06 泛型

泛型是一种“代码模板”，可以用一套代码套用各种类型。

本节我们详细讨论Java的泛型编程。

什么是泛型

在讲解什么是泛型之前，我们先观察Java标准库提供的`ArrayList`，它可以看作“可变长度”的数组，因为用起来比数组更方便。

实际上`ArrayList`内部就是一个`Object[]`数组，配合存储一个当前分配的长度，就可以充当“可变数组”：

```
public class ArrayList {
    private Object[] array;
    private int size;
    public void add(Object e) {...}
    public void remove(int index) {...}
    public Object get(int index) {...}
}
```

如果用上述`ArrayList`存储`String`类型，会有这么几个缺点：

- 需要强制转型；
- 不方便，易出错。

例如，代码必须这么写：

```
ArrayList list = new ArrayList();
list.add("Hello");
// 获取到Object，必须强制转型为String:
String first = (String) list.get(0);
```

很容易出现`ClassCastException`，因为容易“误转型”：

```
list.add(new Integer(123));
// ERROR: ClassCastException:
String second = (String) list.get(1);
```

要解决上述问题，我们可以为`String`单独编写一种`ArrayList`：

```
public class StringArrayList {
    private String[] array;
    private int size;
    public void add(String e) {...}
    public void remove(int index) {...}
    public String get(int index) {...}
}
```

这样一来，存入的必须是`String`，取出的也一定是`String`，不需要强制转型，因为编译器会强制检查放入的类型：

```
StringArrayList list = new StringArrayList();
list.add("Hello");
String first = list.get(0);
// 编译错误：不允许放入非String类型:
list.add(new Integer(123));
```

问题暂时解决。

然而，新的问题是，如果要存储`Integer`，还需要为`Integer`单独编写一种`ArrayList`：

```
public class IntegerArrayList {
    private Integer[] array;
    private int size;
    public void add(Integer e) {...}
    public void remove(int index) {...}
    public Integer get(int index) {...}
}
```

实际上，还需要为其他所有class单独编写一种`ArrayList`：

- LongArrayList
- DoubleArrayList
- PersonArrayList

- ...

这是不可能的，JDK的class就有上千个，而且它还不知道其他人编写的class。

为了解决新的问题，我们必须把`ArrayList`变成一种模板：`ArrayList`，代码如下：

```
public class ArrayList<T> {
    private T[] array;
    private int size;
    public void add(T e) {...}
    public void remove(int index) {...}
    public T get(int index) {...}
}
```

`T`可以是任何class。这样一来，我们就实现了：编写一次模版，可以创建任意类型的`ArrayList`：

```
// 创建可以存储String的ArrayList:
ArrayList<String> strList = new ArrayList<String>();
// 创建可以存储Float的ArrayList:
ArrayList<Float> floatList = new ArrayList<Float>();
// 创建可以存储Person的ArrayList:
ArrayList<Person> personList = new ArrayList<Person>();
```

因此，泛型就是定义一种模板，例如`ArrayList`，然后在代码中为用到的类创建对应的`ArrayList<类型>`：

```
ArrayList<String> strList = new ArrayList<String>();
```

由编译器针对类型作检查：

```
strList.add("hello"); // OK
String s = strList.get(0); // OK
strList.add(new Integer(123)); // compile error!
Integer n = strList.get(0); // compile error!
```

这样一来，既实现了编写一次，万能匹配，又通过编译器保证了类型安全：这就是泛型。

向上转型

在Java标准库中的`ArrayList`实现了`List`接口，它可以向上转型为`List`：

```
public class ArrayList<T> implements List<T> {
    // ...
}

List<String> list = new ArrayList<String>();
```

即类型`ArrayList`可以向上转型为`List`。

要*特别注意*：不能把`ArrayList`向上转型为`ArrayList`或`List`。

这是为什么呢？假设`ArrayList`可以向上转型为`ArrayList`，观察一下代码：

```
// 创建ArrayList<Integer>类型：
ArrayList<Integer> integerList = new ArrayList<Integer>();
// 添加一个Integer：
integerList.add(new Integer(123));
// “向上转型”为ArrayList<Number>：
ArrayList<Number> numberList = integerList;
// 添加一个Float，因为Float也是Number：
numberList.add(new Float(12.34));
// 从ArrayList<Integer>获取索引为1的元素（即添加的Float）：
Integer n = integerList.get(1); // ClassCastException!
```

我们把一个`ArrayList`转型为`ArrayList`类型后，这个`ArrayList`就可以接受`Float`类型，因为`Float`是`Number`的子类。但是，`ArrayList`实际上和`ArrayList`是同一个对象，也就是`ArrayList`类型，它不可能接受`Float`类型，所以在获取`Integer`的时候将产生`ClassCastException`。

