Java泛型

一. 泛型概念的提出（为什么需要泛型）？

首先，我们看下下面这段简短的代码:

```
public class GenericTest {

    public static void main(String[] args) {
        List list = new ArrayList();
        list.add("qqyumidi");
        list.add("corn");
        list.add(100);

        for (int i = 0; i < list.size(); i++) {
            String name = (String) list.get(i); // 1
            System.out.println("name:" + name);
        }
    }
}
```

定义了一个List类型的集合，先向其中加入了两个字符串类型的值，随后加入一个Integer类型的值。这是完全允许的，因为此时list默认的类型为Object类型。在之后的循环中，由于忘记了之前在list中也加入了Integer类型的值或其他编码原因，很容易出现类似于//1中的错误。因为编译阶段正常，而运行时会出现“java.lang.ClassCastException”异常。因此，导致此类错误编码过程中不易发现。

在如上的编码过程中，我们发现主要存在两个问题：

1.当我们将一个对象放入集合中，集合不会记住此对象的类型，当再次从集合中取出此对象时，改对象的编译类型变成了Object类型，但其运行时类型任然为其本身类型。

2.因此，//1处取出集合元素时需要人为的强制类型转化到具体的目标类型，且很容易出现“java.lang.ClassCastException”异常。

那么有没有什么办法可以使集合能够记住集合内元素各类型，且能够达到只要编译时不出现问题，运行时就不会出现“java.lang.ClassCastException”异常呢？答案就是使用泛型。

二.什么是泛型？

泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。
看着好像有点复杂，首先我们看下上面那个例子采用泛型的写法。

```
public class GenericTest {
```

```
    public static void main(String[] args) {
        /*
        List list = new ArrayList();
        list.add("qqyumidi");
        list.add("corn");
        list.add(100);
        */

        List<String> list = new ArrayList<String>();
        list.add("qqyumidi");
        list.add("corn");
        //list.add(100);   // 1  提示编译错误

        for (int i = 0; i < list.size(); i++) {
            String name = list.get(i); // 2
            System.out.println("name:" + name);
        }
    }
}
```

采用泛型写法后，在//1处想加入一个Integer类型的对象时会出现编译错误，通过List，直接限定了list集合中只能含有String类型的元素，从而在//2处无须进行强制类型转换，因为此时，集合能够记住元素的类型信息，编译器已经能够确认它是String类型了。

结合上面的泛型定义，我们知道在List中，String是类型实参，也就是说，相应的List接口中肯定含有类型形参。且get()方法的返回结果也直接是此形参类型（也就是对应的传入的类型实参）。下面就来看看List接口的的具体定义：

```
public interface List<E> extends Collection<E> {

    int size();

    boolean isEmpty();

    boolean contains(Object o);

    Iterator<E> iterator();

    Object[] toArray();

    <T> T[] toArray(T[] a);

    boolean add(E e);

    boolean remove(Object o);

    boolean containsAll(Collection<?> c);

    boolean addAll(Collection<? extends E> c);
```

```
    boolean addAll(int index, Collection<? extends E> c);

    boolean removeAll(Collection<?> c);

    boolean retainAll(Collection<?> c);

    void clear();

    boolean equals(Object o);

    int hashCode();

    E get(int index);

    E set(int index, E element);

    void add(int index, E element);

    E remove(int index);

    int indexOf(Object o);

    int lastIndexOf(Object o);

    ListIterator<E> listIterator();

    ListIterator<E> listIterator(int index);

    List<E> subList(int fromIndex, int toIndex);
}
```

我们可以看到，在List接口中采用泛型化定义之后，中的E表示类型形参，可以接收具体的类型实参，并且此接口定义中，凡是出现E的地方均表示相同的接受自外部的类型实参。
自然的，ArrayList作为List接口的实现类，其定义形式是：

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
 {

    public boolean add(E e) {
        ensureCapacityInternal(size + 1);  // Increments modCount!!
        elementData[size++] = e;
        return true;
    }

    public E get(int index) {
        rangeCheck(index);
        checkForComodification();
        return ArrayList.this.elementData(offset + index);
```

```
    }

    //...省略掉其他具体的定义过程

}
```

由此，我们从源代码角度明白了为什么//1处加入Integer类型对象编译错误，且//2处get()到的类型直接就是String类型了。

三.自定义泛型接口、泛型类和泛型方法

从上面的内容中，大家已经明白了泛型的具体运作过程。也知道了接口、类和方法也都可以使用泛型去定义，以及相应的使用。是的，在具体使用时，可以分为泛型接口、泛型类和泛型方法。
自定义泛型接口、泛型类和泛型方法与上述Java源码中的List、ArrayList类似。如下，我们看一个最简单的泛型类和方法定义：

```
public class GenericTest {

    public static void main(String[] args) {

        Box<String> name = new Box<String>("corn");
        System.out.println("name:" + name.getData());
    }

