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.splitmap;  
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.Put;  
026import org.apache.commons.collections4.Transformer;  
027import org.apache.commons.collections4.map.LinkedMap;  
028  
029/\*\*  
030 \* Decorates another {@link Map} to transform objects that are added.  
031 \* <p>  
032 \* The Map put methods and Map.Entry setValue method are affected by this class.  
033 \* Thus objects must be removed or searched for using their transformed form.  
034 \* For example, if the transformation converts Strings to Integers, you must use  
035 \* the Integer form to remove objects.  
036 \* </p>  
037 \* <p>  
038 \* <strong>Note that TransformedMap is not synchronized and is not  
039 \* thread-safe.</strong> If you wish to use this map from multiple threads  
040 \* concurrently, you must use appropriate synchronization. The simplest approach  
041 \* is to wrap this map using {@link java.util.Collections#synchronizedMap(Map)}.  
042 \* This class may throw exceptions when accessed by concurrent threads without  
043 \* synchronization.  
044 \* </p>  
045 \* <p>  
046 \* The "put" and "get" type constraints of this class are mutually independent;  
047 \* contrast with {@link org.apache.commons.collections4.map.TransformedMap} which,  
048 \* by virtue of its implementing {@link Map}<K, V>, must be constructed in such  
049 \* a way that its read and write parameters are generalized to a common (super-)type.  
050 \* In practice this would often mean <code>>Object, Object></code>, defeating  
051 \* much of the usefulness of having parameterized types.  
052 \* </p>  
053 \* <p>  
054 \* On the downside, this class is not drop-in compatible with {@link java.util.Map}  
055 \* but is intended to be worked with either directly or by {@link Put} and  
056 \* {@link org.apache.commons.collections4.Get Get} generalizations.  
057 \* </p>  
058 \*  
059 \* @param <J> the type of the keys to put in this map  
060 \* @param <K> the type of the keys to get in this map  
061 \* @param <U> the type of the values to put in this map  
062 \* @param <V> the type of the values to get in this map  
063 \* @since 4.0  
064 \*  
065 \* @see org.apache.commons.collections4.SplitMapUtils#readableMap(org.apache.commons.collections4.Get)  
066 \* @see org.apache.commons.collections4.SplitMapUtils#writableMap(Put)  
067 \*/  
068public class TransformedSplitMap<J, K, U, V> extends AbstractIterableGetMapDecorator<K, V>  
069 implements Put<J, U>, Serializable {  
070  
071 /\*\* Serialization version \*/  
072 private static final long serialVersionUID = 5966875321133456994L;  
073  
074 /\*\* The transformer to use for the key \*/  
075 private final Transformer<? super J, ? extends K> keyTransformer;  
076 /\*\* The transformer to use for the value \*/  
077 private final Transformer<? super U, ? extends V> valueTransformer;  
078  
079 /\*\*  
080 \* Factory method to create a transforming map.  
081 \* <p>  
082 \* If there are any elements already in the map being decorated, they are  
083 \* NOT transformed.  
084 \*  
085 \* @param <J> the input key type  
086 \* @param <K> the output key type  
087 \* @param <U> the input value type  
088 \* @param <V> the output value type  
089 \* @param map the map to decorate, must not be null  
090 \* @param keyTransformer the transformer to use for key conversion, must not be null  
091 \* @param valueTransformer the transformer to use for value conversion, must not be null  
092 \* @return a new transformed map  
093 \* @throws NullPointerException if map or either of the transformers is null  
094 \*/  
095 public static <J, K, U, V> TransformedSplitMap<J, K, U, V> transformingMap(final Map<K, V> map,  
096 final Transformer<? super J, ? extends K> keyTransformer,  
097 final Transformer<? super U, ? extends V> valueTransformer) {  
098 return new TransformedSplitMap<>(map, keyTransformer, valueTransformer);  
099 }  
100  
101 //-----------------------------------------------------------------------  
102 /\*\*  
103 \* Constructor that wraps (not copies).  
104 \* <p>  
105 \* If there are any elements already in the collection being decorated, they  
106 \* are NOT transformed.  
