一个很典型的泛型(generic)代码。T是类型变量，可以是任何引用类型：

[复制代码](javascript:void(0);)

public class Pair<T>{

private T first=null;

private T second=null;

public Pair(T fir,T sec){

this.first=fir;

this.second=sec;

}

public T getFirst(){

return this.first;

}

public T getSecond(){

return this.second;

}

public void setFirst(T fir){

this.first=fir;

}

}

[复制代码](javascript:void(0);)

**1、Generic class 创建对象**

Pair<String> pair1=new Pair("string",1); ...①

Pair<String> pair2=new Pair<String>("string",1) ...②

有个很有趣的现象:  ①代码在编译期不会出错，②代码在编译期会检查出错误。   
这个问题其实很简单  
      (1) JVM本身并没有泛型对象这样的一个特殊概念。所有的泛型类对象在编译器会全部变成普通类对象（这一点会在下面详细阐述）。  
      比如①,②两个代码编译器全部调用的是 Pair(Object fir, Object sec)这样的构造器。

      因此代码①中的new Pair("string",1)在编译器是没有问题的，毕竟编译器并不知道你创建的Pair类型中具体是哪一个类型变量T，而且编译器肯定了String对象和Integer对象都属于Object类型的。

       但是一段运行pair1.getSecond()就会抛出ClassCastException异常。这是因为JVM会根据第一个参数"string"推算出T类型变量是String类型，这样getSecond也应该是返回String类型，然后编译器已经默认了second的操作数是一个值为1的Integer类型。当然就不符合JVM的运行要求了，不终止程序才怪。

       (2) 但代码②会在编译器报错，是因为new Pair<String>("string",1)已经指明了创建对象pair2的类型变量T应该是String的。所以在编译期编译器就知道错误出在第二个参数Integer了。

小结一下：   
       创建泛型对象的时候，一定要指出类型变量T的具体类型。争取让编译器检查出错误，而不是留给JVM运行的时候抛出异常。

**2、JVM如何理解泛型概念 —— 类型擦除**  
    事实上，JVM并不知道泛型，所有的泛型在编译阶段就已经被处理成了普通类和方法。   
    处理方法很简单，我们叫做类型变量T的**擦除(erased)**。  
    无论我们如何定义一个泛型类型，相应的都会有一个原始类型被自动提供。原始类型的名字就是擦除类型参数的泛型类型的名字。  
         如果泛型类型的类型变量没有**限定(<T>)**，那么我们就用Object作为原始类型；  
         如果有限定(<T extends XClass>)，我们就XClass作为原始类型；  
         如果有多个限定(<T extends XClass1&XClass2>)，我们就用第一个边界的类型变量XClass1类作为原始类型；   
    比如上面的Pair<T>例子，编译器会把它当成被Object原始类型替代的普通类来替代。

[复制代码](javascript:void(0);)

//编译阶段：类型变量的擦除

public class Pair{

private Object first=null;

private Object second=null;

public Pair(Object fir,Object sec){

this.first=fir;

this.second=sec;

}

public Object getFirst(){

return this.first;

}

public void setFirst(Object fir){

this.first=fir;

}

}

[复制代码](javascript:void(0);)

**3、泛型约束和局限性—— 类型擦除所带来的麻烦**

**(1)  继承泛型类型的多态麻烦。（—— 子类没有覆盖住父类的方法 ）**

     看看下面这个类SonPair

class SonPair extends Pair<String>{

public void setFirst(String fir){....}

}

 很明显，程序员的本意是想在SonPair类中覆盖父类Pair<String>的setFirst(T fir)这个方法。但事实上，SonPair中的setFirst(String fir)方法根本没有覆盖住Pair<String>中的这个方法。   
     原因很简单，Pair<String>在编译阶段已经被类型擦除为Pair了，它的setFirst方法变成了setFirst(Object fir)。 那么SonPair中setFirst(String)当然无法覆盖住父类的setFirst(Object)了。

这对于多态来说确实是个不小的麻烦，我们看看编译器是如何解决这个问题的。

编译器 会自动在 SonPair中生成一个桥方法(bridge method ) ：

public void setFirst(Object fir){

setFirst((String) fir)

}

这样,SonPair的桥方法确实能够覆盖泛型父类的setFirst(Object) 了。而且桥方法内部其实调用的是子类字节setFirst(String)方法。对于多态来说就没问题了。