实际上，编译器为了避免这种错误，根本就不允许把`ArrayList`转型为`ArrayList`。

`ArrayList<Integer>`和`ArrayList<Number>`两者完全没有继承关系。

小结

- 泛型就是编写模板代码来适应任意类型；
- 泛型的好处是使用时不必对类型进行强制转换，它通过编译器对类型进行检查；
- 注意泛型的继承关系：可以把`ArrayList`向上转型为`List`（`T`不能变！），但不能把`ArrayList`向上转型为`ArrayList`（`T`不能变成父类）。

使用泛型

使用`ArrayList`时，如果不定义泛型类型时，泛型类型实际上就是`Object`：

```
// 编译器警告：
List list = new ArrayList();
list.add("Hello");
list.add("World");
String first = (String) list.get(0);
String second = (String) list.get(1);
```

此时，只能把`当作Object`使用，没有发挥泛型的优势。

当我们定义泛型类型`后，List的泛型接口变为强类型List`：

```
// 无编译器警告:
List<String> list = new ArrayList<String>();
list.add("Hello");
list.add("World");
// 无强制转型:
String first = list.get(0);
String second = list.get(1);
```

当我们定义泛型类型`后，List的泛型接口变为强类型List`：

```
List<Number> list = new ArrayList<Number>();
list.add(new Integer(123));
list.add(new Double(12.34));
Number first = list.get(0);
Number second = list.get(1);
```

编译器如果能自动推断出泛型类型，就可以省略后面的泛型类型。例如，对于下面的代码：

```
List<Number> list = new ArrayList<Number>();
```

编译器看到泛型类型`List`就可以自动推断出后面的`ArrayList`的泛型类型必须是`ArrayList`，因此，可以把代码简写为：

```
// 可以省略后面的Number，编译器可以自动推断泛型类型:
List<Number> list = new ArrayList<>();
```

泛型接口

除了`ArrayList`使用了泛型，还可以在接口中使用泛型。例如，`Arrays.sort(Object[])`可以对任意数组进行排序，但待排序的元素必须实现`Comparable`这个泛型接口：

```
public interface Comparable<T> {
    /**
     * 返回-1: 当前实例比参数o小
     * 返回0: 当前实例与参数o相等
     * 返回1: 当前实例比参数o大
     */
    int compareTo(T o);
}
```

可以直接对`String`数组进行排序：

```
// sort
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        String[] ss = new String[] { "Orange", "Apple",
"Pear" };
        Arrays.sort(ss);
        System.out.println(Arrays.toString(ss));
    }
}
```

这是因为`String`本身已经实现了`Comparable`接口。如果换成我们自定义的`Person`类型试试：

```
// sort
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        Person[] ps = new Person[] {
            new Person("Bob", 61),
            new Person("Alice", 88),
            new Person("Lily", 75),
        };
        Arrays.sort(ps);
        System.out.println(Arrays.toString(ps));
    }
}

class Person {
    String name;
    int score;
    Person(String name, int score) {
        this.name = name;
        this.score = score;
    }
    public String toString() {
        return this.name + "," + this.score;
    }
}
```

运行程序，我们会得到`ClassCastException`，即无法将`Person`转型为`Comparable`。我们修改代码，让`Person`实现`Comparable`接口：

```
// sort
import java.util.Arrays;
```

```
public class Main {
    public static void main(String[] args) {
        Person[] ps = new Person[] {
            new Person("Bob", 61),
            new Person("Alice", 88),
            new Person("Lily", 75),
        };
        Arrays.sort(ps);
        System.out.println(Arrays.toString(ps));
    }
}
class Person implements Comparable<Person> {
    String name;
    int score;
    Person(String name, int score) {
        this.name = name;
        this.score = score;
    }
    public int compareTo(Person other) {
        return this.name.compareTo(other.name);
    }
    public String toString() {
        return this.name + "," + this.score;
    }
}
```

运行上述代码，可以正确实现按`name`进行排序。

也可以修改比较逻辑，例如，按`score`从高到低排序。请自行修改测试。

小结

- 使用泛型时，把泛型参数`替换为需要的class类型，例如：ArrayList，ArrayList`等；
- 可以省略编译器能自动推断出的类型，例如：`List list = new ArrayList<>();`；
- 不指定泛型参数类型时，编译器会给出警告，且只能将`视为Object`类型；
  -可以在接口中定义泛型类型，实现此接口的类必须实现正确的泛型类型。

编写泛型

编写泛型类比普通类要复杂。通常来说，泛型类一般用在集合类中，例如`ArrayList`，我们很少需要编写泛型类。

如果我们确实需要编写一个泛型类，那么，应该如何编写它？

可以按照以下步骤来编写一个泛型类。

首先，按照某种类型，例如：`String`，来编写类：

```
public class Pair {
    private String first;
    private String last;
    public Pair(String first, String last) {
        this.first = first;
        this.last = last;
    }
    public String getFirst() {
        return first;
    }
    public String getLast() {
        return last;
    }
}
```

