泛　型

1. 介绍

下面是那种典型用法：

```
List myIntList = new ArrayList();// 1

myIntList.add(new Integer(0));// 2

Integer x = (Integer) myIntList.iterator().next();// 3
```

第 3 行的类型转换有些烦人。通常情况下，程序员知道一个特定的 list 里边放的是什么类型的数据。但是，这个类型转换是必须的(essential)。**编译器只能保证 iterator 返回的是 Object 类型。为了保证对 Integer 类型变量赋值的类型安全，必须进行类型转换**。

当然，这个类型转换不仅仅带来了混乱，它还可能产生一个运行时错误(run time error)，因为程序员可能会犯错。

程序员如何才能明确表示他们的意图，**把一个 list(集合) 中的内容限制为一个特定的数据类型**呢？这就是 generics 背后的核心思想。这是上面程序片断的一个泛型版本:

```
List<Integer> myIntList = new ArrayList<Integer>(); // 1

myIntList.add(new Integer(0)); // 2
```

```
Integer x = myIntList.iterator().next(); // 3
```

注意变量 myIntList 的类型声明。它指定这不是一个任意的 List，而是一个 Integer 的 List，写作：List<Integer>。我们说 **List 是一个带一个类型参数的泛型接口**(a generic interface that takes a type parameter)，本例中，类型参数是 Integer。我们在创建这个 List 对象的时候也指定了一个类型参数。

另一个需要注意的是第 3 行没了类型转换。

现在，你可能认为我们已经成功地去掉了程序里的混乱。我们用第 1 行的类型参数取代了第 3 行的类型转换。然而，这里还有个很大的不同。编译器现在能够在编译时检查程序的正确性。当我们说 myIntList 被声明为 List<Integer>类型，这告诉我们无论何时何地使用 myIntList 变量，编译器保证其中的元素的正确的类型。

实际结果是，这可以增加可读性和稳定性(robustness)，尤其在大型的程序中。

2. 定义简单的泛型

下面是从 java.util 包中的 List 接口和 Iterator 接口的定义中摘录的片断：

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
```

```
}

public interface Iterator<E> {

    E next();

    boolean hasNext();

}
```

这些都应该是很熟悉的，除了尖括号中的部分，那是接口 List 和 Iterator 中的形式类型参数的声明(the declarations of the formal type parameters of the interfaces List and Iterator)。

类型参数在整个类的声明中可用 ,几乎是所有可以使用其他普通类型的地方

在介绍那一节我们看到了对泛型类型声明 List (the generic type declaration List) 的调用，如 List<Integer>。在这个调用中(通常称作一个参数化类型 a parameterized type) , 所有出现的形式类型参数(formal type parameter,这里是 E)都被替换成实体类型参数(actual type argument)(这里是 Integer)。

你可能想象,**List<Integer>代表一个 E 被全部替换成 Integer 的版本**：

```
public interface IntegerList {

    void add(Integer x)
```

```
    Iterator<Integer> iterator();

}
```

类型参数就跟在方法或构造函数中普通的参数一样。就像一个方法有形式参数(formal value parameters)来描述它操作的参数的种类一样，一个泛型声明也有形式类型参数(formal type parameters)。当一个方法被调用，实参(actual arguments)替换形参 ,方法体被执行。当一个泛型声明被调用，实际类型参数(actual type arguments)取代形式类型参数。

一个命名的**习惯：推荐用简练的名字作为形式类型参数的名字(如果可能，单个字符)。最好避免小写字母**

3. 泛型和子类继承

让我们测试一下我们对泛型的理解。下面的代码片断合法么？

```
List<String> ls = new ArrayList<String>(); //1

List<Object> lo = ls; //2
```

第 1 行当然合法，但是这个问题的狡猾之处在于第 2 行。

这产生一个问题：

一个 String 的 List 是一个 Object 的 List 么？大多数人的直觉是回答：“当然！”。

好，在看下面的几行：

```
lo.add(new Object()); // 3

String s = ls.get(0); // 4: 试图把 Object 赋值给 String
```

这里，我们使用 lo 指向 ls。我们通过 lo 来访问 ls,一个 String 的 list。我们可以插入任意对象进去。结果是 ls 中保存的不再是 String。当我们试图从中取出元素的时候，会得到意外的结果。

java 编译器当然会阻止这种情况的发生。第 2 行会导致一个编译错误。

总之，**如果 Foo 是 Bar 的一个子类型(子类或者子接口)，而 G 是某种泛型声明，那么 G<Foo>是 G<Bar>的子类型并不成立!!**

为了处理这种情况，考虑一些更灵活的泛型类型很有用。到现在为止我们看到的规则限制比较大。

4. 通配符(Wildcards)

考虑写一个例程来打印一个集合(Collection)中的所有元素。下面是在老的语言中你可能写的代码：

```
void printCollection(Collection c) {
    Iterator i = c.iterator();
    for (int k = 0; k < c.size(); k++) {
```

```
            System.out.println(i.next());

        }

    }
```

下面是一个使用泛型的幼稚的尝试(使用了新的循环语法):

```
    void printCollection(Collection<Object> c) {

        for (Object e : c) {

            System.out.println(e);

