Java集合有个缺点—-当我们把一个对象“丢进”集合里后，集合就会“忘记”这个对象的数据类型，当再次取出该对象时，该对象的编译类型就变成了Object类型（其运行时类型没变）

List list = **new** ArrayList();

list.add("first");

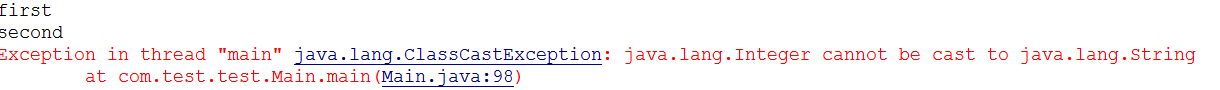
list.add("second");

list.add(10);

**for**(**int** i = 0, size = list.size(); i < size; i++){

System.*out*.println((String)list.get(i));

}







在上面的例子中，尖括号中的内容在java7之后是不需要的，但是在java7以前不能省略

**创建自己的泛型**

**public** **class** General<T> {

**private** T info;

**public** **void** setInfo(T info){

**this**.info = info;

}

**public** T getInfo(){

**return** info;

}

}

General<String> general1 = **new** General<String>();

general1.setInfo("test1");

System.*out*.println(general1.getInfo());

General<Integer> general2 = **new** General<Integer>();

general2.setInfo(20);

System.*out*.println(general2.getInfo());



**泛型类派生子类**

我们可以用如下的方式创建：







但是上面的第二三种方式，java编译器可能会发出警告：使用了未经检查或不安全的操作—-这就是泛型检查的警告

我们不能使用，当创建了带泛型声明的接口、父类之后，可以为该接口创建实现类，或从该父类派生子类，但需要注意的是，当使用这些接口、父类时不能再包含类型形参：



但我们可以使用：





**并不存在泛型类**

前面提到可以把ArrayList<String>类当成ArrayList的子类，事实上，ArrayList<String>类也确实像一种特殊的ArrayList类，这个ArrayList<String>对象只能添加String对象作为集合元素，但实际上，系统并没有为ArrayList<String>生成新的class文件，而且也不会把ArrayList<String>当做新类来处理。

General<String> general1 = **new** General<String>();

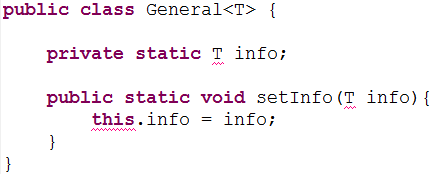
General<Integer> general2 = **new** General<Integer>();

System.*out*.println("general1.getClass() == general2.getClass() = " + (general1.getClass() == general2.getClass()));



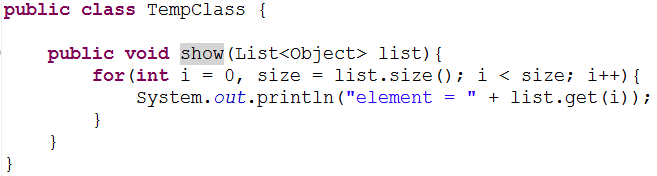
不管泛型的实际参数类型是什么，它们在运行时总有同样的类

不管为泛型的类型形参传入哪一种类型实参，对于java来说，它们依然被当成同一个类处理，在内存中也占用一块内存空间，因此静态方法、静态初始化块或者静态变量的声明和初始化中不允许使用类型形参。



**类型通配符**

注意：如果Foo是Bar的一个子类型（子类或者子几口），而G是具有泛型声明的类或接口。G<Foo>并不是G<Bar>的子类型！这一点是非常重要的，因为它与我们的习惯用法不同。例如下面的用法：





在上面的例子中就会发现List<String>类并不是List<Object>的子类

与数组进行对比，看看数组是如何工作的。在数组中，程序可以直接把一个Integer[]数组赋给一个Number[]变量，如果试图把一个Double对象保存到该Number[]数组中，编译可以通过，但在运行时抛出ArrayStoreException异常。





一门设计优秀的语言，不仅需要提供强大的功能，而且能提供强大的“错误提示”和“出错警告”，这样才能尽量避免开发者犯错，而java允许Integer[]数组赋值给Number[]变量显然不是一种安全的设计。

在java的早期设计中，允许Integer[]数组给Number[]变量存在缺陷，因此java在泛型设计的时候进行改进，它不再允许把List<Integer>对象赋值给List<Number>变量。