然后，标记所有的特定类型，这里是`String`：

```
public class Pair {
    private String first;
    private String last;
    public Pair(String first, String last) {
        this.first = first;
        this.last = last;
    }
    public String getFirst() {
        return first;
    }
    public String getLast() {
        return last;
    }
}
```

最后，把特定类型`String`替换为`T`，并申明``：

```
public class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}
```

熟练后即可直接从`T`开始编写。

静态方法

编写泛型类时，要特别注意，泛型类型``不能用于静态方法。例如：

```
public class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() { ... }
    public T getLast() { ... }

    // 对静态方法使用<T>:
    public static Pair<T> create(T first, T last) {
        return new Pair<T>(first, last);
    }
}
```

上述代码会导致编译错误，我们无法在静态方法`create()`的方法参数和返回类型上使用泛型类型`T`。

有些同学在网上搜索发现，可以在`static`修饰符后面加一个``，编译就能通过：

```
public class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() { ... }
    public T getLast() { ... }

    // 可以编译通过:
    public static <T> Pair<T> create(T first, T last) {
        return new Pair<T>(first, last);
    }
}
```

但实际上，这个`和`Pair`类型的`已经没有任何关系了。

对于静态方法，我们可以单独改写为“泛型”方法，只需要使用另一个类型即可。对于上面的`create()`静态方法，我们应该把它改为另一种泛型类型，例如，``：

```
```

```
public class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() { ... }
    public T getLast() { ... }

    // 静态泛型方法应该使用其他类型区分:
    public static <K> Pair<K> create(K first, K last) {
        return new Pair<K>(first, last);
    }
}
```

这样才能清楚地将静态方法的泛型类型和实例类型的泛型类型区分开。

多个泛型类型

泛型还可以定义多种类型。例如，我们希望`Pair`不总是存储两个类型一样的对象，就可以使用类型``：

```
public class Pair<T, K> {
    private T first;
    private K last;
    public Pair(T first, K last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() { ... }
    public K getLast() { ... }
}
```

使用的时候，需要指出两种类型：

```
Pair<String, Integer> p = new Pair<>("test", 123);
```

Java标准库的`Map`就是使用两种泛型类型的例子。它对Key使用一种类型，对Value使用另一种类型。

小结

- 编写泛型时，需要定义泛型类型``；
- 静态方法不能引用泛型类型，必须定义其他类型（例如）来实现静态泛型方法；
  -泛型可以同时定义多种类型，例如`Map`。

擦拭法

泛型是一种类似”模板代码“的技术，不同语言的泛型实现方式不一定相同。

Java语言的泛型实现方式是擦拭法（Type Erasure）。

所谓擦拭法是指，虚拟机对泛型其实一无所知，所有的工作都是编译器做的。

例如，我们编写了一个泛型类`Pair`，这是编译器看到的代码：

```
public class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}
```

而虚拟机根本不知道泛型。这是虚拟机执行的代码：

```
public class Pair {
    private Object first;
    private Object last;
    public Pair(Object first, Object last) {
        this.first = first;
        this.last = last;
    }
    public Object getFirst() {
        return first;
    }
    public Object getLast() {
        return last;
    }
}
```

因此，Java使用擦拭法实现泛型，导致了：

- 编译器把类型`视为Object`；
- 编译器根据``实现安全的强制转型。

使用泛型的时候，我们编写的代码也是编译器看到的代码：

```
Pair<String> p = new Pair<>("Hello", "world");
String first = p.getFirst();
String last = p.getLast();
```

而虚拟机执行的代码并没有泛型：

```
Pair p = new Pair("Hello", "world");
String first = (String) p.getFirst();
String last = (String) p.getLast();
```

所以，Java的泛型是由编译器在编译时实行的，编译器内部永远把所有类型`T`视为`Object`处理，但是，在需要转型的时候，编译器会根据`T`的类型自动为我们实行安全地强制转型。

了解了Java泛型的实现方式——擦拭法，我们就知道了Java泛型的局限：

局限一：`不能是基本类型，例如int，因为实际类型是Object，Object`类型无法持有基本类型：

```
Pair<int> p = new Pair<>(1, 2); // compile error!
```

局限二：无法取得带泛型的`Class`。观察以下代码：

```
public class Main {
    public static void main(String[] args) {
        Pair<String> p1 = new Pair<>("Hello", "world");
        Pair<Integer> p2 = new Pair<>(123, 456);
        Class c1 = p1.getClass();
        Class c2 = p2.getClass();
        System.out.println(c1==c2); // true
        System.out.println(c1==Pair.class); // true
    }
}

class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}
```

因为`T`是`Object`，我们对`Pair`和`Pair`类型获取`Class`时，获取到的是同一个`Class`，也就是`Pair`类的`Class`。

换句话说，所有泛型实例，无论`T`的类型是什么，`getClass()`返回同一个`Class`实例，因为编译后它们全部都是`Pair`。

局限三：无法判断带泛型的`Class`：

```
Pair<Integer> p = new Pair<>(123, 456);
// Compile error:
if (p instanceof Pair<String>.class) {
}
```

