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.io.IOException;  
020import java.io.ObjectInputStream;  
021import java.io.ObjectOutputStream;  
022import java.io.Serializable;  
023import java.util.Map;  
024  
025import org.apache.commons.collections4.Transformer;  
026  
027/\*\*  
028 \* Decorates another <code>Map</code> to transform objects that are added.  
029 \* <p>  
030 \* The Map put methods and Map.Entry setValue method are affected by this class.  
031 \* Thus objects must be removed or searched for using their transformed form.  
032 \* For example, if the transformation converts Strings to Integers, you must  
033 \* use the Integer form to remove objects.  
034 \* </p>  
035 \* <p>  
036 \* <strong>Note that TransformedMap is not synchronized and is not thread-safe.</strong>  
037 \* If you wish to use this map from multiple threads concurrently, you must use  
038 \* appropriate synchronization. The simplest approach is to wrap this map  
039 \* using {@link java.util.Collections#synchronizedMap(Map)}. This class may throw  
040 \* exceptions when accessed by concurrent threads without synchronization.  
041 \* </p>  
042 \* <p>  
043 \* This class is Serializable from Commons Collections 3.1.  
044 \* </p>  
045 \*  
046 \* @param <K> the type of the keys in this map  
047 \* @param <V> the type of the values in this map  
048 \*  
049 \* @see org.apache.commons.collections4.splitmap.TransformedSplitMap  
050 \* @since 3.0  
051 \*/  
052public class TransformedMap<K, V>  
053 extends AbstractInputCheckedMapDecorator<K, V>  
054 implements Serializable {  
055  
056 /\*\* Serialization version \*/  
057 private static final long serialVersionUID = 7023152376788900464L;  
058  
059 /\*\* The transformer to use for the key \*/  
060 protected final Transformer<? super K, ? extends K> keyTransformer;  
061 /\*\* The transformer to use for the value \*/  
062 protected final Transformer<? super V, ? extends V> valueTransformer;  
063  
064 /\*\*  
065 \* Factory method to create a transforming map.  
066 \* <p>  
067 \* If there are any elements already in the map being decorated, they  
068 \* are NOT transformed.  
069 \* Contrast this with {@link #transformedMap(Map, Transformer, Transformer)}.  
070 \*  
071 \* @param <K> the key type  
072 \* @param <V> the value type  
073 \* @param map the map to decorate, must not be null  
074 \* @param keyTransformer the transformer to use for key conversion, null means no transformation  
075 \* @param valueTransformer the transformer to use for value conversion, null means no transformation  
076 \* @return a new transformed map  
077 \* @throws NullPointerException if map is null  
078 \* @since 4.0  
079 \*/  
080 public static <K, V> TransformedMap<K, V> transformingMap(final Map<K, V> map,  
081 final Transformer<? super K, ? extends K> keyTransformer,  
082 final Transformer<? super V, ? extends V> valueTransformer) {  
083 return new TransformedMap<>(map, keyTransformer, valueTransformer);  
084 }  
085  
086 /\*\*  
087 \* Factory method to create a transforming map that will transform  
088 \* existing contents of the specified map.  
089 \* <p>  
090 \* If there are any elements already in the map being decorated, they  
091 \* will be transformed by this method.  
092 \* Contrast this with {@link #transformingMap(Map, Transformer, Transformer)}.  
093 \*  
094 \* @param <K> the key type  
095 \* @param <V> the value type  
096 \* @param map the map to decorate, must not be null  
097 \* @param keyTransformer the transformer to use for key conversion, null means no transformation  
098 \* @param valueTransformer the transformer to use for value conversion, null means no transformation  
099 \* @return a new transformed map  
100 \* @throws NullPointerException if map is null  
101 \* @since 4.0  
102 \*/  
103 public static <K, V> TransformedMap<K, V> transformedMap(final Map<K, V> map,  
104 final Transformer<? super K, ? extends K> keyTransformer,  
105 final Transformer<? super V, ? extends V> valueTransformer) {  
106 final TransformedMap<K, V> decorated = new TransformedMap<>(map, keyTransformer, valueTransformer);  
107 if (map.size() > 0) {  
108 final Map<K, V> transformed = decorated.transformMap(map);  
109 decorated.clear();  
110 decorated.decorated().putAll(transformed); // avoids double transformation  
111 }  
112 return decorated;  
113 }  
114  
115 //-----------------------------------------------------------------------  
116 /\*\*  
117 \* Constructor that wraps (not copies).  