Java泛型设计的原则是，只要代码在编译时没有出现警告，就不会遇到运行时候ClassCastException异常

注意：数组和发型有所不同，假设Foo是Bar的一个子类型，那么Foo[]依然是Bar[]的子类型；但是G<Foo>不是G<Bar>的子类型。

为了表示各种泛型List的父类，我们需要使用类型通配符，类型通配符是一个问号（？），将一个问号作为类型实参传给List集合，写作：list<?>（意思是未知类型元素的List），但这种带通配符的List仅表示它是各种泛型List的父类，并不能把元素加入到其中，例如下面的代码将会引起编译错误：

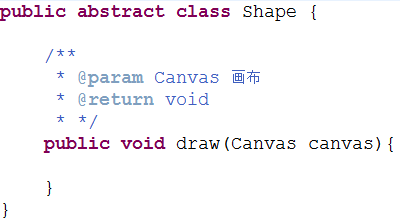


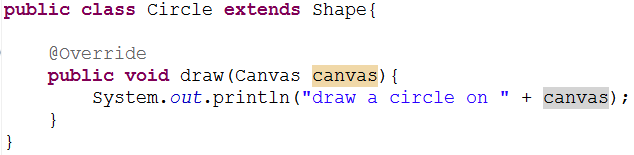
因为我们不知道上面程序中list集合里的元素类型，所以不能向其中添加对象。唯一的例外是null，它是所有引用类型的实例

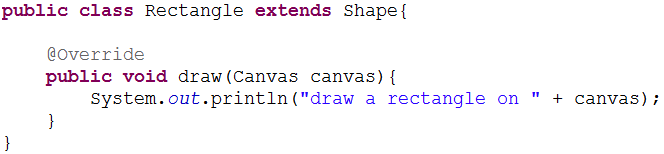
另一方面，程序可以调用get()方法来返回List<?>集合指定索引出的元素，其返回值是一个未知类型，但可以肯定的是，它总是一个Object，因此，把get()的返回值赋值给一个Object类型的变量，或者放在任何希望是Object类型的地方都可以。

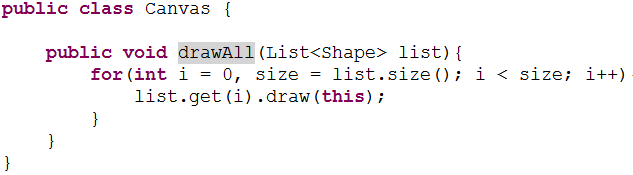
**设定类型通配符的上限**

也就是说我们只想让List<?>表示某一类泛型List的父类。看看下面的例子：





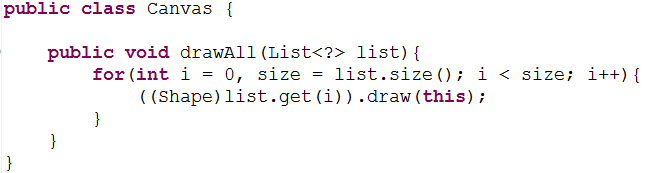




注意如果我们调用：

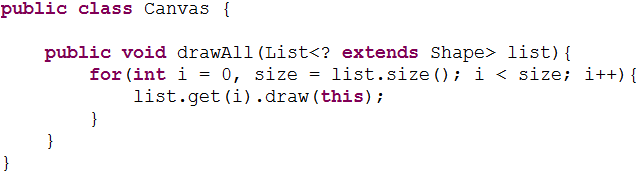


在上面的例子中会发生编译错误，因为List<Circle>并不是List<Shape>的子类，我们可以采用泛型，将Canvas类修改为：



但是这种做法显得非常臃肿而繁琐，使用了泛型还需要进行强制类型转换，实际上我们仅仅需要的是一种可以表示所有Shape泛型List的父类。为了满足这种需求，java泛型提供了被限制的泛型通配符，被限制的泛型通配符表示如下：

List<? Entends Shape>



这样就可以把List<Circle>对象当成List<? Entends Shape>使用，即List<? Extends Shape>可以表示List<Circle>、List<Rectangle>的父类—-只要List后尖括号里的类型是Shape的子类型即可。但是对于泛型，因为我们不知道这个受限制的通配符的具体类型，所以不能把Shape对象或其子类对象加入这个泛型集合中，例如，下面的代码是错误的：



