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.IOException;  
020import java.io.ObjectInputStream;  
021import java.io.ObjectOutputStream;  
022import java.util.AbstractCollection;  
023import java.util.AbstractMap;  
024import java.util.AbstractSet;  
025import java.util.ArrayList;  
026import java.util.Collection;  
027import java.util.Iterator;  
028import java.util.Map;  
029import java.util.Map.Entry;  
030import java.util.Set;  
031  
032import org.apache.commons.collections4.CollectionUtils;  
033import org.apache.commons.collections4.IteratorUtils;  
034import org.apache.commons.collections4.MapIterator;  
035import org.apache.commons.collections4.MultiSet;  
036import org.apache.commons.collections4.MultiValuedMap;  
037import org.apache.commons.collections4.Transformer;  
038import org.apache.commons.collections4.iterators.AbstractIteratorDecorator;  
039import org.apache.commons.collections4.iterators.EmptyMapIterator;  
040import org.apache.commons.collections4.iterators.IteratorChain;  
041import org.apache.commons.collections4.iterators.LazyIteratorChain;  
042import org.apache.commons.collections4.iterators.TransformIterator;  
043import org.apache.commons.collections4.keyvalue.AbstractMapEntry;  
044import org.apache.commons.collections4.keyvalue.UnmodifiableMapEntry;  
045import org.apache.commons.collections4.multiset.AbstractMultiSet;  
046import org.apache.commons.collections4.multiset.UnmodifiableMultiSet;  
047  
048/\*\*  
049 \* Abstract implementation of the {@link MultiValuedMap} interface to simplify  
050 \* the creation of subclass implementations.  
051 \* <p>  
052 \* Subclasses specify a Map implementation to use as the internal storage.  
053 \* </p>  
054 \*  
055 \* @param <K> the type of the keys in this map  
056 \* @param <V> the type of the values in this map  
057 \* @since 4.1  
058 \*/  
059public abstract class AbstractMultiValuedMap<K, V> implements MultiValuedMap<K, V> {  
060  
061 /\*\* The values view \*/  
062 private transient Collection<V> valuesView;  
063  
064 /\*\* The EntryValues view \*/  
065 private transient EntryValues entryValuesView;  
066  
067 /\*\* The KeyMultiSet view \*/  
068 private transient MultiSet<K> keysMultiSetView;  
069  
070 /\*\* The AsMap view \*/  
071 private transient AsMap asMapView;  
072  
073 /\*\* The map used to store the data \*/  
074 private transient Map<K, Collection<V>> map;  
075  
076 /\*\*  
077 \* Constructor needed for subclass serialisation.  
078 \*/  
079 protected AbstractMultiValuedMap() {  
080 super();  
081 }  
082  
083 /\*\*  
084 \* Constructor that wraps (not copies).  
085 \*  
086 \* @param map the map to wrap, must not be null  
087 \* @throws NullPointerException if the map is null  
088 \*/  
089 @SuppressWarnings("unchecked")  
090 protected AbstractMultiValuedMap(final Map<K, ? extends Collection<V>> map) {  
091 if (map == null) {  
092 throw new NullPointerException("Map must not be null.");  
093 }  
094 this.map = (Map<K, Collection<V>>) map;  
095 }  
096  
097 // -----------------------------------------------------------------------  
098 /\*\*  
099 \* Gets the map being wrapped.  
100 \*  
101 \* @return the wrapped map  
102 \*/  
103 protected Map<K, ? extends Collection<V>> getMap() {  
104 return map;  
105 }  
106  
107 /\*\*  
108 \* Sets the map being wrapped.  