118 \* <p>  
119 \* If there are any elements already in the collection being decorated, they  
120 \* are NOT transformed.  
121 \*  
122 \* @param map the map to decorate, must not be null  
123 \* @param keyTransformer the transformer to use for key conversion, null means no conversion  
124 \* @param valueTransformer the transformer to use for value conversion, null means no conversion  
125 \* @throws NullPointerException if map is null  
126 \*/  
127 protected TransformedMap(final Map<K, V> map, final Transformer<? super K, ? extends K> keyTransformer,  
128 final Transformer<? super V, ? extends V> valueTransformer) {  
129 super(map);  
130 this.keyTransformer = keyTransformer;  
131 this.valueTransformer = valueTransformer;  
132 }  
133  
134 //-----------------------------------------------------------------------  
135 /\*\*  
136 \* Write the map out using a custom routine.  
137 \*  
138 \* @param out the output stream  
139 \* @throws IOException if an error occurs while writing to the stream  
140 \* @since 3.1  
141 \*/  
142 private void writeObject(final ObjectOutputStream out) throws IOException {  
143 out.defaultWriteObject();  
144 out.writeObject(map);  
145 }  
146  
147 /\*\*  
148 \* Read the map in using a custom routine.  
149 \*  
150 \* @param in the input stream  
151 \* @throws IOException if an error occurs while reading from the stream  
152 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
153 \* @since 3.1  
154 \*/  
155 @SuppressWarnings("unchecked") // (1) should only fail if input stream is incorrect  
156 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
157 in.defaultReadObject();  
158 map = (Map<K, V>) in.readObject(); // (1)  
159 }  
160  
161 //-----------------------------------------------------------------------  
162 /\*\*  
163 \* Transforms a key.  
164 \* <p>  
165 \* The transformer itself may throw an exception if necessary.  
166 \*  
167 \* @param object the object to transform  
168 \* @return the transformed object  
169 \*/  
170 protected K transformKey(final K object) {  
171 if (keyTransformer == null) {  
172 return object;  
173 }  
174 return keyTransformer.transform(object);  
175 }  
176  
177 /\*\*  
178 \* Transforms a value.  
179 \* <p>  
180 \* The transformer itself may throw an exception if necessary.  
181 \*  
182 \* @param object the object to transform  
183 \* @return the transformed object  
184 \*/  
185 protected V transformValue(final V object) {  
186 if (valueTransformer == null) {  
187 return object;  
188 }  
189 return valueTransformer.transform(object);  
190 }  
191  
192 /\*\*  
193 \* Transforms a map.  
194 \* <p>  
195 \* The transformer itself may throw an exception if necessary.  
196 \*  
197 \* @param map the map to transform  
198 \* @return the transformed object  
199 \*/  
200 @SuppressWarnings("unchecked")  
201 protected Map<K, V> transformMap(final Map<? extends K, ? extends V> map) {  
202 if (map.isEmpty()) {  
203 return (Map<K, V>) map;  
204 }  
205 final Map<K, V> result = new LinkedMap<>(map.size());  
206  
207 for (final Map.Entry<? extends K, ? extends V> entry : map.entrySet()) {  
208 result.put(transformKey(entry.getKey()), transformValue(entry.getValue()));  
209 }  
210 return result;  
211 }  
212  
213 /\*\*  
214 \* Override to transform the value when using <code>setValue</code>.  
215 \*  
216 \* @param value the value to transform  
217 \* @return the transformed value  
218 \* @since 3.1  
219 \*/  
220 @Override  
221 protected V checkSetValue(final V value) {  
222 return valueTransformer.transform(value);  
223 }  
224  
225 /\*\*  
226 \* Override to only return true when there is a value transformer.  
227 \*  
228 \* @return true if a value transformer is in use  
229 \* @since 3.1  
230 \*/  
231 @Override  
232 protected boolean isSetValueChecking() {  
233 return valueTransformer != null;  
234 }  
235  
236 //-----------------------------------------------------------------------  
237 @Override  
238 public V put(K key, V value) {  
239 key = transformKey(key);  
240 value = transformValue(value);  
241 return decorated().put(key, value);  
242 }  
243  
244 @Override  
245 public void putAll(Map<? extends K, ? extends V> mapToCopy) {  
246 mapToCopy = transformMap(mapToCopy);  
247 decorated().putAll(mapToCopy);  
248 }  
249  
250}