 <T extends Comparable<T>> 和 <T extends Comparable<? super T>> 有什么不同

**<T extends Comparable<T>>**

类型 T 必须实现 Comparable 接口，并且这个接口的类型是 T。只有这样，T 的实例之间才能相互比较大小。例如，在实际调用时若使用的具体类是 Dog，那么 Dog 必须 implements Comparable<Dog>

**<T extends Comparable<? super T>>**

类型 T 必须实现 Comparable 接口，并且这个接口的类型是 T 或 T 的任一父类。这样声明后，T 的实例之间，T 的实例和它的父类的实例之间，可以相互比较大小。例如，在实际调用时若使用的具体类是 Dog (假设 Dog 有一个父类 Animal），Dog 可以从 Animal 那里继承 Comparable<Animal> ，或者自己 implements Comparable<Dog> 。

2 我对 <T extends Comparable<? super T>> 类型参数的理解

光看上面的定义除了摸不着头脑，不会有其它感觉。下面用代码来说明为什么要这样声明。

2.1 代码运行环境

我使用的 JDK 版本是： 1.8.0\_60 ，在 Eclipse 中编译运行。因为注释用了中文，编码采用 UTF-8。如果你要在命令行下编译、运行，编译时要使用 -encoding UTF-8 选项：

javac -encoding UTF-8 TypeParameterTest.java

另外，Eclipse 中的警告、错误信息跟命令行中的不一样（个人感觉 Eclipse 中的信息要好懂一些）。以下的示例以 Eclipse 中的信息为准。

2.2 示例代码

1: package generics3;

2:

3: import java.util.ArrayList;

4: import java.util.Collections;

5: import java.util.List;

6:

7. 7: public class TypeParameterTest

8. 8: {

9. 9: //第一种声明：简单，灵活性低

10. 10: public static <T extends Comparable<T>> void mySort1(List<T> list)

11. 11: {

12. 12: Collections.sort(list);

13. 13: }

14. 14:

15. 15: //第二种声明：复杂，灵活性高

16. 16: public static <T extends Comparable<? super T>> void mySort2(List<T> list)

17. 17: {

18. 18: Collections.sort(list);

19. 19: }

20. 20:

21. 21: public static void main(String[] args)

22. 22: {

23. 23: //在这个方法中要创建一个 Animal List 和一个 Dog List，然后分别调用两个排序方法。

24. 24: }

25. 25: }

26. 26:

27. 27: class Animal implements Comparable<Animal>

28. 28: {

29. 29: protected int age;

30. 30:

31. 31: public Animal(int age)

32. 32:

33. 33: {

34. 34: this.age = age;

35. 35: }

36. 36:

37. 37: //使用年龄与另一实例比较大小

38. 38: @Override

39. 39: public int compareTo(Animal other)

40. 40: {

41. 41: return this.age - other.age;

42. 42: }

43. 43: }

44. 44:

45. 45: class Dog extends Animal

46. 46: {

47. 47: public Dog(int age)

48. 48: {

49. 49: super(age);

50. 50: }

51. 51: }

上面的代码包括三个类：

1. Animal 实现了 Comparable<Animal> 接口，通过年龄来比较实例的大小
2. Dog 继承自 Animal 。
3. TypeParameterTest 类中提供了两个排序方法和测试用的 main() 方法：
   * mySort1() 使用 <T extends Comparable<T>> 类型参数
   * mySort2() 使用 <T extends Comparable<? super T>> 类型参数
   * main() 测试方法。在这个方法中要创建一个 Animal List 和一个 Dog List ，然后分别调用两个排序方法

### 2.3 测试 mySort1() 方法

1. 1: // 创建一个 Animal List

2. 2: List<Animal> animals = new ArrayList<Animal>();

3. 3: animals.add(new Animal(25));

4. 4: animals.add(new Dog(35));

5. 5:

6. 6: // 创建一个 Dog List

7. 7: List<Dog> dogs = new ArrayList<Dog>();

8. 8: dogs.add(new Dog(5));

9. 9: dogs.add(new Dog(18));

10. 10:

11. 11: // 测试 mySort1() 方法

12. 12: mySort1(animals);

13. 13: mySort1(dogs);

Line 13 出编译错误了。Eclipse 说：

The method mySort1(List<T>) in the type TypeParameterTest is not applicable for the arguments (List<Dog>)

为什么会出错误呢？ mySort1() 方法的类型参数是 <T extends Comparable<T>> ，它要求的类型参数是类型为 T 的Comparable 。