109 \* <p>  
110 \* <b>NOTE:</b> this method should only be used during deserialization  
111 \*  
112 \* @param map the map to wrap  
113 \*/  
114 @SuppressWarnings("unchecked")  
115 protected void setMap(final Map<K, ? extends Collection<V>> map) {  
116 this.map = (Map<K, Collection<V>>) map;  
117 }  
118  
119 protected abstract Collection<V> createCollection();  
120  
121 // -----------------------------------------------------------------------  
122 @Override  
123 public boolean containsKey(final Object key) {  
124 return getMap().containsKey(key);  
125 }  
126  
127 @Override  
128 public boolean containsValue(final Object value) {  
129 return values().contains(value);  
130 }  
131  
132 @Override  
133 public boolean containsMapping(final Object key, final Object value) {  
134 final Collection<V> coll = getMap().get(key);  
135 return coll != null && coll.contains(value);  
136 }  
137  
138 @Override  
139 public Collection<Entry<K, V>> entries() {  
140 return entryValuesView != null ? entryValuesView : (entryValuesView = new EntryValues());  
141 }  
142  
143 /\*\*  
144 \* Gets the collection of values associated with the specified key. This  
145 \* would return an empty collection in case the mapping is not present  
146 \*  
147 \* @param key the key to retrieve  
148 \* @return the {@code Collection} of values, will return an empty {@code Collection} for no mapping  
149 \*/  
150 @Override  
151 public Collection<V> get(final K key) {  
152 return wrappedCollection(key);  
153 }  
154  
155 Collection<V> wrappedCollection(final K key) {  
156 return new WrappedCollection(key);  
157 }  
158  
159 /\*\*  
160 \* Removes all values associated with the specified key.  
161 \* <p>  
162 \* A subsequent <code>get(Object)</code> would return an empty collection.  
163 \*  
164 \* @param key the key to remove values from  
165 \* @return the <code>Collection</code> of values removed, will return an  
166 \* empty, unmodifiable collection for no mapping found  
167 \*/  
168 @Override  
169 public Collection<V> remove(final Object key) {  
170 return CollectionUtils.emptyIfNull(getMap().remove(key));  
171 }  
172  
173 /\*\*  
174 \* Removes a specific key/value mapping from the multi-valued map.  
175 \* <p>  
176 \* The value is removed from the collection mapped to the specified key.  
177 \* Other values attached to that key are unaffected.  
178 \* <p>  
179 \* If the last value for a key is removed, an empty collection would be  
180 \* returned from a subsequent {@link #get(Object)}.  
181 \*  
182 \* @param key the key to remove from  
183 \* @param value the value to remove  
184 \* @return true if the mapping was removed, false otherwise  
185 \*/  
186 @Override  
187 public boolean removeMapping(final Object key, final Object value) {  
188 final Collection<V> coll = getMap().get(key);  
189 if (coll == null) {  
190 return false;  
191 }  
192 final boolean changed = coll.remove(value);  
193 if (coll.isEmpty()) {  
194 getMap().remove(key);  
195 }  
196 return changed;  
197 }  
198  
199 @Override  
200 public boolean isEmpty() {  
201 return getMap().isEmpty();  
202 }  
203  
204 @Override  
205 public Set<K> keySet() {  
206 return getMap().keySet();  
207 }  
208  
209 /\*\*  
210 \* {@inheritDoc}  
211 \* <p>  
212 \* This implementation does <b>not</b> cache the total size  
213 \* of the multi-valued map, but rather calculates it by iterating  
214 \* over the entries of the underlying map.  
215 \*/  
216 @Override  
217 public int size() {  
218 // the total size should be cached to improve performance  
219 // but this requires that all modifications of the multimap  
220 // (including the wrapped collections and entry/value  
221 // collections) are tracked.  
222 int size = 0;  
223 for (final Collection<V> col : getMap().values()) {  
224 size += col.size();  
225 }  
226 return size;  
227 }  
228  
229 /\*\*  
230 \* Gets a collection containing all the values in the map.  
231 \* <p>  
232 \* Returns a collection containing all the values from all keys.  
233 \*  
234 \* @return a collection view of the values contained in this map  
235 \*/  
236 @Override  
237 public Collection<V> values() {  
238 final Collection<V> vs = valuesView;  
239 return vs != null ? vs : (valuesView = new Values());  
240 }  
241  
242 @Override  
243 public void clear() {  
244 getMap().clear();  
245 }  
246  
247 /\*\*  
248 \* Adds the value to the collection associated with the specified key.  
249 \* <p>  
250 \* Unlike a normal <code>Map</code> the previous value is not replaced.  
251 \* Instead the new value is added to the collection stored against the key.  
