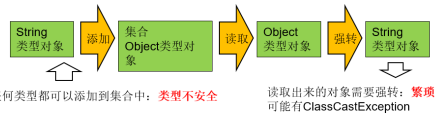
1、泛型定义

所谓泛型，就是允许在定义类、接口时通过一个**标识**表示类中***某个属性***的**类型**或者是***某个方法***的**返回值或参数的类型**。这个类型参数将在**使用时**（例如，**继承或实现这个接口、创建对象**或**调用方法**时）**确定**（即传入实际的类型参数，也称为类型实参）。

2、在集合中不使用泛型可能存在的问题

1）**类型不安全**：因为 add()的参数是 Object 类型，意味着任何类型的对象都可以添加成功。

2）**读取出来的对象**需要使用**强转操作**；繁琐，还可能**导致异常 ClassCastException**。



【使用泛型后】Java 泛型可以保证如果程序在编译时没有发出警告，运行时就不会产生 ClassCastException 异常。即，把不安全的因素**在编译期间就排除**了，而不是运行期；既然通过了编译，那么**类型一定是符合要求的**，就避免了类型转换。同时，代码更加简洁、健壮。把**一个集合中的内容*限制为*一个特定的数据类型**，这就是 **generic 背后的核心思想**。

```
//举例：将学生成绩保存在 ArrayList 中
//标准写法：
//ArrayList<Integer> list = new ArrayList<Integer>();
//jdk7 的新特性：类型推断
ArrayList<Integer> list = new ArrayList<>();
list.add(56); //自动装箱
list.add(76);
//当添加非 Integer 类型数据时，编译不通过
//list.add("Tom");//编译报错
Iterator<Integer> iterator = list.iterator();
while(iterator.hasNext()){
//不需要强转，直接获取添加时元素数据类型
    Integer score = iterator.next();
    System.out.println(score);}
```

3、比较器中使用泛型

```
1>public class Employee implements Comparable<Employee>{
    public int compareTo(Employee o) {…}}
2>Comparator<Employee> comparator = new Comparator<Employee>(){
public int compare(Employee o1, Employee o2) {…}}
3）Iterator<Employee> iterator = set.iterator();
```

4、自定义泛型类/接口

1）在哪里可以声明类型变量<T>

1>声明类或接口时，在**类名**或**接口名后面**声明泛型类型，我们把这样的类或接口称为**泛型类或泛型接口**。

【修饰符】 class **类名<类型变量列表>** 【extends 父类】【implements 接口们】{…}

【修饰符】 interface **接口名<类型变量列表>** 【implements 接口们】{…}

2>声明方法时，在**【修饰符】**与**返回值类型之间**声明类型变量，我们把声明了类型变量的方法，称为**泛型方法**。

[修饰符] **<类型变量列表>** 返回值类型 方法名([形参列表])[throws 异常列表]{...}

2）说明

① 我们在声明完自定义泛型类以后，可以在**类的内部**（比如：属性、方法、构造器中）**使用类的泛型**。

② 我们在**创建**自定义泛型类的**对象**时，可以**指明泛型参数类型**。一旦指明，内部凡是使用类的泛型参数的位置，都**具体化**为指定的类的泛型类型。

③ 如果在**创建**自定义泛型类的**对象**时，***没有指明泛型参数类型***，那么泛型将被擦除，泛型对应的类型均**按照 Object 处理**，但**不等价于 Object**。

【经验】泛型要使用一路都用。要不用，一路都不要用。

④ 泛型的指定中**必须**使用**引用数据类型**。不能使用基本数据类型，此时只能**使用包装类替换**。

⑤ 除创建泛型类对象外，**子类继承**泛型类时、**实现类**实现泛型接口时，也可以**确定泛型结构中的泛型参数**。

【注意】如果我们在给泛型类提供子类时，**子类也不确定泛型的类型**，则可以继续使用泛型参数。

我们**还可以**在现有的父类的泛型参数的基础上，**新增泛型参数**。

```
class Father<T1, T2> {…}
// 类不保留父类的泛型
// 1>没有类型 擦除
class Son1 extends Father {// 看成 class Son extends Father<Object,Object>{…}
class Son<A, B> extends Father{//看成 class Son<A, B> extends Father<Object,Object>{…}
// 2>具体类型
class Son2 extends Father<Integer, String> {…}
class Son2<A, B> extends Father<Integer, String> {…}
// 子类保留父类的泛型
// 1>全部保留
class Son3<T1, T2> extends Father<T1, T2> {…}
class Son3<T1, T2, A, B> extends Father<T1, T2> {…}
// 2>部分保留
class Son4<T2> extends Father<Integer, T2> {..}
class Son4<T2, A, B> extends Father<Integer, T2> {…}
```

