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.collection;  
018  
019import java.util.Collection;  
020import java.util.Iterator;  
021import java.util.function.Predicate;  
022  
023import org.apache.commons.collections4.BoundedCollection;  
024import org.apache.commons.collections4.Unmodifiable;  
025import org.apache.commons.collections4.iterators.UnmodifiableIterator;  
026  
027/\*\*  
028 \* {@link UnmodifiableBoundedCollection} decorates another  
029 \* {@link BoundedCollection} to ensure it can't be altered.  
030 \* <p>  
031 \* If a BoundedCollection is first wrapped in some other collection decorator,  
032 \* such as synchronized or predicated, the BoundedCollection methods are no  
033 \* longer accessible.  
034 \* The factory on this class will attempt to retrieve the bounded nature by  
035 \* examining the package scope variables.  
036 \* </p>  
037 \* <p>  
038 \* This class is Serializable from Commons Collections 3.1.  
039 \* </p>  
040 \* <p>  
041 \* Attempts to modify it will result in an UnsupportedOperationException.  
042 \* </p>  
043 \*  
044 \* @param <E> the type of elements in this collection  
045 \* @since 3.0  
046 \*/  
047public final class UnmodifiableBoundedCollection<E> extends AbstractCollectionDecorator<E>  
048 implements BoundedCollection<E>, Unmodifiable {  
049  
050 /\*\* Serialization version \*/  
051 private static final long serialVersionUID = -7112672385450340330L;  
052  
053 /\*\*  
054 \* Factory method to create an unmodifiable bounded collection.  
055 \*  
056 \* @param <E> the type of the elements in the collection  
057 \* @param coll the <code>BoundedCollection</code> to decorate, must not be null  
058 \* @return a new unmodifiable bounded collection  
059 \* @throws NullPointerException if {@code coll} is {@code null}  
060 \* @since 4.0  
061 \*/  
062 public static <E> BoundedCollection<E> unmodifiableBoundedCollection(final BoundedCollection<? extends E> coll) {  
063 if (coll instanceof Unmodifiable) {  
064 @SuppressWarnings("unchecked") // safe to upcast  
065 final BoundedCollection<E> tmpColl = (BoundedCollection<E>) coll;  
066 return tmpColl;  
067 }  
068 return new UnmodifiableBoundedCollection<>(coll);  
069 }  
070  
071 /\*\*  
072 \* Factory method to create an unmodifiable bounded collection.  
073 \* <p>  
074 \* This method is capable of drilling down through up to 1000 other decorators  
075 \* to find a suitable BoundedCollection.  
076 \*  
077 \* @param <E> the type of the elements in the collection  
078 \* @param coll the <code>BoundedCollection</code> to decorate, must not be null  
079 \* @return a new unmodifiable bounded collection  
080 \* @throws NullPointerException if coll is null  
081 \* @throws IllegalArgumentException if coll is not a {@code BoundedCollection}  
082 \* @since 4.0  
083 \*/  
084 @SuppressWarnings("unchecked")  
085 public static <E> BoundedCollection<E> unmodifiableBoundedCollection(Collection<? extends E> coll) {  
086 if (coll == null) {  
087 throw new NullPointerException("Collection must not be null.");  
088 }  
089  
090 // handle decorators  
091 for (int i = 0; i < 1000; i++) { // counter to prevent infinite looping  
092 if (coll instanceof BoundedCollection) {  
093 break; // normal loop exit  
094 }  
095 if (coll instanceof AbstractCollectionDecorator) {  
096 coll = ((AbstractCollectionDecorator<E>) coll).decorated();  
097 } else if (coll instanceof SynchronizedCollection) {  
098 coll = ((SynchronizedCollection<E>) coll).decorated();  
099 }  
100 }  
101  
102 if (coll instanceof BoundedCollection == false) {  
103 throw new IllegalArgumentException("Collection is not a bounded collection.");  
104 }  
105 return new UnmodifiableBoundedCollection<>((BoundedCollection<E>) coll);  
106 }  
107  
108 /\*\*  
109 \* Constructor that wraps (not copies).  
110 \*  
111 \* @param coll the collection to decorate, must not be null  
112 \* @throws NullPointerException if coll is null  
113 \*/  
114 @SuppressWarnings("unchecked") // safe to upcast  
115 private UnmodifiableBoundedCollection(final BoundedCollection<? extends E> coll) {  
116 super((BoundedCollection<E>) coll);  
117 }  
118  
119 //-----------------------------------------------------------------------  
120 @Override  
121 public Iterator<E> iterator() {  
122 return UnmodifiableIterator.unmodifiableIterator(decorated().iterator());  
123 }  
124  
125 @Override  
126 public boolean add(final E object) {  
127 throw new UnsupportedOperationException();  
128 }  
129  
130 @Override  
131 public boolean addAll(final Collection<? extends E> coll) {  
132 throw new UnsupportedOperationException();  
133 }  
134  
135 @Override  
136 public void clear() {  
137 throw new UnsupportedOperationException();  
138 }  
139  
140 @Override  
141 public boolean remove(final Object object) {  
142 throw new UnsupportedOperationException();  
143 }  
144  
145 /\*\*  
146 \* @since 4.4  
147 \*/  
148 @Override  
149 public boolean removeIf(final Predicate<? super E> filter) {  
150 throw new UnsupportedOperationException();  
151 }  
152  
153 @Override  
154 public boolean removeAll(final Collection<?> coll) {  
155 throw new UnsupportedOperationException();  
156 }  
157  
158 @Override  
159 public boolean retainAll(final Collection<?> coll) {  
160 throw new UnsupportedOperationException();  
161 }  
162  
163 //-----------------------------------------------------------------------  
164 @Override  
165 public boolean isFull() {  
166 return decorated().isFull();  
167 }  
168  
169 @Override  
170 public int maxSize() {  
171 return decorated().maxSize();  
172 }  
173  
174 @Override  
175 protected BoundedCollection<E> decorated() {  
176 return (BoundedCollection<E>) super.decorated();  
177 }  
178}