}

class Box<T> {

    private T data;

    public Box() {

    }

    public Box(T data) {
        this.data = data;
    }

    public T getData() {
        return data;
    }

}
```

在泛型接口、泛型类和泛型方法的定义过程中，我们常见的如T、E、K、V等形式的参数常用于表示泛型形参，由于接收来自外部使用时候传入的类型实参。那么对于不同传入的类型实参，生成的相应对象实例的类型是不是一样的呢？

```
public class GenericTest {
```

```
    public static void main(String[] args) {

        Box<String> name = new Box<String>("corn");
        Box<Integer> age = new Box<Integer>(712);

        System.out.println("name class:" + name.getClass());      // com.
qqyumidi.Box
        System.out.println("age class:" + age.getClass());        // com.
qqyumidi.Box
        System.out.println(name.getClass() == age.getClass());    // true

    }

}
```

由此，我们发现，在使用泛型类时，虽然传入了不同的泛型实参，但并没有真正意义上生成不同的类型，传入不同泛型实参的泛型类在内存上只有一个，即还是原来的最基本的类型（本实例中为Box），当然，在逻辑上我们可以理解成多个不同的泛型类型。
究其原因，在于Java中的泛型这一概念提出的目的，导致其只是作用于代码编译阶段，在编译过程中，对于正确检验泛型结果后，会将泛型的相关信息擦出，也就是说，成功编译过后的class文件中是不包含任何泛型信息的。泛型信息不会进入到运行时阶段。
对此总结成一句话：泛型类型在逻辑上看以看成是多个不同的类型，实际上都是相同的基本类型。

四.类型通配符

接着上面的结论，我们知道，Box和Box实际上都是Box类型，现在需要继续探讨一个问题，那么在逻辑上，类似于Box和Box是否可以看成具有父子关系的泛型类型呢？
为了弄清这个问题，我们继续看下下面这个例子:

```
public class GenericTest {

    public static void main(String[] args) {

        Box<Number> name = new Box<Number>(99);
        Box<Integer> age = new Box<Integer>(712);

        getData(name);

        //The method getData(Box<Number>) in the type GenericTest is
        //not applicable for the arguments (Box<Integer>)
        getData(age);   // 1

    }

    public static void getData(Box<Number> data){
        System.out.println("data :" + data.getData());
    }

}
```

我们发现，在代码//1处出现了错误提示信息：The method getData(Box) in the t ype GenericTest is not applicable for the arguments (Box)。显然，通过提示信息，我们知道Box在逻辑上不能视为Box的父类。那么，原因何在呢？

```
public class GenericTest {

    public static void main(String[] args) {

        Box<Integer> a = new Box<Integer>(712);
        Box<Number> b = a;   // 1
        Box<Float> f = new Box<Float>(3.14f);
        b.setData(f);         // 2

    }

    public static void getData(Box<Number> data) {
        System.out.println("data :" + data.getData());
    }

}

class Box<T> {

    private T data;

    public Box() {

    }

    public Box(T data) {
        setData(data);
    }

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }

}
```

这个例子中，显然//1和//2处肯定会出现错误提示的。在此我们可以使用反证法来进行说明。

假设Box在逻辑上可以视为Box的父类，那么//1和//2处将不会有错误提示了，那么问题就出来了，通过getData()方法取出数据时到底是什么类型呢？Integer? Float? 还是Number？且由于在编程过程中的顺序不可控性，导致在必要的时候必须要进行类型判断，且进行强制类型转换。显然，这与泛型的理念矛盾，因此，在逻辑上Box不能视为Box的父类。

好，那我们回过头来继续看“类型通配符”中的第一个例子，我们知道其具体的错误提示的深层次原因了。那么如何解决呢？总部能再定义一个新的函数吧。这和Java中的多态理念显然是违背的，因此，我们需要一个在逻辑上可以用来表示同时是Box和Box的父类的一个引用类型，由此，类型通配符应运而生。

类型通配符一般是使用？代替具体的类型实参。注意了，此处是类型实参，而不是类型形参！且Box<?>在逻辑上是Box、Box…等所有Box<具体类型实参>的父类。由此，我们依然可以定义泛型方法，来完成此类需求。

```
public class GenericTest {

    public static void main(String[] args) {

        Box<String> name = new Box<String>("corn");
        Box<Integer> age = new Box<Integer>(712);
        Box<Number> number = new Box<Number>(314);

        getData(name);
        getData(age);
        getData(number);
    }

    public static void getData(Box<?> data) {
        System.out.println("data :" + data.getData());
    }

}
```

有时候，我们还可能听到类型通配符上限和类型通配符下限。具体有是怎么样的呢？
在上面的例子中，如果需要定义一个功能类似于getData()的方法，但对类型实参又有进一步的限制：只能是Number类及其子类。此时，需要用到类型通配符上限。