**问题还没有完，多态中的方法覆盖是可以了，但是桥方法却带来了一个疑问：**

现在，假设 我们还想在 SonPair 中覆盖getFirst()方法呢？

class SonPair extends Pair<String>{

public String getFirst(){....}

}

 由于需要桥方法来覆盖父类中的getFirst，编译器会自动在SonPair中生成一个 public Object getFirst()桥方法。   
 但是，疑问来了，SonPair中出现了两个方法签名一样的方法(只是返回类型不同)：

①String getFirst() // 自己定义的方法

②Object getFirst() // 编译器生成的桥方法

      难道，编译器允许出现方法签名相同的多个方法存在于一个类中吗？

      事实上有一个知识点可能大家都不知道：  
      ① 方法签名 确实只有方法名+参数列表 。这毫无疑问！  
      ② 我们绝对不能编写出方法签名一样的多个方法 。如果这样写程序，编译器是不会放过的。这也毫无疑问！  
      ③ 最重要的一点是：JVM会用参数类型和返回类型来确定一个方法。 一旦编译器通过某种方式自己编译出方法签名一样的两个方法(只能编译器自己来创造这种奇迹，我们程序员却不能人为的编写这种代码)。JVM还是能够分清楚这些方法的，前提是需要返回类型不一样。

**(2) 泛型类型中的方法冲突**

//在上面代码中加入equals方法

public class Pair<T>{

public boolean equals(T value){

return (first.equals(value));

}

}

这样看似乎没有问题的代码连编译器都通过不了：

       【Error】    Name clash: The method equals(T) of type Pair<T> has the same erasure as equals(Object) of type Object but does not override it。

        编译器说你的方法与Object中的方法冲突了。这是为什么?

        开始我也不太明白这个问题，觉得好像编译器帮助我们使得equals(T)这样的方法覆盖上了Object中的equals(Object)。经过大家的讨论，我觉得应该这么解释这个问题？

        首先、我们都知道子类方法要覆盖，必须与父类方法具有相同的方法签名（方法名+参数列表）。而且必须保证子类的访问权限>=父类的访问权限。这是大家都知道的事实。

        然后、在上面的代码中，当编译器看到Pair<T>中的equals(T)方法时，第一反应当然是equals(T)没有覆盖住父类Object中的equals(Object)了。

        接着、编译器将泛型代码中的T用Object替代（擦除）。突然发现擦除以后equals(T)变成了equals(Object)，糟糕了，这个方法与Object类中的equals一样了。基于开始确定没有覆盖这样一个想法，编译器彻底的疯了(精神分裂)。然后得出两个结论：①坚持原来的思想：没有覆盖。但现在一样造成了方法冲突了。   ②写这程序的程序员疯了(哈哈)。

        再说了，拿Pair<T>对象和T对象比较equals，就像牛头对比马嘴，哈哈，逻辑上也不通呀。

**(3) 没有泛型数组一说**

Pair<String>[] stringPairs=new Pair<String>[10];

Pair<Integer>[] intPairs=new Pair<Integer>[10];

      这种写法编译器会指定一个Cannot create a generic array of Pair<String>的错误

      我们说过泛型擦除之后，Pair<String>[]会变成Pair[]，进而又可以转换为Object[];

      假设泛型数组存在，那么

Object[0]=stringPairs[0]; Ok

Object[1]=intPairs[0]; Ok

       这就麻烦了，理论上将Object[]可以存储所有Pair对象，但这些Pair对象是泛型对象，他们的类型变量都不一样，那么调用每一个Object[]数组元素的对象方法可能都会得到不同的记过，也许是个字符串，也许是整形，这对于JVM可是无法预料的。

      记住： 数组必须牢记它的元素类型，也就是所有的元素对象都必须一个样，泛型类型恰恰做不到这一点。即使Pair<String>,Pair<Integer>... 都是Pair类型的，但他们还是不一样。

**总结：泛型代码与JVM**  
**① 虚拟机中没有泛型，只有普通类和方法。  
    ② 在编译阶段，所有泛型类的类型参数都会被Object或者它们的限定边界来替换。(类型擦除)  
    ③ 在继承泛型类型的时候，桥方法的合成是为了避免类型变量擦除所带来的多态灾难。**