原因和前面一样，并不存在`Pair.class`，而是只有唯一的`Pair.class`。

局限四：不能实例化`T`类型：

```
public class Pair<T> {
    private T first;
    private T last;
    public Pair() {
        // Compile error:
        first = new T();
        last = new T();
    }
}
```

上述代码无法通过编译，因为构造方法的两行语句：

```
first = new T();
last = new T();
```

擦拭后实际上变成了：

```
first = new Object();
last = new Object();
```

这样一来，创建`new Pair()`和创建`new Pair()`就全部成了`Object`，显然编译器要阻止这种类型不对的代码。

要实例化`T`类型，我们必须借助额外的`Class`参数：

```
public class Pair<T> {
    private T first;
    private T last;
    public Pair(Class<T> clazz) {
        first = clazz.newInstance();
        last = clazz.newInstance();
    }
}
```

上述代码借助`Class`参数并通过反射来实例化`T`类型，使用的时候，也必须传入`Class`。例如：

```
Pair<String> pair = new Pair<>(String.class);
```

因为传入了`Class`的实例，所以我们借助`String.class`就可以实例化`String`类型。

不恰当的覆写方法

有些时候，一个看似正确定义的方法会无法通过编译。例如：

```
public class Pair<T> {
    public boolean equals(T t) {
        return this == t;
    }
}
```

这是因为，定义的`equals(T t)`方法实际上会被擦拭成`equals(Object t)`，而这个方法是继承自`Object`的，编译器会阻止一个实际上会变成覆写的泛型方法定义。

换个方法名，避开与`Object.equals(Object)`的冲突就可以成功编译：

```
public class Pair<T> {
    public boolean same(T t) {
        return this == t;
    }
}
```

泛型继承

一个类可以继承自一个泛型类。例如：父类的类型是`Pair`，子类的类型是`IntPair`，可以这么继承：

```
public class IntPair extends Pair<Integer> {
}
```

使用的时候，因为子类`IntPair`并没有泛型类型，所以，正常使用即可：

```
IntPair ip = new IntPair(1, 2);
```

前面讲了，我们无法获取`Pair`的`T`类型，即给定一个变量`Pair p`，无法从`p`中获取到`Integer`类型。

但是，在父类是泛型类型的情况下，编译器就必须把类型`T`（对`IntPair`来说，也就是`Integer`类型）保存到子类的class文件中，不然编译器就不知道`IntPair`只能存取`Integer`这种类型。

在继承了泛型类型的情况下，子类可以获取父类的泛型类型。例如：`IntPair`可以获取到父类的泛型类型`Integer`。获取父类的泛型类型代码比较复杂：

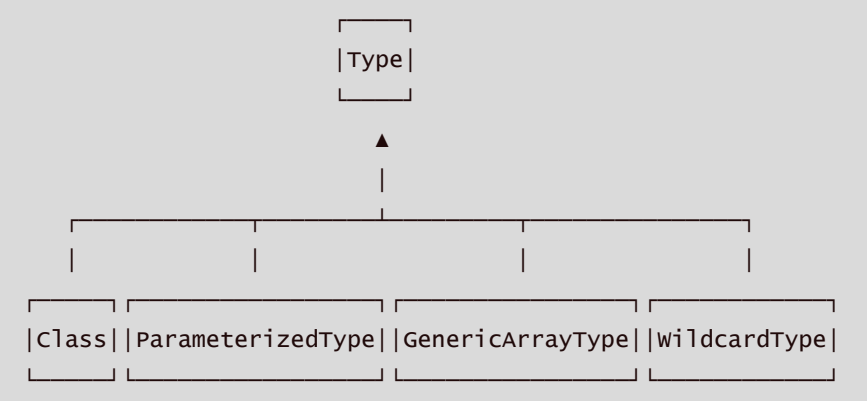
```
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;

public class Main {
    public static void main(String[] args) {
                Class<IntPair> clazz = IntPair.class;
        Type t = clazz.getGenericSuperclass();
        if (t instanceof ParameterizedType) {
            ParameterizedType pt = (ParameterizedType) t;
            Type[] types = pt.getActualTypeArguments(); //
可能有多个泛型类型
            Type firstType = types[0]; // 取第一个泛型类型
            Class<?> typeClass = (Class<?>) firstType;
            System.out.println(typeClass); // Integer
        }
    }
}

class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}

class IntPair extends Pair<Integer> {
    public IntPair(Integer first, Integer last) {
        super(first, last);
    }
}
```