```
public class GenericTest {

    public static void main(String[] args) {

        Box<String> name = new Box<String>("corn");
        Box<Integer> age = new Box<Integer>(712);
        Box<Number> number = new Box<Number>(314);

        getData(name);
        getData(age);
        getData(number);

        //getUpperNumberData(name); // 1
        getUpperNumberData(age);    // 2
        getUpperNumberData(number); // 3
    }
```

```
    public static void getData(Box<?> data) {
        System.out.println("data :" + data.getData());
    }

    public static void getUpperNumberData(Box<? extends Number> data){
        System.out.println("data :" + data.getData());
    }

}
```

此时，显然，在代码//1处调用将出现错误提示，而//2 //3处调用正常。

类型通配符上限通过形如Box<? extends Number>形式定义，相对应的，类型通配符下限为Box<? super Number>形式，其含义与类型通配符上限正好相反，在此不作过多阐述了。

五.话外篇

本文中的例子主要是为了阐述泛型中的一些思想而简单举出的，并不一定有着实际的可用性。另外，一提到泛型，相信大家用到最多的就是在集合中，其实，在实际的编程过程中，自己可以使用泛型去简化开发，且能很好的保证代码质量。并且还要注意的一点是，Java中没有所谓的泛型数组一说。
对于泛型，最主要的还是需要理解其背后的思想和目的。

Java泛型详解

1. 概述

在引入范型之前，Java类型分为原始类型、复杂类型，其中复杂类型分为数组和类。引入范型后，一个复杂类型
就可以在细分成更多的类型。
例如原先的类型List，现在在细分成List, List等更多的类型。
注意，现在List, List是两种不同的类型，
他们之间没有继承关系，即使String继承了Object。下面的代码是非法的

```
    List<String> ls = new ArrayList<String>();
    List<Object> lo = ls;
```

这样设计的原因在于，根据lo的声明，编译器允许你向lo中添加任意对象（例如Integer），但是此对象是List，破坏了数据类型的完整性。
在引入范型之前，要在类中的方法支持多个数据类型，就需要对方法进行重载，在引入范型后，可以解决此问题（多态），更进一步可以定义多个参数以及返回值之间的关系。
例如

```
public void write(Integer i, Integer[] ia);
public void write(Double  d, Double[] da);
```

的范型版本为

```
public <T> void write(T t, T[] ta);
```

2. 定义&使用

类型参数的命名风格为：
推荐你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，这使它和其他的普通的形式参数很容易被区分开来。使用T代表类型，无论何时都没有比这更具体的类型来区分它。这经常见于泛型方法。如果有多个类型参数，我们可能使用字母表中T的临近的字母，比如S。如果一个泛型函数在一个泛型类里面出现，最好避免在方法的类型参数和类的类型参数中使用同样的名字来避免混 淆。对内部类也是同样。

2.1 定义带类型参数的类

在定义带类型参数的类时，在紧跟类命之后的<>内,指定一个或多个类型参数的名字，同时也可以对类型参数的取值范围进行限定，多个类型参数之间用,号分隔。
定义完类型参数后，可以在定义位置之后的类的几乎任意地方（静态块，静态属性，静态方法除外）使用类型参数， 就像使用普通的类型一样。

注意: 父类定义的类型参数不能被子类继承。

```
public class TestClassDefine<T, S extends T> {
    ....
}
```

2.2 定义带类型参数方法

在定义带类型参数的方法时，在紧跟可见范围修饰（例如public）之后的<>内,指定一个或多个类型参数的名字， 同时也可以对类型参数的取值范围进行限定，多个类型参数之间用,号分隔。 定义完类型参数后，可以在定义位置之后的方法的任意地方使用类型参数，就像使用普通的类型一样。
例如：

```
public <T, S extends T> T testGenericMethodDefine(T t, S s){
    ...
}
```

注意： 定义带类型参数的方法，其主要目的是为了表达多个参数以及返回值之间的关系。例如本例子中T和S的继承关系， 返回值的类型和第一个类型参数的值相同。
如果仅仅是想实现多态，请优先使用通配符解决。通配符的使用稍后介绍。

```
public <T> void testGenericMethodDefine2(List<T> s){
    ...
}
```

应改为

```
public void testGenericMethodDefine2(List<? > s){
    ...
}
```

3. 类型参数赋值

当对类或方法的类型参数进行赋值时，要求对所有的类型参数进行赋值。否则，将得到一个编译错误。

3.1 对带类型参数的类进行类型参数赋值

对带类型参数的类进行类型参数赋值有两种方式，
第一声明类变量或者实例化时。例如

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

第二继承类或者实现接口时。例如

```
public class MyList<E> extends ArrayList<E> implements List<E> {...}
```

3.2 对带类型参数方法进行赋值

当调用范型方法时，编译器自动对类型参数进行赋值，当不能成功赋值时报编译错误。例如

