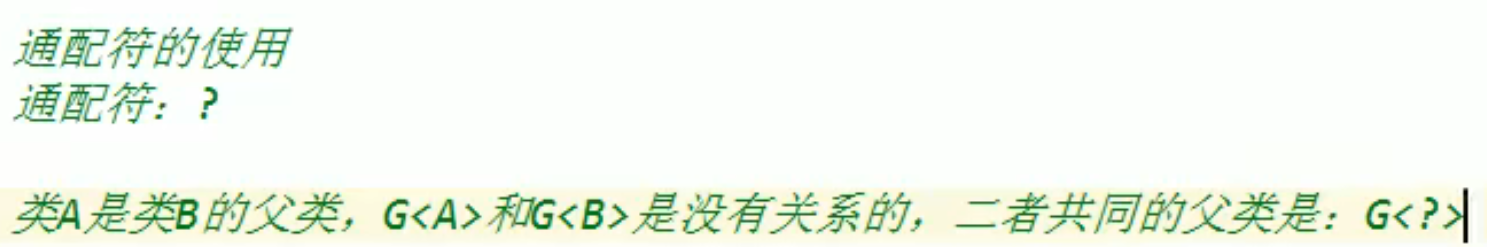
由于ArrayList<A a>和ArrayList<B b>是并列类没有子父类关系。有些方法用到这两种情况时就得分开写了：



或者干脆弄成泛型方法

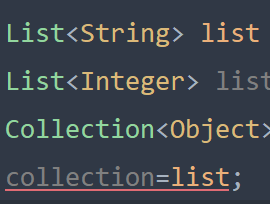


# 通配符的使用



现假设有一个方法，形参需要传入G<A> G<B> G<C>等如List<String> List<Object> List<Integer> 等。由于这三者无子父类关系，所以不能以多态的形式 写进形参。

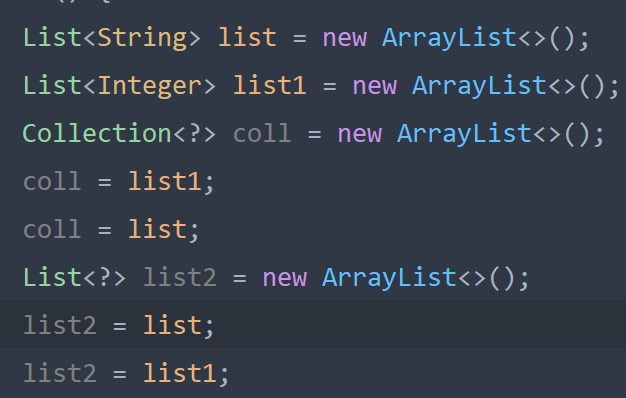
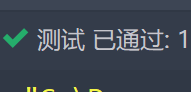
这种方法也是不可以的



那么总不可能说是写好多方法形参一个个的将所有类型都写一遍。

这就产生了通配符 类型+<?>

？类似不确定性的泛型一样。

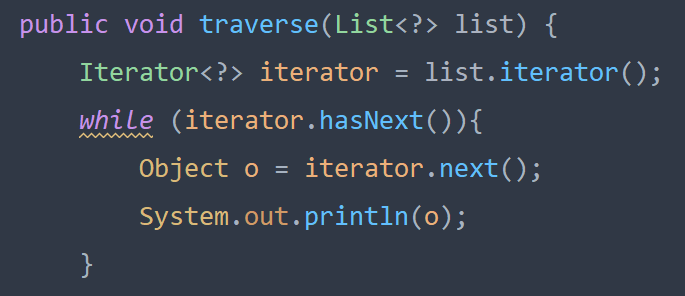
这样就不报错了。

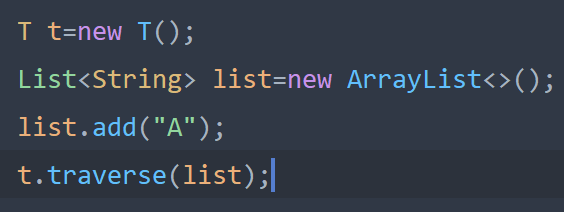
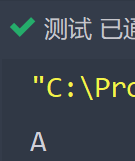
还有一个问题，泛型可以是一种修饰类型如T t。但是没有？ t这种写法。所以

在遍历中我们由于不能使用List<T>的方式作为形参，不可多态。所以以下方式是错误的：



正确的应为通配符方式



## 通配符不可写性

List<?> list=new ArrayList<>();

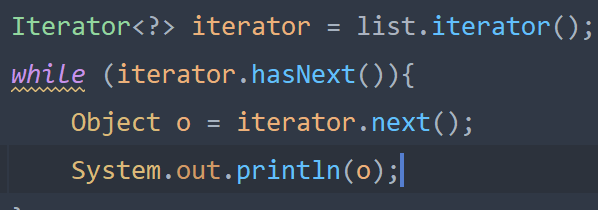
这个list可以作为 List<String>list1 List<Integer>list2等的通用父类使用。即可以list=list1/list2

但是这个list对象不能add东西。报错。反证，如果能add，那么不就可以往list1中add一个任意类型的数据吗，list1都限制了为String类型对象了，就矛盾。

所以list不能执行 涉及到往里面添加数据的操作。

但是像读操作是可以的。也可以生成iterator。

通过迭代器返回的对象为Object类型的。



## 有限制条件的通配符的使用

假设现在有一个方法m，形参为List<? extends Person>

这个就叫做有限制的通配符。

代表只能传入泛型为Person及其子类的List及其子类的对象。

假设现在有一个方法m，形参为List<? super Person>list

super意思是超越。

这个形参表示只能传入Person及Person的父类。

复习一下不能将List（Object o）作为形参的原因：

如果传入的是list<String>对象list1，那么由于在形参中list1的地址给了list。方法中所有的操作实际上都是以形参list来的，而list底层数组编译器认为是Object类型的，所以就可以添加非String类型的数据到list1的地址中。而list1对象底层数组限制为<String>的，所以就矛盾。

现假设方法m形参为List<? extends Person>list 注：Person 的子类是Student

那么list可执行的操作为get操作，iterator操作等。

能不能写入呢？

如果要写入，其写入的类型只能为Person及其子类，但是问题来了，我们并不知道实际传给形参的是Person或是哪个子类。

但是通配符是不确定性的，写方法的时候，list.add()

假设通过形参传进来的实际是List<Person>list1 那么可以add Person及其子类对象。

假设通过形参传进来的实际是List<Student> list1 那么仅可以add Student及其子类对象。

那么通配符在设定的时候必须指明一种通用的情况，因为？extends Person，假设指定为能add Person及其子类对象。

那么传入List<Student>list1时，如果add了一个Person对象，就会出现类型转换异常。因为不能将一个父类对象地址给子类引用。

由于？extends Person的设定，是没有子类下限的。

所以综上所述，<? extends xxx>是不能执行涉及到添加的操作的。

现假设方法m形参为List<? super Person>list 注：Person 的子类是Student

那么list可执行的操作为get操作，iterator操作等。

能不能写入呢？

如果要写入的话，上限是没法确定的。（实际上上限为Object）

下限为Person类，也就是说，能传入Person及其父类。

那么需要设定add（）一种情况。

假设设定为能add Person及其子类对象。

现在传入任意一个List<Person>，其底层数组当然可以add Person及其子类对象。

现在传入任意一个List<Person的父类>，其底层数组当然可以add Person及其子类对象。

所以综上所述，<? super Person>是可以执行添加操作的。但是只能add Person及其子类对象

# 习题

JavaSenior工程-JavaGenrics模块-generic06包