        }

    }
```

问题是新版本的用处比老版本小多了。**老版本的代码可以使用任何类型的 Collection 作为参数，而新版本则只能使用 Collection<Object>**，我们刚才阐述了，它不是所有类型的 collections 的父类。

那么什么是各种 collections 的父类呢？它写作： **Collection<?>**(发音为:"collection of unknown")，就是，一个集合，**它的元素类型可以匹配任何类型**。显然，**它被称为通配符**。我们可以写：

```
    void printCollection(Collection<?> c) {

        for (Object e : c) {
```

```
            System.out.println(e);
        }
    }
```

现在，我们可以使用任何类型的 collection 来调用它。注意，我们**仍然可以读取 c 中的元素，其类型是 Object**。这永远是安全的，因为不管 collection 的真实类型是什么，它包含的都是 Object。

但是**将任意元素加入到其中不是类型安全的**：

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // 编译时错误
```

因为我们不知道 c 的元素类型，我们不能向其中添加对象。

add 方法有类型参数 E 作为集合的元素类型。我们传给 add 的任何参数都必须是一个未知类型的子类。因为我们不知道那是什么类型，所以我们无法传任何东西进去。唯一的例外是 null，它是所有类型的成员。

另一方面，我们可以调用 get()方法并使用其返回值。返回值是一个未知的类型，但是我们知道，它总是一个 Object

4.1. 有限制的通配符(Bounded Wildcards)

考虑一个简单的画图程序，它可以用来画各种形状，比如矩形和圆形。

为了在程序中表示这些形状，你可以定义下面的类继承结构：

```
public abstract class Shape {
    public abstract void draw(Canvas c);
}
public class Circle extends Shape {
    private int x, y, radius;
    public void draw(Canvas c) { // ...
    }
}
public class Rectangle extends Shape {
    private int x, y, width, height;
    public void draw(Canvas c) {
      // ...
    }
}
```

这些类可以在一个画布(Canvas)上被画出来:

```
public class Canvas {
    public void draw(Shape s) {
        s.draw(this);
    }
}
```

所有的图形通常都有很多个形状。假定它们用一个 list 来表示，Canvas 里有一个方法来画出所有的形状会比较方便：

```
public void drawAll(List<Shape> shapes) {
    for (Shape s : shapes) {
        s.draw(this);
    }
}
```

现在，类型规则导致 drawAll()只能使用 Shape 的 list 来调用。它不能，比如说对 List<Circle>来调用。这很不幸，因为这个方法所作的只是从这个 list 读取 shape，因此它应该也能对 List<Circle>调用。我们**真正要的是这个方法能够接受一个任意种类的 Shape**:

```
public void drawAll(List<? extends Shape> shapes) { //..}
```

这里有一处很小但是很重要的不同:我们把类型 List<Shape> 替换成了 List<? extends Shape>。现在 drawAll()可以接受任何 Shape 的子类的 List，所以我们可以对 List<Circle>进行调用。

List<? extends Shape>是有限制通配符的一个例子。这里？代表一个未知的类型，就像我们前面看到的通配符一样。但是，在这里，我们知道这个未知的类型实际上是 Shape 的一个子类。我们说 Shape 是这个通配符的上限(upper bound)。

像平常一样，要得到使用通配符的灵活性有些代价。这个代价是，现在向 shapes 中写入是非法的。比如下面的代码是不允许的：

```
public void addRectangle(List<? extends Shape> shapes) {
    shapes.add(0, new Rectangle()); // compile-time error!
}
```

你应该能够指出为什么上面的代码是不允许的。因为 shapes.add 的第二个参数类型是 ？ extends Shape ——一个 Shape 未知的子类。因此我们不知道这个类型是什么，我们不知道它是不是 Rectangle 的父类；它可能是也可能不是一个父类，所以这里传递一个 Rectangle 不安全。

5. 泛型方法

考虑写一个方法，它用一个 Object 的数组和一个 collection 作为参数，完成把数组中所有 object 放入 collection 中的功能。

下面是第一次尝试：

```
static void fromArrayToCollection(Object[] a, Collection<?> c) {
    for (Object o : a) {
        c.add(o); // 编译期错误
    }
}
```

现在，你应该能够学会避免初学者试图使用 Collection<Object>作为集合参数类型的错误了。或许你已经意识到使用 Collection<?>也不能工作。回忆一下，你不能把对象放进一个未知类型的集合中去。

解决这个问题的办法是使用 *generic methods*。就像**类型声明，方法的声明也可以被泛型化——就是说，带有一个或者多个类型参数**。

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c){
    for (T o : a) {
        c.add(o); // correct
    }
```

```
}
```

我们可以使用任意集合来调用这个方法 ,只要其元素的类型是数组的元素类型的父类。

```
Object[] oa = new Object[100];

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

fromArrayToCollection(oa, co);// T 指 Object

String[] sa = new String[100];

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

fromArrayToCollection(sa, cs);// T inferred to be String

fromArrayToCollection(sa, co);// T inferred to be Object

Integer[] ia = new Integer[100];

Float[] fa = new Float[100];

Number[] na = new Number[100];

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

fromArrayToCollection(ia, cn);// T inferred to be Number

fromArrayToCollection(fa, cn);// T inferred to be Number
```

```
fromArrayToCollection(na, cn);// T inferred to be Number

fromArrayToCollection(na, co);// T inferred to be Object

fromArrayToCollection(na, cs);// compile-time error
```

注意，我们并没有传送真实类型参数(actual type argument)给一个泛型方法。编译器根据实参为我们推断类型参数的值。它通常推断出能使调用类型正确的最明确的类型参数。