因为Java引入了泛型，所以，只用`Class`来标识类型已经不够了。实际上，Java的类型系统结构如下：



小结

- Java的泛型是采用擦拭法实现的；
- 擦拭法决定了泛型`<T>`：
    - 不能是基本类型，例如：`int`；
    - 不能获取带泛型类型的`Class`，例如：`Pair.class`；
    - 不能判断带泛型类型的类型，例如：`x instanceof Pair`；
    - 不能实例化`T`类型，例如：`new T()`。
- 泛型方法要防止重复定义方法，例如：`public boolean equals(T obj)`；
- 子类可以获取父类的泛型类型``。

extends通配符

我们前面已经讲到了泛型的继承关系：`Pair`不是`Pair`的子类。

假设我们定义了`Pair`：

```
public class Pair<T> { ... }
```

然后，我们又针对`Pair`类型写了一个静态方法，它接收的参数类型是`Pair`：

```
public class PairHelper {
    static int add(Pair<Number> p) {
        Number first = p.getFirst();
        Number last = p.getLast();
        return first.intValue() + last.intValue();
    }
}
```

上述代码是可以正常编译的。使用的时候，我们传入：

```
int sum = PairHelper.add(new Pair<Number>(1, 2));
```

注意：传入的类型是`Pair`，实际参数类型是`(Integer, Integer)`。

既然实际参数是`Integer`类型，试试传入`Pair`：

```
public class Main {
        public static void main(String[] args) {
        Pair<Integer> p = new Pair<>(123, 456);
        int n = add(p);
        System.out.println(n);
    }

    static int add(Pair<Number> p) {
        Number first = p.getFirst();
        Number last = p.getLast();
        return first.intValue() + last.intValue();
    }
}

class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}
```

直接运行，会得到一个编译错误：

```
incompatible types: Pair<Integer> cannot be converted to
Pair<Number>
```

原因很明显，因为`Pair`不是`Pair`的子类，因此，`add(Pair)`不接受参数类型`Pair`。

但是从`add()`方法的代码可知，传入`Pair`是完全符合内部代码的类型规范，因为语句：

```
Number first = p.getFirst();
Number last = p.getLast();
```

实际类型是`Integer`，引用类型是`Number`，没有问题。问题在于方法参数类型定死了只能传入`Pair`。

有没有办法使得方法参数接受`Pair`？办法是有的，这就是使用`Pair`使得方法接收所有泛型类型为`Number`或`Number`子类的`Pair`类型。我们把代码改写如下：

```
public class Main {
    public static void main(String[] args) {
        Pair<Integer> p = new Pair<>(123, 456);
        int n = add(p);
        System.out.println(n);
    }

    static int add(Pair<? extends Number> p) {
        Number first = p.getFirst();
        Number last = p.getLast();
        return first.intValue() + last.intValue();
    }
    }

class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}
```

这样一来，给方法传入`Pair`类型时，它符合参数`Pair`类型。这种使用`的泛型定义称之为上界通配符（Upper Bounds Wildcards），即把泛型类型T的上界限定在Number`了。

除了可以传入`Pair`类型，我们还可以传入`Pair`类型，`Pair`类型等等，因为`Double`和`BigDecimal`都是`Number`的子类。

如果我们考察对`Pair`类型调用`getFirst()`方法，实际的方法签名变成了：

```
<? extends Number> getFirst();
```

即返回值是`Number`或`Number`的子类，因此，可以安全赋值给`Number`类型的变量：

```
Number x = p.getFirst();
```

然后，我们不可预测实际类型就是`Integer`，例如，下面的代码是无法通过编译的：

```
Integer x = p.getFirst();
```

这是因为实际的返回类型可能是`Integer`，也可能是`Double`或者其他类型，编译器只能确定类型一定是`Number`的子类（包括`Number`类型本身），但具体类型无法确定。

我们再来考察一下`Pair`的`set`方法：

```
public class Main {
    public static void main(String[] args) {
        Pair<Integer> p = new Pair<>(123, 456);
        int n = add(p);
        System.out.println(n);
    }

    static int add(Pair<? extends Number> p) {
        Number first = p.getFirst();
        Number last = p.getLast();
        p.setFirst(new Integer(first.intValue() + 100));
        p.setLast(new Integer(last.intValue() + 100));
        return p.getFirst().intValue() +
p.getFirst().intValue();
    }
class Pair<T> {
    private T first;
    private T last;

    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }

    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
    public void setFirst(T first) {
        this.first = first;
    }
    public void setLast(T last) {
        this.last = last;
    }
}
```

不出意外，我们会得到一个编译错误：

```
incompatible types: Integer cannot be converted to CAP#1
where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
```

编译错误发生在`p.setFirst()`传入的参数是`Integer`类型。有些童鞋会问了，既然`p`的定义是`Pair`，那么`setFirst(? extends Number)`为什么不能传入`Integer`？

原因还在于擦拭法。如果我们传入的`p`是`Pair`，显然它满足参数定义`Pair`，然而，`Pair`的`setFirst()`显然无法接受`Integer`类型。

这就是`通配符的一个重要限制：方法参数签名setFirst(? extends Number)无法传递任何Number类型给setFirst(? extends Number)`。