107 \*  
108 \* @param map the map to decorate, must not be null  
109 \* @param keyTransformer the transformer to use for key conversion, must not be null  
110 \* @param valueTransformer the transformer to use for value conversion, must not be null  
111 \* @throws NullPointerException if map or either of the transformers is null  
112 \*/  
113 protected TransformedSplitMap(final Map<K, V> map, final Transformer<? super J, ? extends K> keyTransformer,  
114 final Transformer<? super U, ? extends V> valueTransformer) {  
115 super(map);  
116 if (keyTransformer == null) {  
117 throw new NullPointerException("KeyTransformer must not be null.");  
118 }  
119 this.keyTransformer = keyTransformer;  
120 if (valueTransformer == null) {  
121 throw new NullPointerException("ValueTransformer must not be null.");  
122 }  
123 this.valueTransformer = valueTransformer;  
124 }  
125  
126 //-----------------------------------------------------------------------  
127 /\*\*  
128 \* Write the map out using a custom routine.  
129 \*  
130 \* @param out the output stream  
131 \* @throws IOException if an error occurs while writing to the stream  
132 \*/  
133 private void writeObject(final ObjectOutputStream out) throws IOException {  
134 out.defaultWriteObject();  
135 out.writeObject(decorated());  
136 }  
137  
138 /\*\*  
139 \* Read the map in using a custom routine.  
140 \*  
141 \* @param in the input stream  
142 \* @throws IOException if an error occurs while reading from the stream  
143 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
144 \* @since 3.1  
145 \*/  
146 @SuppressWarnings("unchecked") // (1) should only fail if input stream is incorrect  
147 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
148 in.defaultReadObject();  
149 map = (Map<K, V>) in.readObject(); // (1)  
150 }  
151  
152 //-----------------------------------------------------------------------  
153 /\*\*  
154 \* Transforms a key.  
155 \* <p>  
156 \* The transformer itself may throw an exception if necessary.  
157 \*  
158 \* @param object the object to transform  
159 \* @return the transformed object  
160 \*/  
161 protected K transformKey(final J object) {  
162 return keyTransformer.transform(object);  
163 }  
164  
165 /\*\*  
166 \* Transforms a value.  
167 \* <p>  
168 \* The transformer itself may throw an exception if necessary.  
169 \*  
170 \* @param object the object to transform  
171 \* @return the transformed object  
172 \*/  
173 protected V transformValue(final U object) {  
174 return valueTransformer.transform(object);  
175 }  
176  
177 /\*\*  
178 \* Transforms a map.  
179 \* <p>  
180 \* The transformer itself may throw an exception if necessary.  
181 \*  
182 \* @param map the map to transform  
183 \* @return the transformed object  
184 \*/  
185 @SuppressWarnings("unchecked")  
186 protected Map<K, V> transformMap(final Map<? extends J, ? extends U> map) {  
187 if (map.isEmpty()) {  
188 return (Map<K, V>) map;  
189 }  
190 final Map<K, V> result = new LinkedMap<>(map.size());  
191  
192 for (final Map.Entry<? extends J, ? extends U> entry : map.entrySet()) {  
193 result.put(transformKey(entry.getKey()), transformValue(entry.getValue()));  
194 }  
195 return result;  
196 }  
197  
198 /\*\*  
199 \* Override to transform the value when using <code>setValue</code>.  
200 \*  
201 \* @param value the value to transform  
202 \* @return the transformed value  
203 \*/  
204 protected V checkSetValue(final U value) {  
205 return valueTransformer.transform(value);  
206 }  
207  
208 //-----------------------------------------------------------------------  
209 @Override  
210 public V put(final J key, final U value) {  
211 return decorated().put(transformKey(key), transformValue(value));  
212 }  
213  
214 @Override  
215 public void putAll(final Map<? extends J, ? extends U> mapToCopy) {  
216 decorated().putAll(transformMap(mapToCopy));  
217 }  
218  
219 @Override  
220 public void clear() {  
221 decorated().clear();  
222 }  
223}