252 \*  
253 \* @param key the key to store against  
254 \* @param value the value to add to the collection at the key  
255 \* @return the value added if the map changed and null if the map did not change  
256 \*/  
257 @Override  
258 public boolean put(final K key, final V value) {  
259 Collection<V> coll = getMap().get(key);  
260 if (coll == null) {  
261 coll = createCollection();  
262 if (coll.add(value)) {  
263 map.put(key, coll);  
264 return true;  
265 }  
266 return false;  
267 }  
268 return coll.add(value);  
269 }  
270  
271 /\*\*  
272 \* Copies all of the mappings from the specified map to this map. The effect  
273 \* of this call is equivalent to that of calling {@link #put(Object,Object)  
274 \* put(k, v)} on this map once for each mapping from key {@code k} to value  
275 \* {@code v} in the specified map. The behavior of this operation is  
276 \* undefined if the specified map is modified while the operation is in  
277 \* progress.  
278 \*  
279 \* @param map mappings to be stored in this map, may not be null  
280 \* @return true if the map changed as a result of this operation  
281 \* @throws NullPointerException if map is null  
282 \*/  
283 @Override  
284 public boolean putAll(final Map<? extends K, ? extends V> map) {  
285 if (map == null) {  
286 throw new NullPointerException("Map must not be null.");  
287 }  
288 boolean changed = false;  
289 for (final Map.Entry<? extends K, ? extends V> entry : map.entrySet()) {  
290 changed |= put(entry.getKey(), entry.getValue());  
291 }  
292 return changed;  
293 }  
294  
295 /\*\*  
296 \* Copies all of the mappings from the specified MultiValuedMap to this map.  
297 \* The effect of this call is equivalent to that of calling  
298 \* {@link #put(Object,Object) put(k, v)} on this map once for each mapping  
299 \* from key {@code k} to value {@code v} in the specified map. The  
300 \* behavior of this operation is undefined if the specified map is modified  
301 \* while the operation is in progress.  
302 \*  
303 \* @param map mappings to be stored in this map, may not be null  
304 \* @return true if the map changed as a result of this operation  
305 \* @throws NullPointerException if map is null  
306 \*/  
307 @Override  
308 public boolean putAll(final MultiValuedMap<? extends K, ? extends V> map) {  
309 if (map == null) {  
310 throw new NullPointerException("Map must not be null.");  
311 }  
312 boolean changed = false;  
313 for (final Map.Entry<? extends K, ? extends V> entry : map.entries()) {  
314 changed |= put(entry.getKey(), entry.getValue());  
315 }  
316 return changed;  
317 }  
318  
319 /\*\*  
320 \* Returns a {@link MultiSet} view of the key mapping contained in this map.  
321 \* <p>  
322 \* Returns a MultiSet of keys with its values count as the count of the MultiSet.  
323 \* This multiset is backed by the map, so any changes in the map is reflected here.  
324 \* Any method which modifies this multiset like {@code add}, {@code remove},  
325 \* {@link Iterator#remove()} etc throws {@code UnsupportedOperationException}.  
326 \*  
327 \* @return a bag view of the key mapping contained in this map  
328 \*/  
329 @Override  
330 public MultiSet<K> keys() {  
331 if (keysMultiSetView == null) {  
332 keysMultiSetView = UnmodifiableMultiSet.unmodifiableMultiSet(new KeysMultiSet());  
333 }  
334 return keysMultiSetView;  
335 }  
336  
337 @Override  
338 public Map<K, Collection<V>> asMap() {  
339 return asMapView != null ? asMapView : (asMapView = new AsMap(map));  
340 }  
341  
342 /\*\*  
343 \* Adds Iterable values to the collection associated with the specified key.  