这里唯一的例外是可以给方法参数传入`null`：

```
p.setFirst(null); // ok, 但是后面会抛出NullPointerException
p.getFirst().intValue(); // NullPointerException
```

extends通配符的作用

如果我们考察Java标准库的`java.util.List`接口，它实现的是一个类似“可变数组”的列表，主要功能包括：

```
public interface List<T> {
    int size(); // 获取个数
    T get(int index); // 根据索引获取指定元素
    void add(T t); // 添加一个新元素
    void remove(T t); // 删除一个已有元素
}
```

现在，让我们定义一个方法来处理列表的每个元素：

```
int sumOfList(List<? extends Integer> list) {
    int sum = 0;
    for (int i=0; i<list.size(); i++) {
        Integer n = list.get(i);
        sum = sum + n;
    }
    return sum;
}
```

为什么我们定义的方法参数类型是`List`而不是`List`？从方法内部代码看，传入`List`或者`List`是完全一样的，但是，注意到`List`的限制：

- 允许调用`get()`方法获取`Integer`的引用；

- 不允许调用`set(? extends Integer)`方法并传入任何`Integer`的引用（`null`除外）。

因此，方法参数类型`List`表明了该方法内部只会读取`List`的元素，不会修改`List`的元素（因为无法调用`add(? extends Integer)`、`remove(? extends Integer)`这些方法。换句话说，这是一个对参数`List`进行只读的方法（恶意调用`set(null)`除外）。

使用extends限定T类型

在定义泛型类型`Pair`的时候，也可以使用`extends`通配符来限定`T`的类型：

```
public class Pair<T extends Number> { ... }
```

现在，我们只能定义：

```
Pair<Number> p1 = null;
Pair<Integer> p2 = new Pair<>(1, 2);
Pair<Double> p3 = null;
```

因为`Number`、`Integer`和`Double`都符合``。

非`Number`类型将无法通过编译：

```
Pair<String> p1 = null; // compile error!
Pair<Object> p2 = null; // compile error!
```

因为`String`、`Object`都不符合`，因为它们不是Number类型或Number`的子类。

小结

使用类似``通配符作为方法参数时表示：

- 方法内部可以调用获取`Number`引用的方法，例如：`Number n = obj.getFirst();`；
- 方法内部无法调用传入`Number`引用的方法（`null`除外），例如：`obj.setFirst(Number n);`。

即一句话总结：使用`extends`通配符表示可以读，不能写。

使用类似``定义泛型类时表示：

- 泛型类型限定为`Number`以及`Number`的子类。

super通配符

我们前面已经讲到了泛型的继承关系：`Pair`不是`Pair`的子类。

考察下面的`set`方法：

```
void set(Pair<Integer> p, Integer first, Integer last) {
    p.setFirst(first);
    p.setLast(last);
}
```

传入`Pair`是允许的，但是传入`Pair`是不允许的。

和`extends`通配符相反，这次，我们希望接受`Pair`类型，以及`Pair`、`Pair`，因为`Number`和`Object`是`Integer`的父类，`setFirst(Number)`和`setFirst(Object)`实际上允许接受`Integer`类型。

我们使用`super`通配符来改写这个方法：

```
void set(Pair<? super Integer> p, Integer first, Integer
last) {
    p.setFirst(first);
    p.setLast(last);
}
```

注意到`Pair`表示，方法参数接受所有泛型类型为`Integer`或`Integer`父类的`Pair`类型。

下面的代码可以被正常编译：

```
public class Main {
    public static void main(String[] args) {
        Pair<Number> p1 = new Pair<>(12.3, 4.56);
        Pair<Integer> p2 = new Pair<>(123, 456);
        setSame(p1, 100);
        setSame(p2, 200);
        System.out.println(p1.getFirst() + ", " +
p1.getLast());
        System.out.println(p2.getFirst() + ", " +
p2.getLast());
    }

    static void setSame(Pair<? super Integer> p, Integer
n) {
        p.setFirst(n);
        p.setLast(n);
    }
}

class Pair<T> {
    private T first;
    private T last;

    public Pair(T first, T last) {
```

```
        this.first = first;
        this.last = last;
    }

    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
    public void setFirst(T first) {
        this.first = first;
    }
    public void setLast(T last) {
        this.last = last;
    }
}
```

考察`Pair`的`setFirst()`方法，它的方法签名实际上是：

```
void setFirst(? super Integer);
```

因此，可以安全地传入`Integer`类型。

再考察`Pair`的`getFirst()`方法，它的方法签名实际上是：

```
? super Integer getFirst();
```

这里注意到我们无法使用`Integer`类型来接收`getFirst()`的返回值，即下面的语句将无法通过编译：

```
Integer x = p.getFirst();
```

因为如果传入的实际类型是`Pair`，编译器无法将`Number`类型转型为`Integer`。

注意：虽然`Number`是一个抽象类，我们无法直接实例化它。但是，即便`Number`不是抽象类，这里仍然无法通过编译。此外，传入`Pair`类型时，编译器也无法将`Object`类型转型为`Integer`。

