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.io.Serializable;  
020import java.util.Collection;  
021import java.util.Map;  
022import java.util.Map.Entry;  
023import java.util.Set;  
024  
025import org.apache.commons.collections4.MapIterator;  
026import org.apache.commons.collections4.MultiSet;  
027import org.apache.commons.collections4.MultiValuedMap;  
028  
029/\*\*  
030 \* Decorates another <code>MultiValuedMap</code> to provide additional behaviour.  
031 \* <p>  
032 \* Each method call made on this <code>MultiValuedMap</code> is forwarded to the  
033 \* decorated <code>MultiValuedMap</code>. This class is used as a framework to build  
034 \* to extensions such as synchronized and unmodifiable behaviour.  
035 \* </p>  
036 \*  
037 \* @param <K> the type of key elements  
038 \* @param <V> the type of value elements  
039 \*  
040 \* @since 4.1  
041 \*/  
042public abstract class AbstractMultiValuedMapDecorator<K, V>  
043 implements MultiValuedMap<K, V>, Serializable {  
044  
045 /\*\* Serialization version \*/  
046 private static final long serialVersionUID = 20150612L;  
047  
048 /\*\* MultiValuedMap to decorate \*/  
049 private final MultiValuedMap<K, V> map;  
050  
051 /\*\*  
052 \* Constructor that wraps (not copies).  
053 \*  
054 \* @param map the map to decorate, must not be null  
055 \* @throws NullPointerException if the map is null  
056 \*/  
057 protected AbstractMultiValuedMapDecorator(final MultiValuedMap<K, V> map) {  
058 if (map == null) {  
059 throw new NullPointerException("MultiValuedMap must not be null.");  
060 }  
061 this.map = map;  
062 }  
063  
064 // -----------------------------------------------------------------------  
065 /\*\*  
066 \* The decorated multi-valued map.  
067 \*  
068 \* @return the map to decorate  
069 \*/  
070 protected MultiValuedMap<K, V> decorated() {  
071 return map;  
072 }  
073  
074 // -----------------------------------------------------------------------  
075 @Override  
076 public int size() {  
077 return decorated().size();  
078 }  
079  
080 @Override  
081 public boolean isEmpty() {  
082 return decorated().isEmpty();  
083 }  
084  
085 @Override  
086 public boolean containsKey(final Object key) {  
087 return decorated().containsKey(key);  
088 }  
089  
090 @Override  
091 public boolean containsValue(final Object value) {  
092 return decorated().containsValue(value);  
093 }  
094  
095 @Override  
096 public boolean containsMapping(final Object key, final Object value) {  
097 return decorated().containsMapping(key, value);  
098 }  
099  
100 @Override  
101 public Collection<V> get(final K key) {  
102 return decorated().get(key);  
103 }  
104  
105 @Override  
106 public Collection<V> remove(final Object key) {  
107 return decorated().remove(key);  
108 }  
109  
110 @Override  
111 public boolean removeMapping(final Object key, final Object item) {  
112 return decorated().removeMapping(key, item);  
113 }  
114  
115 @Override  
116 public void clear() {  
117 decorated().clear();  
118 }  
119  
120 @Override  
121 public boolean put(final K key, final V value) {  
122 return decorated().put(key, value);  
123 }  
124  
125 @Override  
126 public Set<K> keySet() {  
127 return decorated().keySet();  
128 }  
129  
130 @Override  
131 public Collection<Entry<K, V>> entries() {  
132 return decorated().entries();  
133 }  
134  
135 @Override  
136 public MultiSet<K> keys() {  
137 return decorated().keys();  
138 }  
139  
140 @Override  
141 public Collection<V> values() {  
142 return decorated().values();  
143 }  
144  
145 @Override  
146 public Map<K, Collection<V>> asMap() {  
147 return decorated().asMap();  
148 }  
149  
150 @Override  
151 public boolean putAll(final K key, final Iterable<? extends V> values) {  
152 return decorated().putAll(key, values);  
153 }  
154  
155 @Override  
156 public boolean putAll(final Map<? extends K, ? extends V> map) {  
157 return decorated().putAll(map);  
158 }  
159  
160 @Override  
161 public boolean putAll(final MultiValuedMap<? extends K, ? extends V> map) {  
162 return decorated().putAll(map);  
163 }  
164  
165 @Override  
166 public MapIterator<K, V> mapIterator() {  
167 return decorated().mapIterator();  
168 }  
169  
170 @Override  
171 public boolean equals(final Object object) {  
172 if (object == this) {  
173 return true;  
174 }  
175 return decorated().equals(object);  
176 }  
177  
178 @Override  
179 public int hashCode() {  
180 return decorated().hashCode();  
181 }  
182  
183 @Override  
184 public String toString() {  
185 return decorated().toString();  
186 }  
187  
188}