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.map;  
018  
019import java.util.Comparator;  
020import java.util.Map;  
021import java.util.SortedMap;  
022  
023import org.apache.commons.collections4.Transformer;  
024  
025/\*\*  
026 \* Decorates another <code>SortedMap </code> to transform objects that are added.  
027 \* <p>  
028 \* The Map put methods and Map.Entry setValue method are affected by this class.  
029 \* Thus objects must be removed or searched for using their transformed form.  
030 \* For example, if the transformation converts Strings to Integers, you must  
031 \* use the Integer form to remove objects.  
032 \* </p>  
033 \* <p>  
034 \* <strong>Note that TransformedSortedMap is not synchronized and is not thread-safe.</strong>  
035 \* If you wish to use this map from multiple threads concurrently, you must use  
036 \* appropriate synchronization. The simplest approach is to wrap this map  
037 \* using {@link java.util.Collections#synchronizedSortedMap}. This class may throw  
038 \* exceptions when accessed by concurrent threads without synchronization.  
039 \* </p>  
040 \* <p>  
041 \* This class is Serializable from Commons Collections 3.1.  
042 \* </p>  
043 \*  
044 \* @param <K> the type of the keys in this map  
045 \* @param <V> the type of the values in this map  
046 \* @since 3.0  
047 \*/  
048public class TransformedSortedMap<K, V>  
049 extends TransformedMap<K, V>  
050 implements SortedMap<K, V> {  
051  
052 /\*\* Serialization version \*/  
053 private static final long serialVersionUID = -8751771676410385778L;  
054  
055 /\*\*  
056 \* Factory method to create a transforming sorted map.  
057 \* <p>  
058 \* If there are any elements already in the map being decorated, they are NOT transformed.  
059 \* Contrast this with {@link #transformedSortedMap(SortedMap, Transformer, Transformer)}.  
060 \*  
061 \* @param <K> the key type  
062 \* @param <V> the value type  
063 \* @param map the map to decorate, must not be null  
064 \* @param keyTransformer the predicate to validate the keys, null means no transformation  
065 \* @param valueTransformer the predicate to validate to values, null means no transformation  
066 \* @return a new transformed sorted map  
067 \* @throws NullPointerException if the map is null  
068 \* @since 4.0  
069 \*/  
070 public static <K, V> TransformedSortedMap<K, V> transformingSortedMap(final SortedMap<K, V> map,  
071 final Transformer<? super K, ? extends K> keyTransformer,  
072 final Transformer<? super V, ? extends V> valueTransformer) {  
073 return new TransformedSortedMap<>(map, keyTransformer, valueTransformer);  
074 }  
075  
076 /\*\*  
077 \* Factory method to create a transforming sorted map that will transform  
078 \* existing contents of the specified map.  
079 \* <p>  
080 \* If there are any elements already in the map being decorated, they  
081 \* will be transformed by this method.  
082 \* Contrast this with {@link #transformingSortedMap(SortedMap, Transformer, Transformer)}.  
083 \*  
084 \* @param <K> the key type  
085 \* @param <V> the value type  
086 \* @param map the map to decorate, must not be null  
087 \* @param keyTransformer the transformer to use for key conversion, null means no transformation  
088 \* @param valueTransformer the transformer to use for value conversion, null means no transformation  
089 \* @return a new transformed sorted map  
090 \* @throws NullPointerException if map is null  
091 \* @since 4.0  
092 \*/  
093 public static <K, V> TransformedSortedMap<K, V> transformedSortedMap(final SortedMap<K, V> map,  
094 final Transformer<? super K, ? extends K> keyTransformer,  
095 final Transformer<? super V, ? extends V> valueTransformer) {  
096  
097 final TransformedSortedMap<K, V> decorated =  
098 new TransformedSortedMap<>(map, keyTransformer, valueTransformer);  
099 if (map.size() > 0) {  
100 final Map<K, V> transformed = decorated.transformMap(map);  
101 decorated.clear();  
102 decorated.decorated().putAll(transformed); // avoids double transformation  
103 }  
104 return decorated;  
105 }  
106  
107 //-----------------------------------------------------------------------  
108 /\*\*  
109 \* Constructor that wraps (not copies).  
110 \* <p>  
111 \* If there are any elements already in the collection being decorated, they  
112 \* are NOT transformed.</p>  
113 \*  
114 \* @param map the map to decorate, must not be null  
115 \* @param keyTransformer the predicate to validate the keys, null means no transformation  
116 \* @param valueTransformer the predicate to validate to values, null means no transformation  
117 \* @throws NullPointerException if the map is null  
118 \*/  
119 protected TransformedSortedMap(final SortedMap<K, V> map,  
120 final Transformer<? super K, ? extends K> keyTransformer,  
121 final Transformer<? super V, ? extends V> valueTransformer) {  
122 super(map, keyTransformer, valueTransformer);  
123 }  
124  
125 //-----------------------------------------------------------------------  
126 /\*\*  
127 \* Gets the map being decorated.  
128 \*  
129 \* @return the decorated map  
130 \*/  
131 protected SortedMap<K, V> getSortedMap() {  
132 return (SortedMap<K, V>) map;  
133 }  
134  
135 //-----------------------------------------------------------------------  
136 @Override  
137 public K firstKey() {  
138 return getSortedMap().firstKey();  
139 }  
140  
141 @Override  
142 public K lastKey() {  
143 return getSortedMap().lastKey();  
144 }  
145  
146 @Override  
147 public Comparator<? super K> comparator() {  
148 return getSortedMap().comparator();  
149 }  
150  
151 @Override  
152 public SortedMap<K, V> subMap(final K fromKey, final K toKey) {  
153 final SortedMap<K, V> map = getSortedMap().subMap(fromKey, toKey);  
154 return new TransformedSortedMap<>(map, keyTransformer, valueTransformer);  
155 }  
156  
157 @Override  
158 public SortedMap<K, V> headMap(final K toKey) {  
159 final SortedMap<K, V> map = getSortedMap().headMap(toKey);  
160 return new TransformedSortedMap<>(map, keyTransformer, valueTransformer);  
161 }  
162  
163 @Override  
164 public SortedMap<K, V> tailMap(final K fromKey) {  
165 final SortedMap<K, V> map = getSortedMap().tailMap(fromKey);  
166 return new TransformedSortedMap<>(map, keyTransformer, valueTransformer);  
167 }  
168  
169}