```
public <T> T testGenericMethodDefine3(T t, List<T> list){
    ...
}
```

```
public <T> T testGenericMethodDefine4(List<T> list1, List<T> list2){
    ...
}
```

```
Number n = null;
Integer i = null;
Object o = null;
testGenericMethodDefine(n, i);//此时T为Number, S为Integer
testGenericMethodDefine(o, i);//T为Object, S为Integer

List<Number> list1 = null;
testGenericMethodDefine3(i, list1)//此时T为Number

List<Integer> list2 = null;
testGenericMethodDefine4(list1, list2)//编译报错
```

3.3 通配符

在上面两小节中，对是类型参数赋予具体的值，除此，还可以对类型参数赋予不确定值。例如

```
List<?> unknownList;
List<? extends Number> unknownNumberList;
List<? super Integer> unknownBaseLineIntgerList;
```

注意： 在Java集合框架中，对于参数值是未知类型的容器类，只能读取其中元素，不能像其中添加元素，
因为，其类型是未知，所以编译器无法识别添加元素的类型和容器的类型是否兼容，唯一的例外是NULL

```
List<String> listString;
List<?> unknownList2 = listString;
unknownList = unknownList2;
listString = unknownList;//编译错误
```

4. 数组范型

可以使用带范型参数值的类声明数组，却不可有创建数组

```
List<Integer>[] iListArray;
new ArrayList<Integer>[10];//编译时错误
```

5. 实现原理

5.1. Java范型时编译时技术，在运行时不包含范型信息，仅仅Class的实例中包含了类型参数的定义信息。

泛型是通过java编译器的称为擦除(erasure)的前端处理来实现的。你可以（基本上就是）把它认为是一个从源码到源码的转换，它把泛型版本转换成非泛型版本。基本上，擦除去掉了所有的泛型类型信息。所有在尖括号之间的类型信息都被扔掉了，因此，比如说一个List类型被转换为List。所有对类型变量的引用被替换成类型变量的上限(通常是Object)。而且，无论何时结果代码类型不正确，会插入一个到合适类型的转换。

```
 <T> T badCast(T t, Object o) {
    return (T) o; // unchecked warning
 }
```

类型参数在运行时并不存在。这意味着它们不会添加任何的时间或者空间上的负担，这很好。不幸的是，这也意味着你不能依靠他们进行类型转换。

5.2.一个泛型类被其所有调用共享

下面的代码打印的结果是什么？

```
List<String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass());
```

或许你会说false，但是你想错了。它打印出true。因为一个泛型类的所有实例在运行时具有相同的运行时类(class)，而不管他们的实际类型参数。事实上，泛型之所以叫泛型，就是因为它对所有其可能的类型参数，有同样的行为；同样的类可以被当作许多不同的类型。作为一个结果，类的静态变量和方法也在所有的实例间共享。这就是为什么在静态方法或静态初始化代码中或者在静态变量的声明和初始化时使用类型参数（类型参数是属于具体实例的）是不合法的原因。

5.3. 转型和instanceof

泛型类被所有其实例(instances)共享的另一个暗示是检查一个实例是不是一个特定类型的泛型类是没有意义的。

```
Collection cs = new ArrayList<String>();
if (cs instanceof Collection<String>) { ...} // 非法
```

类似的，如下的类型转换

```
Collection<String> cstr = (Collection<String>) cs;
```

得到一个unchecked warning，因为运行时环境不会为你作这样的检查。

6. Class的范型处理

Java 5之后，Class变成范型化了。
JDK1.5中一个变化是类 java.lang.Class是泛型化的。这是把泛型扩展到容器类之外的一个很有意思的例子。
现在，Class有一个类型参数T, 你很可能会问，T 代表什么？它代表Class对象代表的类型。比如说，
String.class类型代表 Class，Serializable.class代表 Class。
这可以被用来提高你的反射代码的类型安全。
特别的，因为 Class的 newInstance() 方法现在返回一个T, 你可以在使用反射创建对象时得到更精确的类型。
比如说，假定你要写一个工具方法来进行一个数据库查询，给定一个SQL语句，并返回一个数据库中符合查询条件
的对象集合(collection)。
一个方法是显式的传递一个工厂对象，像下面的代码：

```
interface Factory<T> {
      public T make();
}
public <T> Collection<T> select(Factory<T> factory, String statement) {
       Collection<T> result = new ArrayList<T>();
       /* run sql query using jdbc */
```

```
        for ( int i=0; i<10; i++ ) { /* iterate over jdbc results */
             T item = factory.make();
             /* use reflection and set all of item's fields from sql resul
ts */
             result.add( item );
        }
        return result;
}
```

你可以这样调用：

```
select(new Factory<EmpInfo>(){
    public EmpInfo make() {
         return new EmpInfo();
         }
        } , "selection string");
```

也可以声明一个类 EmpInfoFactory 来支持接口 Factory：

