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.multimap;  
018  
019import java.util.Iterator;  
020import java.util.Map;  
021  
022import org.apache.commons.collections4.CollectionUtils;  
023import org.apache.commons.collections4.FluentIterable;  
024import org.apache.commons.collections4.MultiValuedMap;  
025import org.apache.commons.collections4.Transformer;  
026  
027/\*\*  
028 \* Decorates another <code>MultiValuedMap</code> to transform objects that are added.  
029 \* <p>  
030 \* This class affects the MultiValuedMap put methods. Thus objects must be  
031 \* removed or searched for using their transformed form. For example, if the  
032 \* transformation converts Strings to Integers, you must use the Integer form to  
033 \* remove objects.  
034 \* </p>  
035 \* <p>  
036 \* <strong>Note that TransformedMultiValuedMap is not synchronized and is not thread-safe.</strong>  
037 \* </p>  
038 \*  
039 \* @param <K> the type of the keys in this map  
040 \* @param <V> the type of the values in this map  
041 \* @since 4.1  
042 \*/  
043public class TransformedMultiValuedMap<K, V> extends AbstractMultiValuedMapDecorator<K, V> {  
044  
045 /\*\* Serialization Version \*/  
046 private static final long serialVersionUID = 20150612L;  
047  
048 /\*\* The key transformer \*/  
049 private final Transformer<? super K, ? extends K> keyTransformer;  
050  
051 /\*\* The value transformer \*/  
052 private final Transformer<? super V, ? extends V> valueTransformer;  
053  
054 /\*\*  
055 \* Factory method to create a transforming MultiValuedMap.  
056 \* <p>  
057 \* If there are any elements already in the map being decorated, they are  
058 \* NOT transformed. Contrast this with  
059 \* {@link #transformedMap(MultiValuedMap, Transformer, Transformer)}.  
060 \*  
061 \* @param <K> the key type  
062 \* @param <V> the value type  
063 \* @param map the MultiValuedMap to decorate, may not be null  
064 \* @param keyTransformer the transformer to use for key conversion, null means no conversion  
065 \* @param valueTransformer the transformer to use for value conversion, null means no conversion  
066 \* @return a new transformed MultiValuedMap  
067 \* @throws NullPointerException if map is null  
068 \*/  
069 public static <K, V> TransformedMultiValuedMap<K, V> transformingMap(final MultiValuedMap<K, V> map,  
070 final Transformer<? super K, ? extends K> keyTransformer,  
071 final Transformer<? super V, ? extends V> valueTransformer) {  
072 return new TransformedMultiValuedMap<>(map, keyTransformer, valueTransformer);  
073 }  
074  
075 /\*\*  
076 \* Factory method to create a transforming MultiValuedMap that will  
077 \* transform existing contents of the specified map.  
078 \* <p>  
079 \* If there are any elements already in the map being decorated, they will  
080 \* be transformed by this method. Contrast this with  
081 \* {@link #transformingMap(MultiValuedMap, Transformer, Transformer)}.  
082 \*  
083 \* @param <K> the key type  
084 \* @param <V> the value type  
085 \* @param map the MultiValuedMap to decorate, may not be null  
086 \* @param keyTransformer the transformer to use for key conversion, null means no conversion  
087 \* @param valueTransformer the transformer to use for value conversion, null means no conversion  
088 \* @return a new transformed MultiValuedMap  
089 \* @throws NullPointerException if map is null  
090 \*/  
091 public static <K, V> TransformedMultiValuedMap<K, V> transformedMap(final MultiValuedMap<K, V> map,  
092 final Transformer<? super K, ? extends K> keyTransformer,  
093 final Transformer<? super V, ? extends V> valueTransformer) {  
094 final TransformedMultiValuedMap<K, V> decorated =  
095 new TransformedMultiValuedMap<>(map, keyTransformer, valueTransformer);  
096 if (!map.isEmpty()) {  
097 final MultiValuedMap<K, V> mapCopy = new ArrayListValuedHashMap<>(map);  
098 decorated.clear();  
099 decorated.putAll(mapCopy);  
100 }  
101 return decorated;  
102 }  
103  
104 // -----------------------------------------------------------------------  
105 /\*\*  
106 \* Constructor that wraps (not copies).  
107 \* <p>  
108 \* If there are any elements already in the collection being decorated, they  
109 \* are NOT transformed.  
110 \*  
111 \* @param map the MultiValuedMap to decorate, may not be null  
112 \* @param keyTransformer the transformer to use for key conversion, null means no conversion  
113 \* @param valueTransformer the transformer to use for value conversion, null means no conversion  
114 \* @throws NullPointerException if map is null  
115 \*/  
116 protected TransformedMultiValuedMap(final MultiValuedMap<K, V> map,  
117 final Transformer<? super K, ? extends K> keyTransformer,  
118 final Transformer<? super V, ? extends V> valueTransformer) {  
119 super(map);  
120 this.keyTransformer = keyTransformer;  
121 this.valueTransformer = valueTransformer;  
122 }  
123  
124 /\*\*  
125 \* Transforms a key.  
126 \* <p>  
127 \* The transformer itself may throw an exception if necessary.  
128 \*  
129 \* @param object the object to transform  
130 \* @return the transformed object  
131 \*/  
132 protected K transformKey(final K object) {  
133 if (keyTransformer == null) {  
134 return object;  
135 }  
136 return keyTransformer.transform(object);  
137 }  
138  
139 /\*\*  
140 \* Transforms a value.  
141 \* <p>  
142 \* The transformer itself may throw an exception if necessary.  
143 \*  
144 \* @param object the object to transform  
145 \* @return the transformed object  
146 \*/  
147 protected V transformValue(final V object) {  
148 if (valueTransformer == null) {  
149 return object;  
150 }  
151 return valueTransformer.transform(object);  
152 }  
153  
154 @Override  
155 public boolean put(final K key, final V value) {  
156 return decorated().put(transformKey(key), transformValue(value));  
157 }  
158  
159 @Override  
160 public boolean putAll(final K key, final Iterable<? extends V> values) {  
161 if (values == null) {  
162 throw new NullPointerException("Values must not be null.");  
163 }  
164  
165 final Iterable<V> transformedValues = FluentIterable.of(values).transform(valueTransformer);  
166 final Iterator<? extends V> it = transformedValues.iterator();  
167 return it.hasNext() && CollectionUtils.addAll(decorated().get(transformKey(key)), it);  
168 }  
169  
170 @Override  
171 public boolean putAll(final Map<? extends K, ? extends V> map) {  
172 if (map == null) {  
173 throw new NullPointerException("Map must not be null.");  
174 }  
175 boolean changed = false;  
176 for (final Map.Entry<? extends K, ? extends V> entry : map.entrySet()) {  
177 changed |= put(entry.getKey(), entry.getValue());  
178 }  
179 return changed;  
180 }  
181  
182 @Override  
183 public boolean putAll(final MultiValuedMap<? extends K, ? extends V> map) {  
184 if (map == null) {  
185 throw new NullPointerException("Map must not be null.");  
186 }  
187 boolean changed = false;  
188 for (final Map.Entry<? extends K, ? extends V> entry : map.entries()) {  
189 changed |= put(entry.getKey(), entry.getValue());  
190 }  
191 return changed;  
192 }  
193  
194}