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.Collection;  
020import java.util.Collections;  
021import java.util.List;  
022import java.util.ListIterator;  
023import java.util.Map;  
024  
025import org.apache.commons.collections4.ListUtils;  
026import org.apache.commons.collections4.ListValuedMap;  
027  
028/\*\*  
029 \* Abstract implementation of the {@link ListValuedMap} interface to simplify  
030 \* the creation of subclass implementations.  
031 \* <p>  
032 \* Subclasses specify a Map implementation to use as the internal storage and  
033 \* the List implementation to use as values.  
034 \* </p>  
035 \*  
036 \* @param <K> the type of the keys in this map  
037 \* @param <V> the type of the values in this map  
038 \* @since 4.1  
039 \*/  
040public abstract class AbstractListValuedMap<K, V> extends AbstractMultiValuedMap<K, V>  
041 implements ListValuedMap<K, V> {  
042  
043 /\*\*  
044 \* Constructor needed for subclass serialisation.  
045 \*/  
046 protected AbstractListValuedMap() {  
047 super();  
048 }  
049  
050 /\*\*  
051 \* A constructor that wraps, not copies  
052 \*  
053 \* @param map the map to wrap, must not be null  
054 \* @throws NullPointerException if the map is null  
055 \*/  
056 protected AbstractListValuedMap(final Map<K, ? extends List<V>> map) {  
057 super(map);  
058 }  
059  
060 // -----------------------------------------------------------------------  
061 @Override  
062 @SuppressWarnings("unchecked")  
063 protected Map<K, List<V>> getMap() {  
064 return (Map<K, List<V>>) super.getMap();  
065 }  
066  
067 /\*\*  
068 \* Creates a new value collection using the provided factory.  
069 \* @return a new list  
070 \*/  
071 @Override  
072 protected abstract List<V> createCollection();  
073  
074 // -----------------------------------------------------------------------  
075 /\*\*  
076 \* Gets the list of values associated with the specified key. This would  
077 \* return an empty list in case the mapping is not present  
078 \*  
079 \* @param key the key to retrieve  
080 \* @return the {@code List} of values, will return an empty {@link List} for no mapping  
081 \*/  
082 @Override  
083 public List<V> get(final K key) {  
084 return wrappedCollection(key);  
085 }  
086  
087 @Override  
088 List<V> wrappedCollection(final K key) {  
089 return new WrappedList(key);  
090 }  
091  
092 /\*\*  
093 \* Removes all values associated with the specified key.  
094 \* <p>  
095 \* A subsequent <code>get(Object)</code> would return an empty list.  
096 \*  
097 \* @param key the key to remove values from  
098 \* @return the <code>List</code> of values removed, will return an empty,  
099 \* unmodifiable list for no mapping found.  
100 \*/  
101 @Override  
102 public List<V> remove(final Object key) {  
103 return ListUtils.emptyIfNull(getMap().remove(key));  
104 }  
105  
106 // -----------------------------------------------------------------------  
107 /\*\*  
108 \* Wrapped list to handle add and remove on the list returned by get(object)  
109 \*/  
110 private class WrappedList extends WrappedCollection implements List<V> {  
111  
112 public WrappedList(final K key) {  
113 super(key);  
114 }  
115  
116 @Override  
117 protected List<V> getMapping() {  
118 return getMap().get(key);  
119 }  
120  
121 @Override  
122 public void add(final int index, final V value) {  
123 List<V> list = getMapping();  
124 if (list == null) {  
125 list = createCollection();  
126 getMap().put(key, list);  
127 }  
128 list.add(index, value);  
129 }  
130  
131 @Override  
132 public boolean addAll(final int index, final Collection<? extends V> c) {  
133 List<V> list = getMapping();  
134 if (list == null) {  
135 list = createCollection();  
136 final boolean changed = list.addAll(index, c);  
137 if (changed) {  
