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.SortedSet;  
025import java.util.function.Predicate;  
026  
027import org.apache.commons.collections4.Unmodifiable;  
028import org.apache.commons.collections4.iterators.UnmodifiableIterator;  
029  
030/\*\*  
031 \* Decorates another <code>SortedSet</code> to ensure it can't be altered.  
032 \* <p>  
033 \* This class is Serializable from Commons Collections 3.1.  
034 \* </p>  
035 \* <p>  
036 \* Attempts to modify it will result in an UnsupportedOperationException.  
037 \* </p>  
038 \*  
039 \* @param <E> the type of the elements in this set  
040 \* @since 3.0  
041 \*/  
042public final class UnmodifiableSortedSet<E>  
043 extends AbstractSortedSetDecorator<E>  
044 implements Unmodifiable {  
045  
046 /\*\* Serialization version \*/  
047 private static final long serialVersionUID = -725356885467962424L;  
048  
049 /\*\*  
050 \* Factory method to create an unmodifiable set.  
051 \*  
052 \* @param <E> the element type  
053 \* @param set the set to decorate, must not be null  
054 \* @return a new unmodifiable {@link SortedSet}  
055 \* @throws NullPointerException if set is null  
056 \* @since 4.0  
057 \*/  
058 public static <E> SortedSet<E> unmodifiableSortedSet(final SortedSet<E> set) {  
059 if (set instanceof Unmodifiable) {  
060 return set;  
061 }  
062 return new UnmodifiableSortedSet<>(set);  
063 }  
064  
065 //-----------------------------------------------------------------------  
066 /\*\*  
067 \* Constructor that wraps (not copies).  
068 \*  
069 \* @param set the set to decorate, must not be null  
070 \* @throws NullPointerException if set is null  
071 \*/  
072 private UnmodifiableSortedSet(final SortedSet<E> set) {  
073 super(set);  
074 }  
075  
076 //-----------------------------------------------------------------------  
077 @Override  
078 public Iterator<E> iterator() {  
079 return UnmodifiableIterator.unmodifiableIterator(decorated().iterator());  
080 }  
081  
082 @Override  
083 public boolean add(final E object) {  
084 throw new UnsupportedOperationException();  
085 }  
086  
087 @Override  
088 public boolean addAll(final Collection<? extends E> coll) {  
089 throw new UnsupportedOperationException();  
090 }  
091  
092 @Override  
093 public void clear() {  
094 throw new UnsupportedOperationException();  
095 }  
096  
097 @Override  
098 public boolean remove(final Object object) {  
099 throw new UnsupportedOperationException();  
100 }  
101  
102 /\*\*  
103 \* @since 4.4  
104 \*/  
105 @Override  
106 public boolean removeIf(Predicate<? super E> filter) {  
107 throw new UnsupportedOperationException();  
108 }  
109  
110 @Override  
111 public boolean removeAll(final Collection<?> coll) {  
112 throw new UnsupportedOperationException();  
113 }  
114  
115 @Override  
116 public boolean retainAll(final Collection<?> coll) {  
117 throw new UnsupportedOperationException();  
118 }  
119  
120 //-----------------------------------------------------------------------  
121 @Override  
122 public SortedSet<E> subSet(final E fromElement, final E toElement) {  
123 final SortedSet<E> sub = decorated().subSet(fromElement, toElement);  
124 return unmodifiableSortedSet(sub);  
125 }  
126  
127 @Override  
128 public SortedSet<E> headSet(final E toElement) {  
129 final SortedSet<E> head = decorated().headSet(toElement);  
130 return unmodifiableSortedSet(head);  
131 }  
132  
133 @Override  
134 public SortedSet<E> tailSet(final E fromElement) {  
135 final SortedSet<E> tail = decorated().tailSet(fromElement);  
136 return unmodifiableSortedSet(tail);  
137 }  
138  
139 //-----------------------------------------------------------------------  
140 /\*\*  
141 \* Write the collection out using a custom routine.  
142 \*  
143 \* @param out the output stream  
144 \* @throws IOException if an error occurs while writing to the stream  
145 \*/  
146 private void writeObject(final ObjectOutputStream out) throws IOException {  
147 out.defaultWriteObject();  
148 out.writeObject(decorated());  
149 }  
150  
151 /\*\*  
152 \* Read the collection in using a custom routine.  
153 \*  
154 \* @param in the input stream  
155 \* @throws IOException if an error occurs while reading from the stream  
156 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
157 \*/  
158 @SuppressWarnings("unchecked") // (1) should only fail if input stream is incorrect  
159 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
160 in.defaultReadObject();  
161 setCollection((Collection<E>) in.readObject()); // (1)  
162 }  
163  
164}