```
class EmpInfoFactory implements Factory<EmpInfo> { ...
    public EmpInfo make() { return new EmpInfo();}
}
```

然后调用：

```
select(getMyEmpInfoFactory(), "selection string");
```

这个解决方案的缺点是它需要下面的二者之一：
调用处那冗长的匿名工厂类，或为每个要使用的类型声明一个工厂类并传递其对象给调用的地方这很不自然。使用class类型参数值是非常自然的，它可以被反射使用。没有泛型的代码可能是：

```
Collection emps = sqlUtility.select(EmpInfo.class, "select * from emps");
 ...
public static Collection select(Class c, String sqlStatement) {
    Collection result = new ArrayList();
    /* run sql query using jdbc */
    for ( /* iterate over jdbc results */ ) {
        Object item = c.newInstance();
        /* use reflection and set all of item's fields from sql results
 */
        result.add(item);
    }
        return result;
}
```

但是这不能给我们返回一个我们要的精确类型的集合。现在Class是泛型的，我们可以写：

```
Collection<EmpInfo> emps=sqlUtility.select(EmpInfo.class, ”select * from
emps”); ...
public static <T> Collection<T> select(Class<T>c, String sqlStatement) {
    Collection<T> result = new ArrayList<T>();
    /* run sql query using jdbc */
    for ( /* iterate over jdbc results */ ) {
        T item = c.newInstance();
        /* use reflection and set all of item’s fields from sql results
 */
        result.add(item);
    }
    return result;
}
```

来通过一种类型安全的方式得到我们要的集合。
这项技术是一个非常有用的技巧，它已成为一个在处理注释(annotations)的新API中被广泛使用的习惯用法。

7. 新老代码兼容

7.1. 为了保证代码的兼容性，下面的代码编译器（javac）允许，类型安全有你自己保证

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

7.2. 在将你的类库升级为范型版本时，慎用协变式返回值。
例如，将代码

```
public class Foo {
    public Foo create(){
        return new Foo();
    }
}

public class Bar extends Foo {
    public Foo create(){
        return new Bar();
    }
}
```

采用协变式返回值风格，将Bar修改为

```
public class Bar extends Foo {
    public Bar create(){
        return new Bar();
    }
}
```

要小心你类库的客户端。

8. 泛型的使用汇总

泛型有三种使用方式，分别为：泛型类、泛型接口、泛型方法

8.1 泛型类

泛型类型用于类的定义中，被称为泛型类。通过泛型可以完成对一组类的操作对外开放相同的接口。最典型的就是各种容器类，如：List、Set、Map。
泛型类的最基本写法（这么看可能会有点晕，会在下面的例子中详解）：

```
class 类名称 <泛型标识：可以随便写任意标识号，标识指定的泛型的类型>{
  private 泛型标识 /*（成员变量类型）*/ var;
  .....

  }
}
```

一个最普通的泛型类：

```
//此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型
//在实例化泛型类时，必须指定T的具体类型
public class Generic<T>{
    //key这个成员变量的类型为T,T的类型由外部指定
    private T key;

    public Generic(T key) { //泛型构造方法形参key的类型也为T，T的类型由外部指定
        this.key = key;
    }

    public T getKey(){ //泛型方法getKey的返回值类型为T，T的类型由外部指定
        return key;
    }
}
```

```
//泛型的类型参数只能是类类型（包括自定义类），不能是简单类型
//传入的实参类型需与泛型的类型参数类型相同，即为Integer.
Generic<Integer> genericInteger = new Generic<Integer>(123456);

//传入的实参类型需与泛型的类型参数类型相同，即为String.
Generic<String> genericString = new Generic<String>("key_vlaue");
Log.d("泛型测试","key is " + genericInteger.getKey());
Log.d("泛型测试","key is " + genericString.getKey());
```

定义的泛型类，就一定要传入泛型类型实参么？并不是这样，在使用泛型的时候如果传入泛型实参，则会根据传入的泛型实参做相应的限制，此时泛型才会起到本应起到的限制作用。如果不传入

泛型类型实参的话，在泛型类中使用泛型的方法或成员变量定义的类型可以为任何的类型。
看一个例子：

```
Generic generic = new Generic("111111");
Generic generic1 = new Generic(4444);
Generic generic2 = new Generic(55.55);
Generic generic3 = new Generic(false);

Log.d("泛型测试","key is " + generic.getKey());
Log.d("泛型测试","key is " + generic1.getKey());
Log.d("泛型测试","key is " + generic2.getKey());
Log.d("泛型测试","key is " + generic3.getKey());
```

> **注意：**
> 泛型的类型参数只能是类类型，不能是简单类型。
> 不能对确切的泛型类型使用instanceof操作。如下面的操作是非法的，编译时会出错。

```
if(ex_num instanceof Generic<Number>){
}
```

8.2 泛型接口

泛型接口与泛型类的定义及使用基本相同。泛型接口常被用在各种类的生产器中，可以看一个例子：

