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.util.Comparator;  
020import java.util.SortedSet;  
021  
022import org.apache.commons.collections4.Transformer;  
023  
024/\*\*  
025 \* Decorates another <code>SortedSet</code> to transform objects that are added.  
026 \* <p>  
027 \* The add methods are affected by this class.  
028 \* Thus objects must be removed or searched for using their transformed form.  
029 \* For example, if the transformation converts Strings to Integers, you must  
030 \* use the Integer form to remove objects.  
031 \* </p>  
032 \* <p>  
033 \* This class is Serializable from Commons Collections 3.1.  
034 \* </p>  
035 \*  
036 \* @param <E> the type of the elements in this set  
037 \* @since 3.0  
038 \*/  
039public class TransformedSortedSet<E> extends TransformedSet<E> implements SortedSet<E> {  
040  
041 /\*\* Serialization version \*/  
042 private static final long serialVersionUID = -1675486811351124386L;  
043  
044 /\*\*  
045 \* Factory method to create a transforming sorted set.  
046 \* <p>  
047 \* If there are any elements already in the set being decorated, they  
048 \* are NOT transformed.  
049 \* Contrast this with {@link #transformedSortedSet(SortedSet, Transformer)}.  
050 \*  
051 \* @param <E> the element type  
052 \* @param set the set to decorate, must not be null  
053 \* @param transformer the transformer to use for conversion, must not be null  
054 \* @return a new transformed {@link SortedSet}  
055 \* @throws NullPointerException if set or transformer is null  
056 \* @since 4.0  
057 \*/  
058 public static <E> TransformedSortedSet<E> transformingSortedSet(final SortedSet<E> set,  
059 final Transformer<? super E, ? extends E> transformer) {  
060 return new TransformedSortedSet<>(set, transformer);  
061 }  
062  
063 /\*\*  
064 \* Factory method to create a transforming sorted set that will transform  
065 \* existing contents of the specified sorted set.  
066 \* <p>  
067 \* If there are any elements already in the set being decorated, they  
068 \* will be transformed by this method.  
069 \* Contrast this with {@link #transformingSortedSet(SortedSet, Transformer)}.  
070 \*  
071 \* @param <E> the element type  
072 \* @param set the set to decorate, must not be null  
073 \* @param transformer the transformer to use for conversion, must not be null  
074 \* @return a new transformed {@link SortedSet}  
075 \* @throws NullPointerException if set or transformer is null  
076 \* @since 4.0  
077 \*/  
078 public static <E> TransformedSortedSet<E> transformedSortedSet(final SortedSet<E> set,  
079 final Transformer<? super E, ? extends E> transformer) {  
080  
081 final TransformedSortedSet<E> decorated = new TransformedSortedSet<>(set, transformer);  
082 if (set.size() > 0) {  
083 @SuppressWarnings("unchecked") // set is type E  
084 final E[] values = (E[]) set.toArray(); // NOPMD - false positive for generics  
085 set.clear();  
086 for (final E value : values) {  
087 decorated.decorated().add(transformer.transform(value));  
088 }  
089 }  
090 return decorated;  
091 }  
092  
093 //-----------------------------------------------------------------------  
094 /\*\*  
095 \* Constructor that wraps (not copies).  
096 \* <p>  
097 \* If there are any elements already in the set being decorated, they  
098 \* are NOT transformed.  
099 \*  
100 \* @param set the set to decorate, must not be null  
101 \* @param transformer the transformer to use for conversion, must not be null  
102 \* @throws NullPointerException if set or transformer is null  
103 \*/  
104 protected TransformedSortedSet(final SortedSet<E> set, final Transformer<? super E, ? extends E> transformer) {  
105 super(set, transformer);  
106 }  
107  
108 /\*\*  
109 \* Gets the decorated set.  
110 \*  
111 \* @return the decorated set  
112 \*/  
113 protected SortedSet<E> getSortedSet() {  
114 return (SortedSet<E>) decorated();  
115 }  
116  
117 //-----------------------------------------------------------------------  
118 @Override  
119 public E first() {  
120 return getSortedSet().first();  
121 }  
122  
123 @Override  
124 public E last() {  
125 return getSortedSet().last();  
126 }  
127  
128 @Override  
129 public Comparator<? super E> comparator() {  
130 return getSortedSet().comparator();  
131 }  
132  
133 //-----------------------------------------------------------------------  
134 @Override  
135 public SortedSet<E> subSet(final E fromElement, final E toElement) {  
136 final SortedSet<E> set = getSortedSet().subSet(fromElement, toElement);  
137 return new TransformedSortedSet<>(set, transformer);  
138 }  
139  
140 @Override  
141 public SortedSet<E> headSet(final E toElement) {  
142 final SortedSet<E> set = getSortedSet().headSet(toElement);  
143 return new TransformedSortedSet<>(set, transformer);  
144 }  
145  
146 @Override  
147 public SortedSet<E> tailSet(final E fromElement) {  
148 final SortedSet<E> set = getSortedSet().tailSet(fromElement);  
149 return new TransformedSortedSet<>(set, transformer);  
150 }  
151  
152}