3)【注意】

① 泛型类可能有多个参数，此时应将**多个参数一起放在尖括号内**。比如：<E1,E2,E3>

② JDK7.0 开始，泛型的简化操作：ArrayList flist = new ArrayList<>();**//类型推断**

③ 如果泛型结构是**一个接口或抽象类**，则不可创建泛型类的对象。

④ **不能使用 new E[]**去 new 一个 E 类型的数组。但是**可以使用强转的方式**写：E[] elements = (E[]) new Object[capacity];

参考：ArrayList 源码中声明：Object[] elementData，而非泛型参数类型数组。

⑤ 在类/接口上声明的泛型，在本类或本接口中即代表某种类型，但**不可以在静态方法中使用类的泛型**。//public static void show(T t) {...}

⑥ **异常类不能是带泛型**的。（加了就错）

5、自定义泛型方法

1）泛型方法的格式：

[访问权限] <泛型> 返回值类型 方法名([泛型标识 参数名称]) [抛出的异常]{…}

【注意】通常在**形参列表**或**返回值类型**的位置会出现泛型参数 T。

例：**Public <T> T method(T t){…}**

2）说明

1>泛型化方法，一定要添加泛型参数<T>；

2>**与其所在的类**是否是泛型类**没有关系**；

3>泛型方法中泛型参数**在方法被调用时确定**；

4>泛型方法根据需要**可以声明为 static** 的。

6、泛型在继承上的体现

```
Object obj = null;
String str = "AA";
obj = str;  //基于继承性的多态的使用
Object[] arr = null;
String[] arr1 = null;
arr = arr1;//基于继承性的多态的使用
```

【总结】子父类为泛型类型的同一类，无关系；**子父类有同一泛型类型**时，有关系。

1）类 SuperA 是类 A 的父类，则 G<SuperA>与 G<A>的关系：G<SuperA>和 G<A>是并列的两个类，没有任何子父类的关系。

```
ArrayList<Object> list1 = null;
ArrayList<String> list2 = new ArrayList<>();
//list1 = list2;错误的
/* 反证法：假设 list1 = list2 是可以的。
* list2.add("AA");  list1.add(123);
* String str = list2.get(1); //相当于取出的 123 赋值给了 str，错误的。* */
```

2）类 SuperA 是类 A 的父类或接口，SuperA<G>与 A<G>的关系：SuperA<G>与 A<G>有继承或实现的关系。即 A<G>的实例**可以赋值**给 SuperA<G>类型的引用（或变量）。

```
List<String> list1 = null;
ArrayList<String> list2 = new ArrayList<>();
list1 = list2;
list1.add("AA");
```

7、通配符？

1）**G<?>**可以看做是 **G<A>类型的父类**，即可以将 G<A>的对象**赋值**给 G<?>类型的引用（或变量）。

```
List<?> list = null;
List<Object> list1 = null;
List<String> list2 = null;
list = list1;
list = list2;
```

2)读写数据的特点(以集合 ArrayList<?>为例)

1> **读取**数据：**允许的**，读取的值的类型为**Object 类型**；

2> **写入**数据：**不允许的**；**特例：**写入 null 值（因为**?肯定是一个引用类型**，故都存在 null）。

```
List<?> list = null;
List<String> list1 = new ArrayList<>();
list1.add("AA");
list = list1;
//读取数据（以集合为例说明）
Object obj = list.get(0);
System.out.println(obj);
//写入数据（以集合为例说明）
//写入数据，操作失败（类型不确定，写不了）。
//list.add("BB");
//特例：可以将 null 写入集合中。
list.add(null);
```

8、有限制条件的通配符的使用

1）List<? extends A>：可以将 List<A>或 List<B>赋值给 List<? extends A>。其中 B 类是 A 类的子类。

2）List <? super A>：可以将 List<A>或 List<B>赋值给 List<? super A>。其中 B 类是 A 类的父类。

3）读写数据的特点

1>针对于? extends A 的读写

```
List<? extends Father> list = null;
List<Father> list1 = new ArrayList<>();
list1.add(new Father());
list = list1;
//读取数据：可以的
Father father = list.get(0);
//写入数据：不可以的。例外：null
list.add(null);
//list.add(new Father());
//list.add(new Son());
```

2>针对于? super A 的读写

```
List<? super Father> list = null;
List<Father> list1 = new ArrayList<>();
list1.add(new Father());
list = list1;
//读取数据：可以的
Object obj = list.get(0);
//写入数据：可以将 Father 及其子类的对象添加进来
list.add(null);
//list.add(new Object());
list.add(new Father());
list.add(new Son());
```