```
//定义一个泛型接口
public interface Generator<T> {
    public T next();
}
```

当实现泛型接口的类，未传入泛型实参时：

```
/**
 * 未传入泛型实参时，与泛型类的定义相同，在声明类的时候，需将泛型的声明也一起加到类中
 * 即：class FruitGenerator<T> implements Generator<T>{
 * 如果不声明泛型，如：class FruitGenerator implements Generator<T>，编译器会报错："Unknown class"
 */
class FruitGenerator<T> implements Generator<T>{
    @Override
    public T next() {
        return null;
    }
}
```

当实现泛型接口的类，传入泛型实参时：

```
/**
 * 传入泛型实参时：
 * 定义一个生产器实现这个接口,虽然我们只创建了一个泛型接口Generator<T>
 * 但是我们可以为T传入无数个实参，形成无数种类型的Generator接口。
 * 在实现类实现泛型接口时，如已将泛型类型传入实参类型，则所有使用泛型的地方都要替换成传入的实参类型
 * 即：Generator<T>，public T next();中的的T都要替换成传入的String类型。
 */
public class FruitGenerator implements Generator<String> {

    private String[] fruits = new String[]{"Apple", "Banana", "Pear"};

    @Override
    public String next() {
        Random rand = new Random();
        return fruits[rand.nextInt(3)];
    }
}
```

8.3 泛型通配符

我们知道Ingeter是Number的一个子类，同时在特性章节中我们也验证过Generic与Generic实际上是相同的一种基本类型。那么问题来了，在使用Generic作为形参的方法中，能否使用Generic的实例传入呢？在逻辑上类似于Generic和Generic是否可以看成具有父子关系的泛型类型呢？
为了弄清楚这个问题，我们使用Generic这个泛型类继续看下面的例子：

```
public void showKeyValue1(Generic<Number> obj){
    Log.d("泛型测试","key value is " + obj.getKey());
}
```

```
Generic<Integer> gInteger = new Generic<Integer>(123);
Generic<Number> gNumber = new Generic<Number>(456);

showKeyValue(gNumber);

// showKeyValue这个方法编译器会为我们报错：Generic<java.lang.Integer>
// cannot be applied to Generic<java.lang.Number>
// showKeyValue(gInteger);
```

通过提示信息我们可以看到Generic不能被看作为`Generic的子类。由此可以看出:同一种泛型可以对应多个版本（因为参数类型是不确定的），不同版本的泛型类实例是不兼容的。

回到上面的例子，如何解决上面的问题？总不能为了定义一个新的方法来处理Generic类型的类，这显然与java中的多台理念相违背。因此我们需要一个在逻辑上可以表示同时是Generic和Generic父类的引用类型。由此类型通配符应运而生。
我们可以将上面的方法改一下：

```
public void showKeyValue1(Generic<?> obj){
    Log.d("泛型测试","key value is " + obj.getKey());
}
```

类型通配符一般是使用？代替具体的类型实参，注意了，此处’？’是类型实参，而不是类型形参 。重要说三遍！此处’？’是类型实参，而不是类型形参 ！ 此处’？’是类型实参，而不是类型形参 ！再直白点的意思就是，此处的？和Number、String、Integer一样都是一种实际的类型，可以把？看成所有类型的父类。是一种真实的类型。可以解决当具体类型不确定的时候，这个通配符就是？；当操作类型时，不需要使用类型的具体功能时，只使用Object类中的功能。那么可以用？通配符来表未知类型。

8.4 泛型方法

在java中,泛型类的定义非常简单，但是泛型方法就比较复杂了。

> 尤其是我们见到的大多数泛型类中的成员方法也都使用了泛型，有的甚至泛型类中也包含着泛型方法，这样在初学者中非常容易将泛型方法理解错了。

泛型类，是在实例化类的时候指明泛型的具体类型；泛型方法，是在调用方法的时候指明泛型的具体类型 。

```
/**
 * 泛型方法的基本介绍
 * @param tClass 传入的泛型实参
 * @return T 返回值为T类型
 * 说明：
 *      1）public 与 返回值中间<T>非常重要，可以理解为声明此方法为泛型方法。
 *      2）只有声明了<T>的方法才是泛型方法，泛型类中的使用了泛型的成员方法并不是泛型方法。
 *      3）<T>表明该方法将使用泛型类型T，此时才可以在方法中使用泛型类型T。
 *      4）与泛型类的定义一样，此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型。
 */
public <T> T genericMethod(Class<T> tClass)throws InstantiationException
 ,
  IllegalAccessException{
        T instance = tClass.newInstance();
        return instance;
}
```

```
Object obj = genericMethod(Class.forName("com.test.test"));
```

8.4..1 泛型方法的基本用法

光看上面的例子有的同学可能依然会非常迷糊，我们再通过一个例子，把我泛型方法再总结一下。

