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.set;  
018  
019import java.io.IOException;  
020import java.io.ObjectInputStream;  
021import java.io.ObjectOutputStream;  
022import java.util.Collection;  
023import java.util.Iterator;  
024import java.util.NavigableSet;  
025import java.util.SortedSet;  
026import java.util.function.Predicate;  
027  
028import org.apache.commons.collections4.Unmodifiable;  
029import org.apache.commons.collections4.iterators.UnmodifiableIterator;  
030  
031/\*\*  
032 \* Decorates another <code>NavigableSet</code> to ensure it can't be altered.  
033 \* <p>  
034 \* Attempts to modify it will result in an UnsupportedOperationException.  
035 \* </p>  
036 \*  
037 \* @param <E> the type of the elements in this set  
038 \* @since 4.1  
039 \*/  
040public final class UnmodifiableNavigableSet<E>  
041 extends AbstractNavigableSetDecorator<E>  
042 implements Unmodifiable {  
043  
044 /\*\* Serialization version \*/  
045 private static final long serialVersionUID = 20150528L;  
046  
047 /\*\*  
048 \* Factory method to create an unmodifiable set.  
049 \*  
050 \* @param <E> the element type  
051 \* @param set the set to decorate, must not be null  
052 \* @return a new unmodifiable {@link NavigableSet}  
053 \* @throws NullPointerException if set is null  
054 \*/  
055 public static <E> NavigableSet<E> unmodifiableNavigableSet(final NavigableSet<E> set) {  
056 if (set instanceof Unmodifiable) {  
057 return set;  
058 }  
059 return new UnmodifiableNavigableSet<>(set);  
060 }  
061  
062 //-----------------------------------------------------------------------  
063 /\*\*  
064 \* Constructor that wraps (not copies).  
065 \*  
066 \* @param set the set to decorate, must not be null  
067 \* @throws NullPointerException if set is null  
068 \*/  
069 private UnmodifiableNavigableSet(final NavigableSet<E> set) {  
070 super(set);  
071 }  
072  
073 //-----------------------------------------------------------------------  
074 @Override  
075 public Iterator<E> iterator() {  
076 return UnmodifiableIterator.unmodifiableIterator(decorated().iterator());  
077 }  
078  
079 @Override  
080 public boolean add(final E object) {  
081 throw new UnsupportedOperationException();  
082 }  
083  
084 @Override  
085 public boolean addAll(final Collection<? extends E> coll) {  
086 throw new UnsupportedOperationException();  
087 }  
088  
089 @Override  
090 public void clear() {  
091 throw new UnsupportedOperationException();  
092 }  
093  
094 @Override  
095 public boolean remove(final Object object) {  
096 throw new UnsupportedOperationException();  
097 }  
098  
099 /\*\*  
100 \* @since 4.4  
101 \*/  
102 @Override  
103 public boolean removeIf(Predicate<? super E> filter) {  
104 throw new UnsupportedOperationException();  
105 }  
106  
107 @Override  
108 public boolean removeAll(final Collection<?> coll) {  
109 throw new UnsupportedOperationException();  
110 }  
111  
112 @Override  
113 public boolean retainAll(final Collection<?> coll) {  
114 throw new UnsupportedOperationException();  
115 }  
116  
117 // SortedSet  
118 //-----------------------------------------------------------------------  
119 @Override  
120 public SortedSet<E> subSet(final E fromElement, final E toElement) {  
121 final SortedSet<E> sub = decorated().subSet(fromElement, toElement);  
122 return UnmodifiableSortedSet.unmodifiableSortedSet(sub);  
123 }  
124  
125 @Override  
126 public SortedSet<E> headSet(final E toElement) {  
127 final SortedSet<E> head = decorated().headSet(toElement);  
128 return UnmodifiableSortedSet.unmodifiableSortedSet(head);  
129 }  
130  
131 @Override  
132 public SortedSet<E> tailSet(final E fromElement) {  
133 final SortedSet<E> tail = decorated().tailSet(fromElement);  
134 return UnmodifiableSortedSet.unmodifiableSortedSet(tail);  
135 }  
136  
137 // NavigableSet  
138 //-----------------------------------------------------------------------  
139 @Override  
140 public NavigableSet<E> descendingSet() {  
141 return unmodifiableNavigableSet(decorated().descendingSet());  
142 }  
143  
144 @Override  
145 public Iterator<E> descendingIterator() {  
146 return UnmodifiableIterator.unmodifiableIterator(decorated().descendingIterator());  
147 }  
148  
149 @Override  
150 public NavigableSet<E> subSet(final E fromElement, final boolean fromInclusive, final E toElement,  
151 final boolean toInclusive) {  
152 final NavigableSet<E> sub = decorated().subSet(fromElement, fromInclusive, toElement, toInclusive);  
153 return unmodifiableNavigableSet(sub);  
154 }  
155  
156 @Override  
157 public NavigableSet<E> headSet(final E toElement, final boolean inclusive) {  
158 final NavigableSet<E> head = decorated().headSet(toElement, inclusive);  
159 return unmodifiableNavigableSet(head);  
160 }  
161  
162 @Override  
163 public NavigableSet<E> tailSet(final E fromElement, final boolean inclusive) {  
164 final NavigableSet<E> tail = decorated().tailSet(fromElement, inclusive);  
165 return unmodifiableNavigableSet(tail);  
166 }  
167  
168 //-----------------------------------------------------------------------  
169 /\*\*  
170 \* Write the collection out using a custom routine.  
171 \*  
172 \* @param out the output stream  
173 \* @throws IOException if an error occurs while writing to the stream  
174 \*/  
175 private void writeObject(final ObjectOutputStream out) throws IOException {  
176 out.defaultWriteObject();  
177 out.writeObject(decorated());  
178 }  
179  
180 /\*\*  
181 \* Read the collection in using a custom routine.  
182 \*  
183 \* @param in the input stream  
184 \* @throws IOException if an error occurs while reading from the stream  
185 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
186 \*/  
187 @SuppressWarnings("unchecked") // (1) should only fail if input stream is incorrect  
188 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
189 in.defaultReadObject();  
190 setCollection((Collection<E>) in.readObject()); // (1)  
191 }  
192  
193}