唯一可以接收`getFirst()`方法返回值的是`Object`类型：

```
Object obj = p.getFirst();
```

因此，使用``通配符表示：

- 允许调用`set(? super Integer)`方法传入`Integer`的引用；

- 不允许调用`get()`方法获得`Integer`的引用。

唯一例外是可以获取`Object`的引用：`Object o = p.getFirst()`。

换句话说，使用``通配符作为方法参数，表示方法内部代码对于参数只能写，不能读。

对比extends和super通配符

我们再回顾一下`extends`通配符。作为方法参数，`类型和`类型的区别在于：

- `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）。

一个是允许读不允许写，另一个是允许写不允许读。

先记住上面的结论，我们来看Java标准库的`Collections`类定义的`copy()`方法：

```
public class Collections {
    // 把src的每个元素复制到dest中:
    public static <T> void copy(List<? super T> dest,
List<? extends T> src) {
        for (int i=0; i<src.size(); i++) {
            T t = src.get(i);
            dest.add(t);
        }
    }
}
```

它的作用是把一个`List`的每个元素依次添加到另一个`List`中。它的第一个参数是`List`，表示目标`List`，第二个参数`List`，表示要复制的`List`。我们可以简单地用`for`循环实现复制。在`for`循环中，我们可以看到，对于类型`的变量`src`，我们可以安全地获取类型`T`的引用，而对于类型`的变量`dest`，我们可以安全地传入`T`的引用。

这个`copy()`方法的定义就完美地展示了`extends`和`super`的意图：

- `copy()`方法内部不会读取`dest`，因为不能调用`dest.get()`来获取`T`的引用；
- `copy()`方法内部也不会修改`src`，因为不能调用`src.add(T)`。

这是由编译器检查来实现的。如果在方法代码中意外修改了`src`，或者意外读取了`dest`，就会导致一个编译错误：

```
public class Collections {
    // 把src的每个元素复制到dest中:
    public static <T> void copy(List<? super T> dest,
List<? extends T> src) {
        ...
        T t = dest.get(0); // compile error!
        src.add(t); // compile error!
    }
}
```

这个`copy()`方法的另一个好处是可以安全地把一个`List`添加到`List`，但是无法反过来添加：

```
// copy List<Integer> to List<Number> ok:
List<Number> numList = ...;
List<Integer> intList = ...;
Collections.copy(numList, intList);

// ERROR: cannot copy List<Number> to List<Integer>:
Collections.copy(intList, numList);
```

而这些都是通过`super`和`extends`通配符，并由编译器强制检查来实现的。

PECS原则

何时使用`extends`，何时使用`super`？为了便于记忆，我们可以用PECS原则：Producer Extends Consumer Super。

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

还是以`Collections`的`copy()`方法为例：

```
public class Collections {
    public static <T> void copy(List<? super T> dest,
List<? extends T> src) {
        for (int i=0; i<src.size(); i++) {
            T t = src.get(i); // src是producer
            dest.add(t); // dest是consumer
        }
    }
}
```

需要返回`T`的`src`是生产者，因此声明为`List`，需要写入`T`的`dest`是消费者，因此声明为`List`。

无限定通配符

我们已经讨论了和作为方法参数的作用。实际上，Java的泛型还允许使用无限定通配符（Unbounded Wildcard Type），即只定义一个?：

```
void sample(Pair<?> p) {
}
```

因为`通配符既没有extends，也没有super`，因此：

- 不允许调用set(T)方法并传入引用（null除外）；
- 不允许调用T get()方法并获取T引用（只能获取Object引用）。

换句话说，既不能读，也不能写，那只能做一些null判断：

```
static boolean isNull(Pair<?> p) {
    return p.getFirst() == null || p.getLast() == null;
}
```

大多数情况下，可以引入泛型参数消除通配符：

```
static <T> boolean isNull(Pair<T> p) {
    return p.getFirst() == null || p.getLast() == null;
}
```

`通配符有一个独特的特点，就是：Pair是所有Pair`的超类：

```
public class Main {
    public static void main(String[] args) {
        Pair<Integer> p = new Pair<>(123, 456);
        Pair<?> p2 = p; // 安全地向上转型
        System.out.println(p2.getFirst() + ", " +
p2.getLast());
    }
}

class Pair<T> {
    private T first;
    private T last;

    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }

    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
    public void setFirst(T first) {
```

```
        this.first = first;
    }
    public void setLast(T last) {
        this.last = last;
    }
}
```

上述代码是可以正常编译运行的，因为`Pair`是`Pair`的子类，可以安全地向上转型。

小结

- 使用类似``通配符作为方法参数时表示：
    - 方法内部可以调用传入`Integer`引用的方法，例如：`obj.setFirst(Integer n);`；
    - 方法内部无法调用获取`Integer`引用的方法（`Object`除外），例如：`Integer n = obj.getFirst();`。
- 即使用`super`通配符表示只能写不能读。
- 使用`extends`和`super`通配符要遵循PECS原则。
- 无限定通配符`很少使用，可以用`替换，同时它是所有``类型的超类。

