001/\*  
002 \* Licensed to the Apache Software Foundation (ASF) under one or more  
003 \* contributor license agreements. See the NOTICE file distributed with  
004 \* this work for additional information regarding copyright ownership.  
005 \* The ASF licenses this file to You under the Apache License, Version 2.0  
006 \* (the "License"); you may not use this file except in compliance with  
007 \* the License. You may obtain a copy of the License at  
008 \*  
009 \* http://www.apache.org/licenses/LICENSE-2.0  
010 \*  
011 \* Unless required by applicable law or agreed to in writing, software  
012 \* distributed under the License is distributed on an "AS IS" BASIS,  
013 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
014 \* See the License for the specific language governing permissions and  
015 \* limitations under the License.  
016 \*/  
017package org.apache.commons.collections4.iterators;  
018  
019import java.util.ArrayList;  
020import java.util.Arrays;  
021import java.util.Collection;  
022import java.util.HashMap;  
023import java.util.Iterator;  
024import java.util.List;  
025import java.util.Map;  
026import java.util.NoSuchElementException;  
027  
028/\*\*  
029 \* This iterator creates permutations of an input collection, using the  
030 \* Steinhaus-Johnson-Trotter algorithm (also called plain changes).  
031 \* <p>  
032 \* The iterator will return exactly n! permutations of the input collection.  
033 \* The {@code remove()} operation is not supported, and will throw an  
034 \* {@code UnsupportedOperationException}.  
035 \* <p>  
036 \* NOTE: in case an empty collection is provided, the iterator will  
037 \* return exactly one empty list as result, as 0! = 1.  
038 \*  
039 \* @param <E> the type of the objects being permuted  
040 \*  
041 \* @since 4.0  
042 \*/  
043public class PermutationIterator<E> implements Iterator<List<E>> {  
044  
045 /\*\*  
046 \* Permutation is done on theses keys to handle equal objects.  
047 \*/  
048 private final int[] keys;  
049  
050 /\*\*  
051 \* Mapping between keys and objects.  
052 \*/  
053 private final Map<Integer, E> objectMap;  
054  
055 /\*\*  
056 \* Direction table used in the algorithm:  
057 \* <ul>  
058 \* <li>false is left</li>  
059 \* <li>true is right</li>  
060 \* </ul>  
061 \*/  
062 private final boolean[] direction;  
063  
064 /\*\*  
065 \* Next permutation to return. When a permutation is requested  
066 \* this instance is provided and the next one is computed.  
067 \*/  
068 private List<E> nextPermutation;  
069  
070 /\*\*  
071 \* Standard constructor for this class.  
072 \* @param coll the collection to generate permutations for  
073 \* @throws NullPointerException if coll is null  
074 \*/  
075 public PermutationIterator(final Collection<? extends E> coll) {  
076 if (coll == null) {  
077 throw new NullPointerException("The collection must not be null");  
078 }  
079  
080 keys = new int[coll.size()];  
081 direction = new boolean[coll.size()];  
082 Arrays.fill(direction, false);  
083 int value = 1;  
084 objectMap = new HashMap<>();  
085 for (final E e : coll) {  
086 objectMap.put(Integer.valueOf(value), e);  
087 keys[value - 1] = value;  
088 value++;  
089 }  
090 nextPermutation = new ArrayList<>(coll);  
091 }  
092  
093 /\*\*  
094 \* Indicates if there are more permutation available.  
095 \* @return true if there are more permutations, otherwise false  
096 \*/  
097 @Override  
098 public boolean hasNext() {  
099 return nextPermutation != null;  
100 }  
101  
102 /\*\*  
103 \* Returns the next permutation of the input collection.  
104 \* @return a list of the permutator's elements representing a permutation  
105 \* @throws NoSuchElementException if there are no more permutations  
106 \*/  
107 @Override  
108 public List<E> next() {  
109 if (!hasNext()) {  
110 throw new NoSuchElementException();  
111 }  
112  
113 // find the largest mobile integer k  
114 int indexOfLargestMobileInteger = -1;  
115 int largestKey = -1;  
116 for (int i = 0; i < keys.length; i++) {  
117 if ((direction[i] && i < keys.length - 1 && keys[i] > keys[i + 1]) ||  
118 (!direction[i] && i > 0 && keys[i] > keys[i - 1])) {  
119 if (keys[i] > largestKey) { // NOPMD  
120 largestKey = keys[i];  
121 indexOfLargestMobileInteger = i;  
122 }  
123 }  
124 }  
125 if (largestKey == -1) {  
126 final List<E> toReturn = nextPermutation;  
127 nextPermutation = null;  
128 return toReturn;  
129 }  
130  
131 // swap k and the adjacent integer it is looking at  
132 final int offset = direction[indexOfLargestMobileInteger] ? 1 : -1;  
133 final int tmpKey = keys[indexOfLargestMobileInteger];  
134 keys[indexOfLargestMobileInteger] = keys[indexOfLargestMobileInteger + offset];  
135 keys[indexOfLargestMobileInteger + offset] = tmpKey;  
136 final boolean tmpDirection = direction[indexOfLargestMobileInteger];  
137 direction[indexOfLargestMobileInteger] = direction[indexOfLargestMobileInteger + offset];  
138 direction[indexOfLargestMobileInteger + offset] = tmpDirection;  
139  
140 // reverse the direction of all integers larger than k and build the result  
141 final List<E> nextP = new ArrayList<>();  
142 for (int i = 0; i < keys.length; i++) {  
143 if (keys[i] > largestKey) {  
144 direction[i] = !direction[i];  
145 }  
146 nextP.add(objectMap.get(Integer.valueOf(keys[i])));  
147 }  
148 final List<E> result = nextPermutation;  
149 nextPermutation = nextP;  
150 return result;  
151 }  
152  
153 @Override  
154 public void remove() {  
155 throw new UnsupportedOperationException("remove() is not supported");  
156 }  
157  
158}