**设定类型形参的上限**

Java泛型不仅允许在使用通配符形参时设定上限，而且可以在定义类型形参时设定上限，用于表示传给该类型形参的实际类型要么是该上限类型，要么是该上限类型的子类。

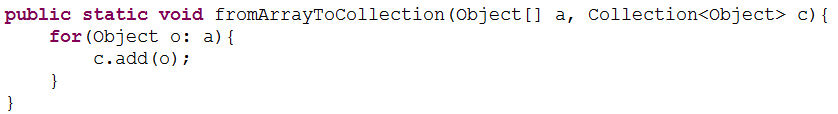
正确

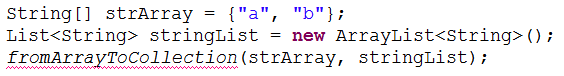
错误

**泛型方法**

前面介绍了在定义类、接口时可以使用类型形参，在该类的方法定义和Field定义、接口的方法定义中，这些类型形参可被当成普通类型来用。在另外一些情况下，我们定义类、接口时没有使用类型形参，但定义方法时想自己定义类型形参，这也是可以的。

我们需要实现这样一个方法—-该方法负责将一个Object数组的所有元素添加到一个Collection几何中，考虑下面的实现方法：





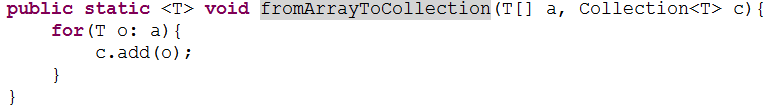
在上面的例子中，Collection<String>不是Collection<Object>的子类型—-所以这个方法的功能非常有限，它只能将Object数组的元素复制到Object（Object的子类不行）Collection集合中。那么用通配符是否可行呢？显然也不行，我们不能把一个对象放进一个未知类型的集合中。为了解决这个问题，我们可以使用java5 提供的泛型方法。所谓泛型方法，就是在声明方法时定义一个或多个类型形参，泛型方法的用法格式如下：

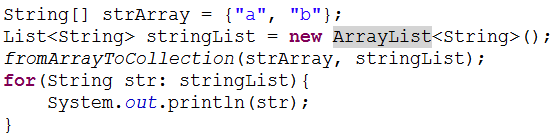
修饰符 <T, S> 返回值类型 方法名（形参列表）{

//方法体

}

基于上面的方法我们可以修改为：



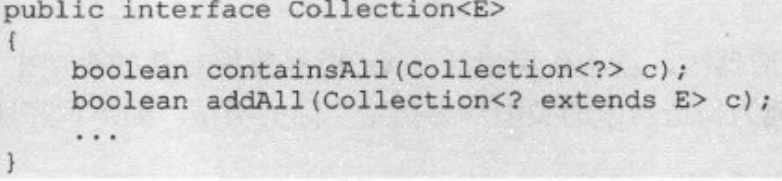


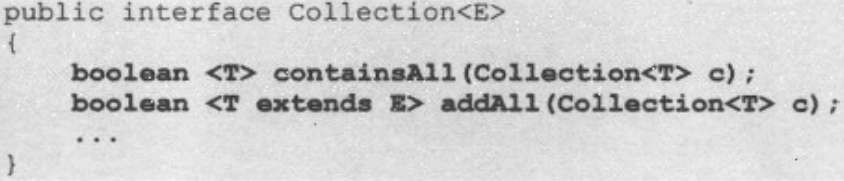


与接口、类声明中定义的类型形参不同的是，方法声明中定义的形参只能在该方法中使用，而接口、类声明中定义的类型形参则可以在整个接口、类中使用。

**泛型方法和类型通配符的区别**

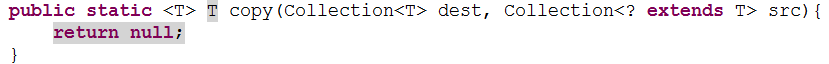
大多数时候都可以使用泛型方法来代替类型通配符。

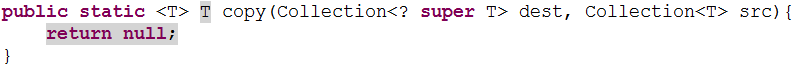




**设定通配符下限**

注意下面的两种方式：





在上面的两种方式中都可以实现相同的功能，但是由于有返回值，在第一种方式中，src可以为Number类型，dest可以为Integer类型，但是返回值是Number类型，但是在第二种方式中T则变为了Integer类型

**擦除和转换**

在严格的泛型代码中，带泛型声明的类型总应该带着类型参数，但为了与老的java代码保持一致，也允许在使用带泛型声明的类时不指定实际的类型参数，如果没有为这个泛型类指定实际的类型参数，则该类型参数被称作raw type（原始类型），默认是声明该参数时指定的第一个上限类型。

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

在上面的代码中，list中的元素就直接变为了Object类型