```
public class GenericTest {
    //这个类是个泛型类，在上面已经介绍过
    public class Generic<T>{
         private T key;

         public Generic(T key) {
             this.key = key;
         }

         //我想说的其实是这个，虽然在方法中使用了泛型，但是这并不是一个泛型方法。
         //这只是类中一个普通的成员方法，只不过他的返回值是在声明泛型类已经声明过的泛型。
         //所以在这个方法中才可以继续使用 T 这个泛型。
         public T getKey(){
             return key;
         }

         /**
          * 这个方法显然是有问题的，在编译器会给我们提示这样的错误信息"cannot reslove symbol E"
          * 因为在类的声明中并未声明泛型E，所以在使用E做形参和返回值类型时，编译器会无法识别。
         public E setKey(E key){
              this.key = keu
         }
         */
    }

    /**
     * 这才是一个真正的泛型方法。
     * 首先在public与返回值之间的<T>必不可少，这表明这是一个泛型方法，并且声明了一个泛型T
     * 这个T可以出现在这个泛型方法的任意位置.
     * 泛型的数量也可以为任意多个
     *    如: public <T,K> K showKeyName(Generic<T> container){
     *         ...
     *         }
     */
    public <T> T showKeyName(Generic<T> container){
        System.out.println("container key :" + container.getKey());
        //当然这个例子举的不太合适，只是为了说明泛型方法的特性。
        T test = container.getKey();
        return test;
    }

    //这也不是一个泛型方法，这就是一个普通的方法，只是使用了Generic<Number>这个泛型类做形参而已。
    public void showKeyValue1(Generic<Number> obj){
        Log.d("泛型测试","key value is " + obj.getKey());
    }
```

```
    //这也不是一个泛型方法，这也是一个普通的方法，只不过使用了泛型通配符?
    //同时这也印证了泛型通配符章节所描述的，?是一种类型实参，可以看做为Number等所有类
的父类
    public void showKeyValue2(Generic<?> obj){
        Log.d("泛型测试","key value is " + obj.getKey());
    }

     /**
     * 这个方法是有问题的，编译器会为我们提示错误信息："UnKnown class 'E' "
     * 虽然我们声明了<T>,也表明了这是一个可以处理泛型的类型的泛型方法。
     * 但是只声明了泛型类型T，并未声明泛型类型E，因此编译器并不知道该如何处理E这个类
型。
    public <T> T showKeyName(Generic<E> container){
        ...
    }
    */

    /**
     * 这个方法也是有问题的，编译器会为我们提示错误信息："UnKnown class 'T' "
     * 对于编译器来说T这个类型并未项目中声明过，因此编译也不知道该如何编译这个类。
     * 所以这也不是一个正确的泛型方法声明。
    public void showkey(T genericObj){

    }
    */

    public static void main(String[] args) {
    }
```

8.4.2 类中的泛型方法

当然这并不是泛型方法的全部，泛型方法可以出现杂任何地方和任何场景中使用。但是有一种情况是非常特殊的，当泛型方法出现在泛型类中时，我们再通过一个例子看一下

```
public class GenericFruit {
    class Fruit{
        @Override
        public String toString() {
            return "fruit";
        }
    }

    class Apple extends Fruit{
        @Override
        public String toString() {
            return "apple";
        }
    }

    class Person{
        @Override
```

```
        public String toString() {
            return "Person";
        }
    }

    class GenerateTest<T>{
        public void show_1(T t){
            System.out.println(t.toString());
        }

        //在泛型类中声明了一个泛型方法，使用泛型E，这种泛型E可以为任意类型。可以类型与T相同，也可以不同。
        //由于泛型方法在声明的时候会声明泛型<E>，因此即使在泛型类中并未声明泛型，编译器也能够正确识别泛型方法中识别的泛型。
        public <E> void show_3(E t){
            System.out.println(t.toString());
        }

        //在泛型类中声明了一个泛型方法，使用泛型T，注意这个T是一种全新的类型，可以与泛型类中声明的T不是同一种类型。
        public <T> void show_2(T t){
            System.out.println(t.toString());
        }
    }

    public static void main(String[] args) {
        Apple apple = new Apple();
        Person person = new Person();

        GenerateTest<Fruit> generateTest = new GenerateTest<Fruit>();
        //apple是Fruit的子类，所以这里可以
        generateTest.show_1(apple);
        //编译器会报错，因为泛型类型实参指定的是Fruit，而传入的实参类是Person
        //generateTest.show_1(person);

        //使用这两个方法都可以成功
        generateTest.show_2(apple);
        generateTest.show_2(person);

        //使用这两个方法也都可以成功
        generateTest.show_3(apple);
        generateTest.show_3(person);
    }
}
```

8.4.3 泛型方法与可变参数

再看一个泛型方法和可变参数的例子：

