泛型

一、泛型入门

1、为什么需要泛型

在没有泛型之前，一旦把一个对象保存到集合中，集合就会忘记对象的类型，把所有的对象当成Object处理。当程序从集合中取出对象后，需要进行强制类型转换，这种强制类型转换不仅代码臃肿，而且很容易造成ClassCastException异常。

2、泛型的好处

增加泛型支持后的集合，完全可以记住集合中元素的类型，并可以在编译时检查集合中元素的类型，如果试图向集合中添加不满足类型需求的对象，编译器就会提示错误。增加泛型支持后的集合，让代码更简洁，程序更健壮。除此外java泛型还增加了枚举类、反射等方面的功能。

3、泛型的概念

把java的参数化类型称之为泛型，所谓参数化类型即创建集合时允许为之指定集合元素的类型。

二、定义泛型接口、类

1、java集合使用泛型的部分代码示例：

//定义接口时指定一个类型形参，形参名为E

**public** **interface** List<E> **extends** Collection<E> {

//E可以作为返回类型，E可以作为形参类型

Iterator<E> iterator();

**boolean** add(E e);

List<E> subList(**int** fromIndex, **int** toIndex);

}

//定义接口时指定两个类型形参，形参名为K,V

**public** **interface** Map<K,V> {

//K, V可以作为返回类型，K,V可以作为形参类型

Set<K> keySet();

V put(K key, V value);

}

泛型实质：允许在定义接口、类时指定类型形参，类型形参在整个接口、类体内中可以当成类型使用，几乎所有可使用其他普通类型的地方都可以使用这种类型形参。

2、自定义泛型类

我们可以为任何类增加泛型声明，并不是只有集合类才可以使用泛型声明，虽然泛型是集合类的重要使用场所。

当创建带泛型声明的自定义类，为该类定义构造器时，构造器名还是原来的类名，不要增加泛型声明。通过构造器实例化对象的时候可以使用Apple<T>形式。

示例：

**public** **class** Apple<T> {

**private** T info;

**public** Apple(T info){

**this**.info = info;

}

**public** T getInfo(){

**return** **this**.info;

}

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

Apple<String> ap1 = **new** Apple<String>("apple");

Apple<Integer> ap2 = **new** Apple<Integer>(2);

}

}

3、从泛型类派生子类

当创建了带泛型声明的接口、父类之后，可以为该接口创建实现类，或从该父类派生子类，父类或接口的类型参数应特别注意，如下：

//父类不带类型参数

**class** AppleA **extends** Apple{

//……

}

}

//父类带类型形参，子类也要带类型形参，否则编译出错

**class** AppleB<T> **extends** Apple<T>{

**public** AppleB(T info) {

**super**(info);

}

}

//父类带类型实参，即指定具体的类型参数

**class** AppleC **extends** Apple<String>{

**public** AppleC(String info) {

**super**(info);

}

}

4、实际并不存在泛型类

系统并不会为ArrayList<String>生成新的class文件，也不会把ArrayList<String>当成新类来处理。不管泛型类型的实际类型参数是什么，它们在运行时总有同样的类。

类的静态方法、静态初始化块和静态变量的声明和初始化都不允许使用类型参数。

instanceof运算符后不能使用泛型类。

三、类型通配符

1、使用类型通配符

类型通配符是一个问号（？），是为了表示某一个类的各种泛型类的父类，例如：List<String>,List<Integer>,都是类型参数类型已知的，而List的所有泛型类可以用List<?>，List<?>可以代表List<String>,List<Integer>中的任意。

带通配符的List近代表它是各种泛型List的父类，并不能将元素加入其中。

2、使用类型通配符上限

被限制的泛型通配符表示：

//它表示所有sharp泛型List的父类

List<? **extends** sharp>

3、设定类型形参的上限

Java泛型不仅允许在使用通配符形参时设定类型上限，也可以在定义类型形参时设定上限。用于表示传入该类型形参的实际类型必须是该上限类型，或该上限类型的子类。

**class** Apple1<T **extends** Number>{

T col;

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

Apple1<Integer> ai =**new** Apple1<Integer>();

Apple1<Double> ad =**new** Apple1<Double>();

//编译错误

Apple1<String> ac =**new** Apple1<String>();

}

}

四、泛型方法

有这样一种情况，我们在定义类，接口时没有使用泛型，但定义方法时想自己定义类型形参，也是可以的。

1、定义泛型方法

泛型方法用法格式：

修饰符 <T,S> 返回值类型 方法名(形参列表){

//方法体

}

**public** **class** MethodGeneric {

//声明一个泛型方法

**public** **static** <T> **void** fromArrayToCollection(T[] a,Collection<T> c){

**for**(T o:a){

c.add(o);

}

}

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

Object[] oa = **new** Object[20];

Collection<Object> co = **new** ArrayList<Object>();

*fromArrayToCollection*(oa,co);

String[] sa = **new** String[20];

Collection<String> cs = **new** ArrayList<String>();

*fromArrayToCollection*(sa,cs);

*fromArrayToCollection*(sa,co);

Integer[] ia = **new** Integer[20];

Double[] da = **new** Double[20];

Collection<Number> cn = **new** ArrayList<Number>();

*fromArrayToCollection*(ia,cn);

*fromArrayToCollection*(da,cn);

//编译出错，

*fromArrayToCollection*(da,cs);

}

}

2、泛型方法与类型通配符的区别

通配符是用来支持灵活的子类化的。

泛型方法允许类型形参被用来表示方法的一个或多个参数之间的类型依赖关系，或者方法返回值与参数之间的类型依赖关系。如果没有这种依赖关系，不应该使用泛型。

3、设定通配符下限

**public** **static** <T> T copy(Collection<T> dest,Collection<? **extends** T> src){

T last = **null**;

**for**(T el:src){

last = el;

dest.add(el);

}

**return** last;

}

//设定通配符下限

**public** **static**<T> T copy(Collection<? **super** T> dest,Collection<T> src){

T last = **null**;

**for**(T el:src){

last = el;

dest.add(el);

}

**return** last;

}