如果传入的是 List<Animal> ，没问题，因为 Animal implements Comparable<Animal> 。

但是，如果传入的参数是 List<Dog> 有问题，因为 Dog 没有 implements Comparable<Dog> ，它只是从 Animal 继承了一个Comparable<Animal> 。

不知道大家注意到没有，那个 animals list 中实际上是包含一个 Dog 实例的。如果你碰上类似的情况（子类 list 不能传入到一个方法中），可以考虑把子类实例放到一个父类 List 中，避免编译错误。

### 2.4 测试 mySort2() 方法

1. 1: public static void main(String[] args)

1. 2: {

2. 3: // 创建一个 Animal List

3. 4: List<Animal> animals = new ArrayList<Animal>();

4. 5: animals.add(new Animal(25));

5. 6: animals.add(new Dog(35));

6. 7:

7. 8: // 创建一个 Dog List

8. 9: List<Dog> dogs = new ArrayList<Dog>();

9. 10: dogs.add(new Dog(5));

10. 11: dogs.add(new Dog(18));

11. 12:

12. 13: // 测试 mySort2() 方法

13. 14: mySort2(animals);

14. 15: mySort2(dogs);

15. 16: }

两个方法调用都没有问题。 第二个方法不但可以接受 Animal implements Comparable<Animal> 这样的参数，也可以接收： Dog implements Comparable<Animal> 这样的参数。

### 2.5 Dog 可以 implements Comparable<Dog> 吗？

如果让 Dog implements Comparable<Dog> 不也可以解决前面的那个编译错误吗？

1. 1: class Dog extends Animal implements Comparable<Dog>

2. 2: {

3. 3: public Dog(int age)

4. 4: {

5. 5: super(age);

6. 6: }

7. 7: }

很不幸，出错了。Eclipse 说：

The interface Comparable cannot be implemented more than once with different arguments: Comparable<Animal> and Comparable<Dog>

就是说，Dog 已经从父类 Animal 那里继承了一个 Comparable ，它不能再实现一个 Comparable 。

如果子类不喜欢父类的实现怎么办？ Override 父类的 public int compareTo(Animal other) 方法。

### 2.6 <T extends Comparable<? super T>> 类型参数声明的好处

对 Animal/Dog 这两个有父子关系的类来说： <T extends Comparable<? super T>> 可以接受 List<Animal> ，也可以接收 List<Dog> 。 而 <T extends Comparable<T>> 只可以接收 List<Animal>

所以，<T extends Comparable<? super T>> 这样的类型参数对所传入的参数限制更少，提高了 API 的灵活性。总的来说，在保证类型安全的前提下，要使用限制最少的类型参数。

3 其他

3.1 JDK 中的例子

JDK 中这样的例子很多，比如 java.util.Date 和 java.sql.Date 这两个类：

1. public class Date

2. implements java.io.Serializable, Cloneable, Comparable<Date>

3.

4. public class Date extends java.util.Date

* java.sql.Date 是 java.util.Date 的子类。
* java.util.Date 实现了 Comparable<java.util.Date>~，所以 ~java.sql.Date 也拥有了 Comparable<java.util.Date> 类型。
* java.sql.Date 不能再 implements Comparable<java.sql.Date> 。
* 如果你有一个 List<java.sql.Date> 并对它排序的话，只能传给拥有 <T extends Comparable<? super T>> 这种类型参数的方法。

3.2 《Effective Java》 一书对 <T extends Comparable<? super T>> 这种类型参数的解释

这本书使用 Produce-Extends, Consume-Super (PESC) 原则来解释。这个原则不但可以帮你理解复杂的声明，而且可以指导你在定义类型参数时，何时使用 extends ，何时使用 super，有助于你写出复杂的、适应性强的类型参数来。

有兴趣的同学可以看看这本书的 Item 28: Use bounded wildcards to increase API flexibility

3.3 泛型是个脑力活

简单的泛型很好理解很好用，但稍微复杂一点，就变得很难理解。

3.3.1 脑子开窍开大了

在琢磨这个问题时，我脑洞一开，心想，T 这样的东西太一般化，有点摸不着头脑，不好理解。如果把 T 换成一个具体类，应该会好理解。于是我就想出了这样两个声明：

<Dog **extends** **Comparable**<**Dog**>>

<Dog **extends** **Comparable**<? **super** **Dog**>>

我挺得意，觉得这样先用具体的类理解，然后再换成一般的类型，由具体到一般，多符合逻辑啊！后来发现这样的声明有个大问题，Eclipse 给了个黄色警告：

The type parameter Dog is hiding the type Dog

上面这句话翻译过来就是： **类型参数** Dog 掩盖了 **类型** Dog 。

在 <Dog extends Comparable<Dog>> 这个声明中，extends 前面的部分必须是类型参数。类型参数一般用 T，E 这样的大写字母，但也可以是小写或者一个单词（只要是个标识符就行）。所以，Dog 在这里是一个类型参数，不是一个具体类。但我已经创建过一个具体的 Dog 类了。怎么办？类型参数 Dog 赢了，具体类 Dog 暂时靠边站。类似于你有一个实例变量 x 。然后你在一个方法中又声明了一个局部变量也叫 x 。在执行这个方法时，方法中的这个局部变量 x 就暂时掩盖了(shadow) 实例变量 x 。

