JAVA认证:用J2SE5.0创建定制的泛型集合Java认证考试 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/559/2021_2022_JAVA_E8_AE_A4_E8_AF_81_c104_559577.htm 一、 创建支持泛型的类 首先，你必须学习如何创建一个允许存在"泛型类型"的类。这意味着无论何时实例化你的类，你都能够指定一个或多个Java类型与该类相关联。为了说明这个问题，请考虑列表1中的一个简单示例类。 注意，列表1中的类是如何声明的。它在尖括号之间指定三个泛型。这些泛型是真实类型的占位符。当你声明一个这种类型的类时，你可以指定一个类来代替ONE，TWO和THREE。如果你不这样做，那么该类将使用Object的默认类型。 这个类显示出怎样设计一个类来接收三个泛型类型。当你创建一个这种类型的类时你要支持准确的类型。 列表1.泛型类: package com.heatonresearch.examples.collections.public class Example { private ONE one. private TWO two. private THREE three. public ONE getOne() { return one. } public void setOne(ONE one) { this.one = one. } public THREE getThree() { return three. } public void setThree(THREE three) { this.three = three. } public TWO getTwo() { return two. } public void setTwo(TWO two) { this.two = two. } public static void main(String args[]) {Example example = new Example().example.setOne(1.5).example.setTwo(2).example.setThree("Three"). }} 下面是如何实例化一个Example类型的类的情形：Example example=new Example(). 前面的代码将代替具体的Double，Integer和String类型-相当于在列表1中的"ONE"

、"TWO"和"THREE"占位符。你可以看到这些变量都有这些类型，通过下面三行设置它们的值。 example.setOne(1.5).example.setTwo(2).example.setThree("Three"). 现在，既然你已经知道如何创建一个使用泛型的定制类，那么创建一个使用泛型的定制集合类则更为简单些。 二、 创建一个Queue类 一个队列是一个很有用的数据结构。为了理解一个队列的功能，你可以想像在一个娱乐公园人们排队骑马的情形。人们从队的后面进入到队中。为此，他们等待而最后到达队伍的前端。其顺序不能改变。 这种情形可以被应用到一个队列类上去。它共有两个方法，分别是"push"和"pop"。你使用push方法来把对象放置到队列中，而使用pop方法从队列中删除一项。例如，如果你使用push方法把三个对象添加到队列上，那么连续调用pop三次将以同样顺序从队列中删除这三个元素。这正与娱乐公园的情形相一致。如果有三个人以一特定的顺序进入队中，他们将以相同的顺序得到骑马娱乐。 下列代码显示出怎么实现一个使用泛型的Java队列。 package com.heatonresearch.examples.collections.import java.util.*.public class Queue {private ArrayList list = new ArrayList().public void push(T obj) { list.add(obj). }public T pop() throws QueueException {if (size() == 0)throw new QueueException("Tried to pop something from the queue, " "when it was empty").T result = list.get(0).list.remove(0).return result.}public boolean isEmpty() { return list.isEmpty(). }public int size() { return list.size(). }public void clear() { list.clear(). }} 前面的代码声明了队列类，这样它可以接收一个泛型类型。 public class Queue 泛型类型"T"是该类

类型-它将被放入到该队列中去。为了把这些项存储到一个队列中，该类还要创建一个接收"T"类型的ArrayList。 push方法很简单的。它接收单一的类型为泛型"T"的对象，并且把它添加到ArrayList上。 pop方法稍微复杂些。首先，如果你要从队列中弹出一个对象，并且如果在队列中没有对象，那么该类将抛出一个QueueException类型的异常。下面是QueueException类。 package com.heatonresearch.examples.collections.public class QueueException extends Exception { public QueueException(String msg) {super(msg). }} 下面是抛出QueueException类型异常的代码： if (size() == 0)throw new QueueException("Tried to pop something from the queue, " "when it was empty"). 如果队列不空，该方法将从队列中检索最后一个元素，在一个名叫result的变量中存储它，然后从该列表中删除这个项。下面几行代码实现了这一功能： T result = list.get(0).list.remove(0).return result. 注意，该临时变量也是泛型类型"T"。当这个类与真实的代表泛型类型的Java类型一起使用时，为了实现最大程度上的兼容性，无论你何时存取这些变量，确保总是使用泛型类型是非常重要的。 三、 测试Queue类 下列类用于测试"泛型"队列。 package com.heatonresearch.examples.collections. public class TestQueue {public static void main(String args[]) { Queue queue = new Queue(). queue.push(1). queue.push(2). queue.push(3). try {System.out.println("Pop 1:" queue.pop()).System.out.println("Pop 2:" queue.pop()).System.out.println("Pop 3:" queue.pop()). } catch (QueueException e) { e.printStackTrace(). }} } 前面的代码中创建