344 \*  
345 \* @param key the key to store against  
346 \* @param values the values to add to the collection at the key, may not be null  
347 \* @return true if this map changed  
348 \* @throws NullPointerException if values is null  
349 \*/  
350 @Override  
351 public boolean putAll(final K key, final Iterable<? extends V> values) {  
352 if (values == null) {  
353 throw new NullPointerException("Values must not be null.");  
354 }  
355  
356 if (values instanceof Collection<?>) {  
357 final Collection<? extends V> valueCollection = (Collection<? extends V>) values;  
358 return !valueCollection.isEmpty() && get(key).addAll(valueCollection);  
359 }  
360 final Iterator<? extends V> it = values.iterator();  
361 return it.hasNext() && CollectionUtils.addAll(get(key), it);  
362 }  
363  
364 @Override  
365 public MapIterator<K, V> mapIterator() {  
366 if (size() == 0) {  
367 return EmptyMapIterator.emptyMapIterator();  
368 }  
369 return new MultiValuedMapIterator();  
370 }  
371  
372 @Override  
373 public boolean equals(final Object obj) {  
374 if (this == obj) {  
375 return true;  
376 }  
377 if (obj instanceof MultiValuedMap) {  
378 return asMap().equals(((MultiValuedMap<?, ?>) obj).asMap());  
379 }  
380 return false;  
381 }  
382  
383 @Override  
384 public int hashCode() {  
385 return getMap().hashCode();  
386 }  
387  
388 @Override  
389 public String toString() {  
390 return getMap().toString();  
391 }  
392  
393 // -----------------------------------------------------------------------  
394  
395 /\*\*  
396 \* Wrapped collection to handle add and remove on the collection returned  
397 \* by get(object).  
398 \* <p>  
399 \* Currently, the wrapped collection is not cached and has to be retrieved  
400 \* from the underlying map. This is safe, but not very efficient and  
401 \* should be improved in subsequent releases. For this purpose, the  
402 \* scope of this collection is set to package private to simplify later  
403 \* refactoring.  
404 \*/  
405 class WrappedCollection implements Collection<V> {  
406  
407 protected final K key;  
408  
409 public WrappedCollection(final K key) {  
410 this.key = key;  
411 }  
412  
413 protected Collection<V> getMapping() {  
414 return getMap().get(key);  
415 }  
416  
417 @Override  
418 public boolean add(final V value) {  
419 Collection<V> coll = getMapping();  
420 if (coll == null) {  
421 coll = createCollection();  
422 AbstractMultiValuedMap.this.map.put(key, coll);  
423 }  
424 return coll.add(value);  
425 }  
426  
427 @Override  
428 public boolean addAll(final Collection<? extends V> other) {  
429 Collection<V> coll = getMapping();  
430 if (coll == null) {  
431 coll = createCollection();  
432 AbstractMultiValuedMap.this.map.put(key, coll);  
433 }  
434 return coll.addAll(other);  
435 }  
436  
437 @Override  
438 public void clear() {  
439 final Collection<V> coll = getMapping();  
440 if (coll != null) {  
441 coll.clear();  
442 AbstractMultiValuedMap.this.remove(key);  
443 }  
444 }  
445  
446 @Override  
447 public Iterator<V> iterator() {  
448 final Collection<V> coll = getMapping();  
449 if (coll == null) {  
450 return IteratorUtils.EMPTY\_ITERATOR;  
451 }  
452 return new ValuesIterator(key);  
453 }  
454  
455 @Override  
456 public int size() {  
457 final Collection<V> coll = getMapping();  
458 return coll == null ? 0 : coll.size();  
459 }  
460  
461 @Override  
462 public boolean contains(final Object obj) {  
463 final Collection<V> coll = getMapping();  
464 return coll != null && coll.contains(obj);  
465 }  
466  
467 @Override  
468 public boolean containsAll(final Collection<?> other) {  
469 final Collection<V> coll = getMapping();  
470 return coll != null && coll.containsAll(other);  
471 }  
472  
473 @Override  
474 public boolean isEmpty() {  
475 final Collection<V> coll = getMapping();  
476 return coll == null || coll.isEmpty();  
477 }  
478  
479 @Override  
480 public boolean remove(final Object item) {  
481 final Collection<V> coll = getMapping();  
482 if (coll == null) {  
483 return false;  
484 }  
485  
486 final boolean result = coll.remove(item);  
487 if (coll.isEmpty()) {  
488 AbstractMultiValuedMap.this.remove(key);  
489 }  
490 return result;  
491 }  
492  
493 @Override  
494 public boolean removeAll(final Collection<?> c) {  
495 final Collection<V> coll = getMapping();  
496 if (coll == null) {  
497 return false;  
498 }  
499  
500 final boolean result = coll.removeAll(c);  
501 if (coll.isEmpty()) {  
502 AbstractMultiValuedMap.this.remove(key);  
503 }  
504 return result;  
505 }  
506  
507 @Override  
508 public boolean retainAll(final Collection<?> c) {  
509 final Collection<V> coll = getMapping();  
510 if (coll == null) {  
511 return false;  
512 }  
513  
514 final boolean result = coll.retainAll(c);  
515 if (coll.isEmpty()) {  
516 AbstractMultiValuedMap.this.remove(key);  
517 }  
518 return result;  
519 }  
520  
521 @Override  
522 public Object[] toArray() {  
523 final Collection<V> coll = getMapping();  
524 if (coll == null) {  
525 return CollectionUtils.EMPTY\_COLLECTION.toArray();  
526 }  
527 return coll.toArray();  
528 }  
529  
530 @Override  
531 @SuppressWarnings("unchecked")  
532 public <T> T[] toArray(final T[] a) {  
533 final Collection<V> coll = getMapping();  
534 if (coll == null) {  
535 return (T[]) CollectionUtils.EMPTY\_COLLECTION.toArray(a);  
536 }  
537 return coll.toArray(a);  
538 }  
539  
540 @Override  
541 public String toString() {  
542 final Collection<V> coll = getMapping();  
543 if (coll == null) {  
544 return CollectionUtils.EMPTY\_COLLECTION.toString();  
545 }  
546 return coll.toString();  
547 }  
548  
549 }  
550  
551 /\*\*  
552 \* Inner class that provides a MultiSet<K> keys view.  