3.3.2 脑子一点也不开窍

有时候想得多了，脑子就糊涂了，一点儿也不开窍，连简单问题也不明白了。 比如，我可以这样定义一个方法：

1. public static <T extends Animal> void mySort3(List<T> list)

2. {

3. Collections.sort(list);

4. }

也可以这样定义一个方法：

1. public static void mySort4(List<? extends Animal> list)

2. {

3. Collections.sort(list);

4. }

第二个方法没有 T ，也能实现跟第一个方法同样的功能，我为什么非得要一个 T 呢？在脑子思虑过度的情况下，进死胡同了。在我准备放狗搜之前，总算想明白了。

第二个方法中，参数是： List<? extends Animal> list 。 这个方法可以接收 List<Animal> ，也可以接收 List<Dog> 。这里没有使用类型参数，只是使用泛型的限定符对所传入的 List 的类型做了一个限定。

而在第一个方法中，使用了一个类型参数 T 。这个 T 可以是 Animal ，也可以是 Animal 的子类 Dog。

在第一个方法中，看不出定义一个类型参数有什么作用。但是，类型参数不但可以在方法参数中使用，也可以在方法返回值和方法体内使用。比如下面这个方法：

1. 1: public <T extends Comparable<? super T>> T test1(T t, List<T> list)

2. 2: {

3. 3: for (T element : list)

4. 4: {

5. 5: if (element.equals(t))

6. 6: return t;

7. 7: }

8. 8: return null;

9. 9: }

你定义了一个类型参数 T ，这个 T 定义成 ： <T extends Comparable<? super T>> 。定义好之后，你就可以在参数中，返回值中，以及方法体内使用这个 T 了。如果不使用类型参数，是达不到这种效果的。

你也可以定义多个类型参数，并让这些参数之间有关联：

#### 1. 1: public <T, S extends T> T test2(T t, S s)

#### 2. 2: {

#### 3. 3: return s;

#### 4. 4: }

3.3.3 多练习练习

我从 JDK 中找了两段程序，看看能不能看明白。

1. 程序一

TreeSet 类是这样声明的：

* 1. 1: public class TreeSet<E> extends AbstractSet<E>
  2. 2: implements NavigableSet<E>, Cloneable, java.io.Serializable

它有个 constructor 是这样定义的：

* 1. 1: public TreeSet(Comparator<? super E> comparator) {
  2. 2: this(new **TreeMap**<>(comparator));
  3. 3: }

1. 程序二

Collections 类的 max() 方法：

* 1. 1: public static <**T**> T max(Collection<? extends T> coll, Comparator<? super T> comp) {
  2. 2: if (comp==null)
  3. 3: return (**T**)max((**Collection**) coll);
  4. 4:
  5. 5: **Iterator**<? **extends** **T**> **i** = coll.iterator();
  6. 6: **T** **candidate** = i.next();
  7. 7:
  8. 8: while (i.hasNext()) {
  9. 9: **T** **next** = i.next();
  10. 10: if (comp.compare(next, candidate) > 0)
  11. 11: candidate = next;
  12. 12: }
  13. 13: return candidate;
  14. 14: }

这段程序也展示了类型参数的另一种用法：类型参数本身只是简简单单的 <T> ，但在方法参数和方法体中，却以 T 为基础，向上向下进行了扩展： <? super T> ， <? extends T>

## 4 推荐一本书：《SCJP Sun Certified Programmer for Java 6 Exam 310-065》

这本书的作者是：Kathy Sierra 和 Bert Bates，所以此书也被称为“KB书”。从书名上就可以看出这本书是为 SCJP 6 认证考试而写的。因为是认证考试教材，这本书讲的不是很深，有很多方面也没有涉及到。但这本书对 Java 基本概念的讲解非常透彻而又不啰嗦。它不但是认证宝典中的宝典，而且也非常适合已经入门的 Java 程序员阅读。这本书现在有 Java 7 的版本，不知道以后会不会出 Java 8 版本。如果出的话，我会去买一本。