138 getMap().put(key, list);  
139 }  
140 return changed;  
141 }  
142 return list.addAll(index, c);  
143 }  
144  
145 @Override  
146 public V get(final int index) {  
147 final List<V> list = ListUtils.emptyIfNull(getMapping());  
148 return list.get(index);  
149 }  
150  
151 @Override  
152 public int indexOf(final Object o) {  
153 final List<V> list = ListUtils.emptyIfNull(getMapping());  
154 return list.indexOf(o);  
155 }  
156  
157 @Override  
158 public int lastIndexOf(final Object o) {  
159 final List<V> list = ListUtils.emptyIfNull(getMapping());  
160 return list.lastIndexOf(o);  
161 }  
162  
163 @Override  
164 public ListIterator<V> listIterator() {  
165 return new ValuesListIterator(key);  
166 }  
167  
168 @Override  
169 public ListIterator<V> listIterator(final int index) {  
170 return new ValuesListIterator(key, index);  
171 }  
172  
173 @Override  
174 public V remove(final int index) {  
175 final List<V> list = ListUtils.emptyIfNull(getMapping());  
176 final V value = list.remove(index);  
177 if (list.isEmpty()) {  
178 AbstractListValuedMap.this.remove(key);  
179 }  
180 return value;  
181 }  
182  
183 @Override  
184 public V set(final int index, final V value) {  
185 final List<V> list = ListUtils.emptyIfNull(getMapping());  
186 return list.set(index, value);  
187 }  
188  
189 @Override  
190 public List<V> subList(final int fromIndex, final int toIndex) {  
191 final List<V> list = ListUtils.emptyIfNull(getMapping());  
192 return list.subList(fromIndex, toIndex);  
193 }  
194  
195 @Override  
196 public boolean equals(final Object other) {  
197 final List<V> list = getMapping();  
198 if (list == null) {  
199 return Collections.emptyList().equals(other);  
200 }  
201 if (!(other instanceof List)) {  
202 return false;  
203 }  
204 final List<?> otherList = (List<?>) other;  
205 return ListUtils.isEqualList(list, otherList);  
206 }  
207  
208 @Override  
209 public int hashCode() {  
210 final List<V> list = getMapping();  
211 return ListUtils.hashCodeForList(list);  
212 }  
213  
214 }  
215  
216 /\*\* Values ListIterator \*/  
217 private class ValuesListIterator implements ListIterator<V> {  
218  
219 private final K key;  
220 private List<V> values;  
221 private ListIterator<V> iterator;  
222  
223 public ValuesListIterator(final K key) {  
224 this.key = key;  
225 this.values = ListUtils.emptyIfNull(getMap().get(key));  
226 this.iterator = values.listIterator();  
227 }  
228  
229 public ValuesListIterator(final K key, final int index) {  
230 this.key = key;  
231 this.values = ListUtils.emptyIfNull(getMap().get(key));  
232 this.iterator = values.listIterator(index);  
233 }  
234  
235 @Override  
236 public void add(final V value) {  
237 if (getMap().get(key) == null) {  
238 final List<V> list = createCollection();  
239 getMap().put(key, list);  
240 this.values = list;  
241 this.iterator = list.listIterator();  
242 }  
243 this.iterator.add(value);  
244 }  
245  
246 @Override  
247 public boolean hasNext() {  
248 return iterator.hasNext();  
249 }  
250  
251 @Override  
252 public boolean hasPrevious() {  
253 return iterator.hasPrevious();  
254 }  
255  
256 @Override  
257 public V next() {  
258 return iterator.next();  
259 }  
260  
261 @Override  
262 public int nextIndex() {  
263 return iterator.nextIndex();  
264 }  
265  
266 @Override  
267 public V previous() {  
268 return iterator.previous();  
269 }  
270  
271 @Override  
272 public int previousIndex() {  
273 return iterator.previousIndex();  
274 }  
275  
276 @Override  
277 public void remove() {  
278 iterator.remove();  
279 if (values.isEmpty()) {  
280 getMap().remove(key);  
281 }  
282 }  
283  
284 @Override  
285 public void set(final V value) {  
286 iterator.set(value);  
287 }  
288  
289 }  
290  
291}