553 \*/  
554 private class KeysMultiSet extends AbstractMultiSet<K> {  
555  
556 @Override  
557 public boolean contains(final Object o) {  
558 return getMap().containsKey(o);  
559 }  
560  
561 @Override  
562 public boolean isEmpty() {  
563 return getMap().isEmpty();  
564 }  
565  
566 @Override  
567 public int size() {  
568 return AbstractMultiValuedMap.this.size();  
569 }  
570  
571 @Override  
572 protected int uniqueElements() {  
573 return getMap().size();  
574 }  
575  
576 @Override  
577 public int getCount(final Object object) {  
578 int count = 0;  
579 final Collection<V> col = AbstractMultiValuedMap.this.getMap().get(object);  
580 if (col != null) {  
581 count = col.size();  
582 }  
583 return count;  
584 }  
585  
586 @Override  
587 protected Iterator<MultiSet.Entry<K>> createEntrySetIterator() {  
588 final MapEntryTransformer transformer = new MapEntryTransformer();  
589 return IteratorUtils.transformedIterator(map.entrySet().iterator(), transformer);  
590 }  
591  
592 private final class MapEntryTransformer  
593 implements Transformer<Map.Entry<K, Collection<V>>, MultiSet.Entry<K>> {  
594 @Override  
595 public MultiSet.Entry<K> transform(final Map.Entry<K, Collection<V>> mapEntry) {  
596 return new AbstractMultiSet.AbstractEntry<K>() {  
597 @Override  
598 public K getElement() {  
599 return mapEntry.getKey();  
600 }  
601  
602 @Override  
603 public int getCount() {  
604 return mapEntry.getValue().size();  
605 }  
606 };  
607 }  
608 }  
609 }  
610  
611 /\*\*  
612 \* Inner class that provides the Entry<K, V> view  
613 \*/  
614 private class EntryValues extends AbstractCollection<Entry<K, V>> {  
615  
616 @Override  
617 public Iterator<Entry<K, V>> iterator() {  
618 return new LazyIteratorChain<Entry<K, V>>() {  
619  
620 final Collection<K> keysCol = new ArrayList<>(getMap().keySet());  
621 final Iterator<K> keyIterator = keysCol.iterator();  
622  
623 @Override  
624 protected Iterator<? extends Entry<K, V>> nextIterator(final int count) {  
625 if (!keyIterator.hasNext()) {  
626 return null;  
627 }  
628 final K key = keyIterator.next();  
629 final Transformer<V, Entry<K, V>> entryTransformer = new Transformer<V, Entry<K, V>>() {  
630  
631 @Override  
632 public Entry<K, V> transform(final V input) {  
633 return new MultiValuedMapEntry(key, input);  
634 }  
635  
636 };  
637 return new TransformIterator<>(new ValuesIterator(key), entryTransformer);  
638 }  
639 };  
640 }  
641  
642 @Override  
643 public int size() {  
644 return AbstractMultiValuedMap.this.size();  
645 }  
646  
647 }  
648  
649 /\*\*  
650 \* Inner class for MultiValuedMap Entries.  
