Glimmertask8

泛型类

什么是泛型？

- 参数化类型（**我的理解是可以存放多种类型数据的自定义的类型**），传入的内容有类型要求，比如不能将int型放入一个存放String类型的容器中（ArrayList < String > ）如果错误地将传入数据类型搞错，程序会报错
- 如果没有规定ArrayList的放入对象的类型，默认会储存Object类的所有内容，但在输出时可以通过造型将所需类型的数据输出
- 相对于Object类，泛型的优势在于可扩展性、安全性、可读性

泛型类？

典型：List、Set、Map等容器。

基本语法：

```
class 类名称 <泛型标识> {
private 泛型标识 /* (成员变量类型) */ 变量名;
.....
}
```

< 类型参数 >	对应可装类型
T	一般的任何类
E	Element或者Exception
K	key
V	Value,通常与K一起配合使用
S	Subtype

注：

1. 泛型类中静态方法和变量不能用泛型类声明的类型参数，但可以使用泛型方法
2. 泛型类不只接受一个类型参数，它还可以接受多个类型参数

泛型接口？

和泛型类相类似 注：

1. 和泛型类相类似，在泛型接口中，静态成员也不能使用新定义的类型参数（同时要注意的是，接口中的属性默认为静态）
2. 同理，泛型接口可在方法中使用类型参数

3. 如果一个类要实现泛型接口，要确定泛型接口的类型参数，如果没有确定，默认设置为Object类

泛型方法?

基本语法

```
public <类型参数> 返回类型 方法名（类型参数 变量名） {
    ...
}
```

注：

1. 只有在方法签名中声明了< T >的方法才是泛型方法，仅使用了泛型类定义的类型参数的方法并不是泛型方法。 例：

   ```
   public <T> T testMethod(T t){
       return t;
   }
   ```

2. 泛型方法中可以同时声明多个类型参数

3. 泛型类中定义的类型参数和泛型方法中定义的类型参数是相互独立的，它们一点关系都没有。

类型判断

```
public class Test {

    public static <T> T add(T x,T y) {
        return x,y;
    }

    public static void main(String[] args) {
        int i = Test.add(1,2);

        Number f = Test.add(1,1.2);

        Object o = Test.add(1,"asd");

        int a = Test.<Integer>add(1,2);

        Number c = Test.<Number>add(1,2.2);
    }
}
```

< T >由传入的实参决定，如果传入多个不同类型的数，取传入实参的最小级别的父类
或者可以通过"Test.<参数类型>add(传入实参);"语句来控制传入的对象类型，传入不符合的参数会导致编译报错

类型擦除

我自己的理解是：一个泛型，比如ArrayList< Integer >,它的泛型类型在编译时会被擦除，变成了ArrayList< Object > 集合 但是被擦除不代表可以放入例如String类的对象，它还是只能放入Integer类的对象
注：

1. 擦除 ArrayList< Integer > 的泛型信息后，get() 方法的返回值将返回 Object 类型，但编译器会自动插入 Integer 的强制类型转换
2. **泛型信息（包括泛型类、接口、方法）只在代码编译阶段存在，在代码成功编译后，其内的所有泛型信息都会被擦除，并且类型参数 T 会被统一替换为其原始类型**

泛型通配符

为什么会用到泛型通配符？
ArrayList< Integer > 和 ArrayList< Number > 看似是子类和父类的关系，但事实并非如此
而引入上界通配符的概念后，我们便可以在逻辑上将 ArrayList<? extends Number> 看做是 ArrayList< Integer > 的父类，但实质上它们之间没有继承关系

三种形式的泛型通配符

Name	Description
<? extends T>	T 代表了类型参数的上界，该泛型可取到上界以下的所有子类型
<? super T>	T 代表了类型参数的下界，该泛型可取到下界以上的所有子类型
<?>	代表了任何一种数据类型

< ? extends T >：

1. ArrayList<? extends Number> 可以代表 ArrayList< Integer >、ArrayList< Float >、... 、ArrayList< Number >中的某一个集合
2. 但我们不能指定 ArrayList<? extends Number> 的数据类型

注意要点：

- 在 ArrayList<? extends Number> 集合中，不能添加任何数据类型的对象
- 一句话总结：使用 extends 通配符表示可以读，不能写。

<? super T>：
其实就和extends相类似。

1. ArrayList<? super Integer> 在逻辑上表示为 Integer 类以及 Integer 类的所有父类，它可以代表 ArrayList< Integer>、ArrayList< Number >、 ArrayList< Object >中的某一个集
2. 但实质上它们之间没有继承关系
3. ArrayList<? super Integer> 只能表示指定类型参数范围中的某一个集合，但我们不能指定 ArrayList<? super Integer> 的数据类型

- 一句话总结：使用 super 通配符表示可以写，不能读。

< ? >:

- ArrayList 集合的数据类型是不确定的，因此我们只能往集合中添加 null；而我们从 ArrayList 集合中取出的元素，也只能赋值给 Object 对象，不然会产生ClassCastException 异常
- 大多数情况下，可以用类型参数 < T > 代替 <?> 通配符。

PECS 原则(什么时候用extends？什么时候用super？)

如果需要返回 T，则它是生产者（Producer），要使用 extends 通配符；如果需要写入 T，则它是消费者（Consumer），要使用 super 通配符。

- 允许调用读方法T get()获取 T 的引用，但不允许调用写方法 set(T)传入 T 的引用（传入 null 除外）
- 允许调用写方法set(T)传入 T 的引用，但不允许调用读方法 T get()获取 T 的引用（获取 Object 除外）

思考：我自己对于PECS原则的理解是这样的：

1. 当你想输出一个数的时候，你得有对应的想要输出的数据类型才行，所以要框定一个含许多子类的范围，向下取子类
2. 而相对的 ,当你想输入一个数时，你得有一个较大的父类才能满足数据类型，把数读进去，所以要向上取父类