泛型和反射

Java的部分反射API也是泛型。例如：`Class`就是泛型：

```
// compile warning:
Class clazz = String.class;
String str = (String) clazz.newInstance();

// no warning:
Class<String> clazz = String.class;
String str = clazz.newInstance();
```

调用`Class`的`getSuperclass()`方法返回的`Class`类型是`Class`：

```
Class<? super String> sup = String.class.getSuperclass();
```

构造方法`Constructor`也是泛型：

```
Class<Integer> clazz = Integer.class;
Constructor<Integer> cons =
clazz.getConstructor(int.class);
Integer i = cons.newInstance(123);
```

我们可以声明带泛型的数组，但不能用`new`操作符创建带泛型的数组：

```
Pair<String>[] ps = null; // ok
Pair<String>[] ps = new Pair<String>[2]; // compile error!
```

必须通过强制转型实现带泛型的数组：

```
@SuppressWarnings("unchecked")
Pair<String>[] ps = (Pair<String>[]) new Pair[2];
```

使用泛型数组要特别小心，因为数组实际上在运行期没有泛型，编译器可以强制检查变量`ps`，因为它的类型是泛型数组。但是，编译器不会检查变量`arr`，因为它不是泛型数组。因为这两个变量实际上指向同一个数组，所以，操作`arr`可能导致从`ps`获取元素时报错，例如，以下代码演示了不安全地使用带泛型的数组：

```
Pair[] arr = new Pair[2];
Pair<String>[] ps = (Pair<String>[]) arr;

ps[0] = new Pair<String>("a", "b");
arr[1] = new Pair<Integer>(1, 2);

// ClassCastException:
Pair<String> p = ps[1];
String s = p.getFirst();
```

要安全地使用泛型数组，必须扔掉`arr`的引用：

```
@SuppressWarnings("unchecked")
Pair<String>[] ps = (Pair<String>[]) new Pair[2];
```

上面的代码中，由于拿不到原始数组的引用，就只能对泛型数组`ps`进行操作，这种操作就是安全的。

带泛型的数组实际上是编译器的类型擦除：

```
Pair[] arr = new Pair[2];
Pair<String>[] ps = (Pair<String>[]) arr;

System.out.println(ps.getClass() == Pair[].class); // true

String s1 = (String) arr[0].getFirst();
String s2 = ps[0].getFirst();
```

所以我们不能直接创建泛型数组`T[]`，因为擦拭后代码变为`Object[]`：

```
// compile error:
public class Abc<T> {
    T[] createArray() {
        return new T[5];
    }
}
```

必须借助`Class`来创建泛型数组：

```
T[] createArray(Class<T> cls) {
    return (T[]) Array.newInstance(cls, 5);
}
```

我们还可以利用可变参数创建泛型数组`T[]`：

```
public class ArrayHelper {
    @SafeVarargs
    static <T> T[] asArray(T... objs) {
        return objs;
    }
}

String[] ss = ArrayHelper.asArray("a", "b", "c");
Integer[] ns = ArrayHelper.asArray(1, 2, 3);
```

谨慎使用泛型可变参数

在上面的例子中，我们看到，通过：

```
static <T> T[] asArray(T... objs) {
    return objs;
}
```

似乎可以安全地创建一个泛型数组。但实际上，这种方法非常危险。以下代码来自《Effective Java》的示例：

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        String[] arr = asArray("one", "two", "three");
        System.out.println(Arrays.toString(arr));
        // ClassCastException:
        String[] firstTwo = pickTwo("one", "two",
"three");
        System.out.println(Arrays.toString(firstTwo));
    }

    static <K> K[] pickTwo(K k1, K k2, K k3) {
```

```
        return asArray(k1, k2);
    }

    static <T> T[] asArray(T... objs) {
        return objs;
    }
}
```

直接调用`asArray(T...)`似乎没有问题，但是在另一个方法中，我们返回一个泛型数组就会产生`ClassCastException`，原因还是因为擦拭法，在`pickTwo()`方法内部，编译器无法检测`K[]`的正确类型，因此返回了`Object[]`。

如果仔细观察，可以发现编译器对所有可变泛型参数都会发出警告，除非确认完全没有问题，才可以用`@SafeVarargs`消除警告。

如果在方法内部创建了泛型数组，最好不要将它返回给外部使用。

更详细的解释请参考《Effective Java》“Item 32: Combine generics and varargs judiciously”。

小结

- 部分反射API是泛型，例如：`Class`，`Constructor`；
- 可以声明带泛型的数组，但不能直接创建带泛型的数组，必须强制转型；
- 可以通过`Array.newInstance(Class, int)`创建`T[]`数组，需要强制转型；
- 同时使用泛型和可变参数时需要特别小心。