651 \*/  
652 private class MultiValuedMapEntry extends AbstractMapEntry<K, V> {  
653  
654 public MultiValuedMapEntry(final K key, final V value) {  
655 super(key, value);  
656 }  
657  
658 @Override  
659 public V setValue(final V value) {  
660 throw new UnsupportedOperationException();  
661 }  
662  
663 }  
664  
665 /\*\*  
666 \* Inner class for MapIterator.  
667 \*/  
668 private class MultiValuedMapIterator implements MapIterator<K, V> {  
669  
670 private final Iterator<Entry<K, V>> it;  
671  
672 private Entry<K, V> current = null;  
673  
674 public MultiValuedMapIterator() {  
675 this.it = AbstractMultiValuedMap.this.entries().iterator();  
676 }  
677  
678 @Override  
679 public boolean hasNext() {  
680 return it.hasNext();  
681 }  
682  
683 @Override  
684 public K next() {  
685 current = it.next();  
686 return current.getKey();  
687 }  
688  
689 @Override  
690 public K getKey() {  
691 if (current == null) {  
692 throw new IllegalStateException();  
693 }  
694 return current.getKey();  
695 }  
696  
697 @Override  
698 public V getValue() {  
699 if (current == null) {  
700 throw new IllegalStateException();  
701 }  
702 return current.getValue();  
703 }  
704  
705 @Override  
706 public void remove() {  
707 it.remove();  
708 }  
709  
710 @Override  
711 public V setValue(final V value) {  
712 if (current == null) {  
713 throw new IllegalStateException();  
714 }  
715 return current.setValue(value);  
716 }  
717  
718 }  
719  
720 /\*\*  
721 \* Inner class that provides the values view.  
722 \*/  
723 private class Values extends AbstractCollection<V> {  
724 @Override  
725 public Iterator<V> iterator() {  
726 final IteratorChain<V> chain = new IteratorChain<>();  
727 for (final K k : keySet()) {  
728 chain.addIterator(new ValuesIterator(k));  
729 }  
730 return chain;  
731 }  
732  
733 @Override  
734 public int size() {  
735 return AbstractMultiValuedMap.this.size();  
736 }  
737  
738 @Override  
739 public void clear() {  
740 AbstractMultiValuedMap.this.clear();  
741 }  
742 }  
743  
744 /\*\*  
745 \* Inner class that provides the values iterator.  
746 \*/  
747 private class ValuesIterator implements Iterator<V> {  
748 private final Object key;  
749 private final Collection<V> values;  
750 private final Iterator<V> iterator;  
751  
752 public ValuesIterator(final Object key) {  
753 this.key = key;  
754 this.values = getMap().get(key);  
755 this.iterator = values.iterator();  
756 }  
757  
758 @Override  
759 public void remove() {  
760 iterator.remove();  
761 if (values.isEmpty()) {  
762 AbstractMultiValuedMap.this.remove(key);  
763 }  
764 }  
765  
766 @Override  
767 public boolean hasNext() {  
768 return iterator.hasNext();  
769 }  
770  
771 @Override  
772 public V next() {  
773 return iterator.next();  
774 }  
775 }  
776  
777 /\*\*  
778 \* Inner class that provides the AsMap view.  