```
public <T> void printMsg( T... args){
    for(T t : args){
```

```
        Log.d("泛型测试","t is " + t);
    }
}
/**
printMsg("111",222,"aaaa","2323.4",55.55);
*/
```

8.4.4 静态方法与泛型

静态方法有一种情况需要注意一下，那就是在类中的静态方法使用泛型：静态方法无法访问类上定义的泛型；如果静态方法操作的引用数据类型不确定的时候，必须要将泛型定义在方法上。
即：如果静态方法要使用泛型的话，必须将静态方法也定义成泛型方法 。

```
public class StaticGenerator<T> {
    ....
    ....
    /**
     * 如果在类中定义使用泛型的静态方法，需要添加额外的泛型声明（将这个方法定义成泛型方法）
     * 即使静态方法要使用泛型类中已经声明过的泛型也不可以。
     * 如：public static void show(T t){..},此时编译器会提示错误信息：
          "StaticGenerator cannot be refrenced from static context"
     */
    public static <T> void show(T t){

    }
}
```

8.4.6 泛型方法总结

泛型方法能使方法独立于类而产生变化，以下是一个基本的指导原则：

> 无论何时，如果你能做到，你就该尽量使用泛型方法。也就是说，如果使用泛型方法将整个类泛型化，那么就应该使用泛型方法。另外对于一个static的方法而已，无法访问泛型类型的参数。所以如果static方法要使用泛型能力，就必须使其成为泛型方法。

8.5 泛型上下边界

在使用泛型的时候，我们还可以为传入的泛型类型实参进行上下边界的限制，如：类型实参只准传入某种类型的父类或某种类型的子类。
为泛型添加上边界，即传入的类型实参必须是指定类型的子类型

```
public void showKeyValue1(Generic<? extends Number> obj){
    Log.d("泛型测试","key value is " + obj.getKey());
}
```

```
Generic<String> generic1 = new Generic<String>("11111");
```

```
Generic<Integer> generic2 = new Generic<Integer>(2222);
Generic<Float> generic3 = new Generic<Float>(2.4f);
Generic<Double> generic4 = new Generic<Double>(2.56);

//这一行代码编译器会提示错误，因为String类型并不是Number类型的子类
//showKeyValue1(generic1);

showKeyValue1(generic2);
showKeyValue1(generic3);
showKeyValue1(generic4);
```

如果我们把泛型类的定义也改一下:

```
public class Generic<T extends Number>{
    private T key;

    public Generic(T key) {
        this.key = key;
    }

    public T getKey(){
        return key;
    }
}
```

```
//这一行代码也会报错，因为String不是Number的子类
Generic<String> generic1 = new Generic<String>("11111");
```

再来一个泛型方法的例子：

```
//在泛型方法中添加上下边界限制的时候，必须在权限声明与返回值之间的<T>上添加上下边界，即在泛型声明的时候添加
//public <T> T showKeyName(Generic<T extends Number> container)，编译器会报错："Unexpected bound"
public <T extends Number> T showKeyName(Generic<T> container){
    System.out.println("container key :" + container.getKey());
    T test = container.getKey();
    return test;
}
```

通过上面的两个例子可以看出：泛型的上下边界添加，必须与泛型的声明在一起 。

8.6 关于泛型数组

看到了很多文章中都会提起泛型数组，经过查看sun的说明文档，在java中是”不能创建一个确切的泛型类型的数组”的。
也就是说下面的这个例子是不可以的：

```
```

```
List<String>[] ls = new ArrayList<String>[10];
```

而使用通配符创建泛型数组是可以的，如下面这个例子：

```
List<?>[] ls = new ArrayList<?>[10];
```

这样也是可以的：

```
List<String>[] ls = new ArrayList[10];
```

下面使用Sun的一篇文档的一个例子来说明这个问题：

```
List<String>[] lsa = new List<String>[10]; // Not really allowed.
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // Unsound, but passes run time store check
String s = lsa[1].get(0); // Run-time error: ClassCastException.
```

> 这种情况下，由于JVM泛型的擦除机制，在运行时JVM是不知道泛型信息的，所以可以给oa[1]赋上一个ArrayList而不会出现异常，但是在取出数据的时候却要做一次类型转换，所以就会出现ClassCastException，如果可以进行泛型数组的声明，上面说的这种情况在编译期将不会出现任何的警告和错误，只有在运行时才会出错。
> 而对泛型数组的声明进行限制，对于这样的情况，可以在编译期提示代码有类型安全问题，比没有任何提示要强很多。

下面采用通配符的方式是被允许的:数组的类型不可以是类型变量，除非是采用通配符的方式，因为对于通配符的方式，最后取出数据是要做显式的类型转换的。

```
List<?>[] lsa = new List<?>[10]; // OK, array of unbounded wildcard type.

Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // Correct.
Integer i = (Integer) lsa[1].get(0); // OK
```