的队列仅接收整型对象。 Queue queue = new Queue(). 接下来的测试把三个整数添加到该队列上。 queue.push(1).queue.push(2).queue.push(3). 注意，添加到该队列中的这些数字都是原始的类型。因为J2SE的自动装箱特性，这些原始的int类型被自动地转变成Integer对象。 接下来，该测试使用pop方法检索对象。在该队列为空的情况下，该测试捕获到QueueException异常。从队列中弹出三个数字的结果是： 123 尽管在这里作为一接收的整数队列显示，但是因为泛型，所以队列类对于任何Java对象情况都能正常工作。 四、 创建一个可预知的Stack集合 这里是一个更复杂的集合类型-它实现了一个堆栈以使你在实际删除一个对象之前能够预知或"可偷看"。你可以或者通过使用一个迭代算子或使用J2SE 5.0的新的"for each"结构语句来进行预知。 这个PeekableStack类是一个先进后出（FILO）栈-让你遍历当前栈中的内容。它的实现使用了两个类。首先，PeekableStack类实现实际的栈部分。其次，PeekableStackIterator类实现一个"Java标准的"Iterator类-你可以用它来遍历整个栈。列表 2（见所附源代码文件）显示出PeekableStack类的具体编码。 注意，列表2中的PeekableStack类实现了Iterable接口。这对于支持新型的J2SE 5.0"for-each"结构语句是必要的。该Iterable接口用于指定你的集合支持"iterator"方法-它返回一个迭代算子。如果没有这个接口，你的类将无法与新型的"for-each"结构语句相兼容。 这个可预知的栈包含push和pop方法，就象队列一样。该push方法仅仅是比队列稍微复杂些。而push方法负责把对象添加到栈上去并增加版本数（version）。 这个version变量允许PeekableStackIterator类保证没有修改操作发生。在迭代算子

创建时，这个算子保留一份当前版本数。如果栈上通过调用push方法发生任何变化，那么这个版本数就不会匹配；此不匹配将导致算子抛出一个 ConcurrentModificationException 异常。 pop方法稍微复杂些。首先，它必须决定在该列表中的最后一个元素，这是通过获得列表的大小并且减去1而得到的。 int last = list.size() - 1. 如果这个结果是一个小于零的数字，那么该栈就是空的，因此pop方法就返回null。 if (last &lt. 0) return null. 如果在栈中存在最后一个元素，那么就从列表中检索它。在从列表中成功地检索这个项后，你可以把它删除。 T result = list.get(last).list.remove(last). 最后，返回从列表中检索的对象。 return result. 为支持"for each"迭代，PeekableStack类的iterator方法返回一个"Java标准的"Iterator类-你可以用它来遍历包含在栈中的所有对象。iterator方法创建一个新的iterator并且返回之。 PeekableStackIterator peekableStackIterator=newPeekableStackIterator(this, list). 如你所见，该iterator类接收当前栈和栈的项目列表作为构造器参数。这些值将为PeekableStackIterator所用-下一节将讨论之。 五、 创建一个可预知的Stack迭代算子 如果PeekableStack类将要同Java中新的"for each"结构语句一起使用，那么你必须创建一个"Java标准的"Iterator。列表3显示出一个PeekableStackIterator类的实现。 在列表3中，迭代子实际上并没有以任何方式改变栈的值；代之的是，该迭代子追踪它在元素列表中的当前位置并且总是返回下一个元素。因为这个信息被存储在iteration类本身，所以有可能存在多个算子运行于相同的栈上。 下列程序用于测试可预知的栈。 package com.heatonresearch.examples.collections. import java.util.*. public

class TestPeekableStack {public static void main(String args[]) { PeekableStack stack = new PeekableStack(). stack.push(1). stack.push(2). stack.push(3). for (int i : stack) { System.out.println(i). } System.out.println("Pop 1:" stack.pop()). System.out.println("Pop 2:" stack.pop()). System.out.println("Pop 3:" stack.pop()).}} 如你所见，有三个项被添加到栈上去。然后，这三个项被使用新的"for each"结构语句显示出来。 for( int i: stack){ System.out.println( i ). } 因此，你看到怎样成功地实现一集合-它支持新型的J2SE惯例-既有泛型也有"for each"结构语句。如你所见，创建与J2SE 5.0中新型的结构相兼容的集合是相当容易的-这只需要利用泛型并且实现恰当的接口即可。你会发现这样的集合类被无缝地集成到J2SE 5.0中。 