779 \*/  
780 private class AsMap extends AbstractMap<K, Collection<V>> {  
781 final transient Map<K, Collection<V>> decoratedMap;  
782  
783 AsMap(final Map<K, Collection<V>> map) {  
784 this.decoratedMap = map;  
785 }  
786  
787 @Override  
788 public Set<Map.Entry<K, Collection<V>>> entrySet() {  
789 return new AsMapEntrySet();  
790 }  
791  
792 @Override  
793 public boolean containsKey(final Object key) {  
794 return decoratedMap.containsKey(key);  
795 }  
796  
797 @Override  
798 public Collection<V> get(final Object key) {  
799 final Collection<V> collection = decoratedMap.get(key);  
800 if (collection == null) {  
801 return null;  
802 }  
803 @SuppressWarnings("unchecked")  
804 final  
805 K k = (K) key;  
806 return wrappedCollection(k);  
807 }  
808  
809 @Override  
810 public Set<K> keySet() {  
811 return AbstractMultiValuedMap.this.keySet();  
812 }  
813  
814 @Override  
815 public int size() {  
816 return decoratedMap.size();  
817 }  
818  
819 @Override  
820 public Collection<V> remove(final Object key) {  
821 final Collection<V> collection = decoratedMap.remove(key);  
822 if (collection == null) {  
823 return null;  
824 }  
825  
826 final Collection<V> output = createCollection();  
827 output.addAll(collection);  
828 collection.clear();  
829 return output;  
830 }  
831  
832 @Override  
833 public boolean equals(final Object object) {  
834 return this == object || decoratedMap.equals(object);  
835 }  
836  
837 @Override  
838 public int hashCode() {  
839 return decoratedMap.hashCode();  
840 }  
841  
842 @Override  
843 public String toString() {  
844 return decoratedMap.toString();  
845 }  
846  
847 @Override  
848 public void clear() {  
849 AbstractMultiValuedMap.this.clear();  
850 }  
851  
852 class AsMapEntrySet extends AbstractSet<Map.Entry<K, Collection<V>>> {  
853  
854 @Override  
855 public Iterator<Map.Entry<K, Collection<V>>> iterator() {  
856 return new AsMapEntrySetIterator(decoratedMap.entrySet().iterator());  
857 }  
858  
859 @Override  
860 public int size() {  
861 return AsMap.this.size();  
862 }  
863  
864 @Override  
865 public void clear() {  
866 AsMap.this.clear();  
867 }  
868  
869 @Override  
870 public boolean contains(final Object o) {  
871 return decoratedMap.entrySet().contains(o);  
872 }  
873  
874 @Override  
875 public boolean remove(final Object o) {  
876 if (!contains(o)) {  
877 return false;  
878 }  
879 final Map.Entry<?, ?> entry = (Map.Entry<?, ?>) o;  
880 AbstractMultiValuedMap.this.remove(entry.getKey());  
881 return true;  
882 }  
883 }  
884  
885 /\*\*  
886 \* EntrySet iterator for the asMap view.  
887 \*/  
888 class AsMapEntrySetIterator extends AbstractIteratorDecorator<Map.Entry<K, Collection<V>>> {  
889  
890 AsMapEntrySetIterator(final Iterator<Map.Entry<K, Collection<V>>> iterator) {  
891 super(iterator);  
892 }  
893  
894 @Override  
895 public Map.Entry<K, Collection<V>> next() {  
896 final Map.Entry<K, Collection<V>> entry = super.next();  
897 final K key = entry.getKey();  
898 return new UnmodifiableMapEntry<>(key, wrappedCollection(key));  
899 }  
900 }  
901 }  
902  
903 //-----------------------------------------------------------------------  
904 /\*\*  
905 \* Write the map out using a custom routine.  
906 \* @param out the output stream  
907 \* @throws IOException any of the usual I/O related exceptions  
908 \*/  
909 protected void doWriteObject(final ObjectOutputStream out) throws IOException {  
910 out.writeInt(map.size());  
911 for (final Map.Entry<K, Collection<V>> entry : map.entrySet()) {  
912 out.writeObject(entry.getKey());  
913 out.writeInt(entry.getValue().size());  
914 for (final V value : entry.getValue()) {  
915 out.writeObject(value);  
916 }  
917 }  
918 }  
919  
920 /\*\*  
921 \* Read the map in using a custom routine.  
922 \* @param in the input stream  
923 \* @throws IOException any of the usual I/O related exceptions  
924 \* @throws ClassNotFoundException if the stream contains an object which class can not be loaded  
925 \* @throws ClassCastException if the stream does not contain the correct objects  
926 \*/  
927 protected void doReadObject(final ObjectInputStream in)  
928 throws IOException, ClassNotFoundException {  
929 final int entrySize = in.readInt();  
930 for (int i = 0; i < entrySize; i++) {  
931 @SuppressWarnings("unchecked") // This will fail at runtime if the stream is incorrect  
932 final K key = (K) in.readObject();  
933 final Collection<V> values = get(key);  
934 final int valueSize = in.readInt();  
935 for (int j = 0; j < valueSize; j++) {  
936 @SuppressWarnings("unchecked") // see above  
937 final  
938 V value = (V) in.readObject();  
939 values.add(value);  
940 }  
941 }  
942 }  
943  
944}