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.AbstractCollection;  
024import java.util.ArrayList;  
025import java.util.Collection;  
026import java.util.HashMap;  
027import java.util.Iterator;  
028import java.util.Map;  
029import java.util.Set;  
030  
031import org.apache.commons.collections4.CollectionUtils;  
032import org.apache.commons.collections4.Factory;  
033import org.apache.commons.collections4.FunctorException;  
034import org.apache.commons.collections4.MultiMap;  
035import org.apache.commons.collections4.Transformer;  
036import org.apache.commons.collections4.iterators.EmptyIterator;  
037import org.apache.commons.collections4.iterators.IteratorChain;  
038import org.apache.commons.collections4.iterators.LazyIteratorChain;  
039import org.apache.commons.collections4.iterators.TransformIterator;  
040  
041/\*\*  
042 \* A MultiValueMap decorates another map, allowing it to have  
043 \* more than one value for a key.  
044 \* <p>  
045 \* A <code>MultiMap</code> is a Map with slightly different semantics.  
046 \* Putting a value into the map will add the value to a Collection at that key.  
047 \* Getting a value will return a Collection, holding all the values put to that key.  
048 \* </p>  
049 \* <p>  
050 \* This implementation is a decorator, allowing any Map implementation  
051 \* to be used as the base.  
052 \* </p>  
053 \* <p>  
054 \* In addition, this implementation allows the type of collection used  
055 \* for the values to be controlled. By default, an <code>ArrayList</code>  
056 \* is used, however a <code>Class</code> to instantiate may be specified,  
057 \* or a factory that returns a <code>Collection</code> instance.  
058 \* </p>  
059 \* <p>  
060 \* <strong>Note that MultiValueMap is not synchronized and is not thread-safe.</strong>  
061 \* If you wish to use this map from multiple threads concurrently, you must use  
062 \* appropriate synchronization. This class may throw exceptions when accessed  
063 \* by concurrent threads without synchronization.  
064 \* </p>  
065 \*  
066 \* @param <K> the type of the keys in this map  
067 \* @param <V> the type of the values in this map  
068 \* @since 3.2  
069 \* @deprecated since 4.1, use {@link org.apache.commons.collections4.MultiValuedMap MultiValuedMap} instead  
070 \*/  
071@Deprecated  
072public class MultiValueMap<K, V> extends AbstractMapDecorator<K, Object> implements MultiMap<K, V>, Serializable {  
073  
074 /\*\* Serialization version \*/  
075 private static final long serialVersionUID = -2214159910087182007L;  
076  
077 /\*\* The factory for creating value collections. \*/  
078 private final Factory<? extends Collection<V>> collectionFactory;  
079 /\*\* The cached values. \*/  
080 private transient Collection<V> valuesView;  
081  
082 /\*\*  
083 \* Creates a map which wraps the given map and  
084 \* maps keys to ArrayLists.  
085 \*  
086 \* @param <K> the key type  
087 \* @param <V> the value type  
088 \* @param map the map to wrap  
089 \* @return a new multi-value map  
090 \* @since 4.0  
091 \*/  
092 @SuppressWarnings({ "unchecked", "rawtypes" })  
093 public static <K, V> MultiValueMap<K, V> multiValueMap(final Map<K, ? super Collection<V>> map) {  
094 return MultiValueMap.<K, V, ArrayList> multiValueMap((Map<K, ? super Collection>) map, ArrayList.class);  
095 }  
096  
097 /\*\*  
098 \* Creates a map which decorates the given <code>map</code> and  
099 \* maps keys to collections of type <code>collectionClass</code>.  
100 \*  
101 \* @param <K> the key type  
102 \* @param <V> the value type  
103 \* @param <C> the collection class type  
104 \* @param map the map to wrap  
105 \* @param collectionClass the type of the collection class  
106 \* @return a new multi-value map  
107 \* @since 4.0  
108 \*/  
109 public static <K, V, C extends Collection<V>> MultiValueMap<K, V> multiValueMap(final Map<K, ? super C> map,  
110 final Class<C> collectionClass) {  
111 return new MultiValueMap<>(map, new ReflectionFactory<>(collectionClass));  
112 }  
113  
114 /\*\*  
115 \* Creates a map which decorates the given <code>map</code> and  
116 \* creates the value collections using the supplied <code>collectionFactory</code>.  
117 \*  
118 \* @param <K> the key type  
119 \* @param <V> the value type  
120 \* @param <C> the collection class type  
121 \* @param map the map to decorate  
122 \* @param collectionFactory the collection factory (must return a Collection object).  
123 \* @return a new multi-value map  
124 \* @since 4.0  
125 \*/  
126 public static <K, V, C extends Collection<V>> MultiValueMap<K, V> multiValueMap(final Map<K, ? super C> map,  
127 final Factory<C> collectionFactory) {  
128 return new MultiValueMap<>(map, collectionFactory);  
129 }  
130  
131 //-----------------------------------------------------------------------  
132 /\*\*  
133 \* Creates a MultiValueMap based on a <code>HashMap</code> and  
134 \* storing the multiple values in an <code>ArrayList</code>.  
135 \*/  
136 @SuppressWarnings({ "unchecked", "rawtypes" })  
137 public MultiValueMap() {  
138 this(new HashMap<K, V>(), new ReflectionFactory(ArrayList.class));  
139 }  
140  
141 /\*\*  
142 \* Creates a MultiValueMap which decorates the given <code>map</code> and  
143 \* creates the value collections using the supplied <code>collectionFactory</code>.  
144 \*  
145 \* @param <C> the collection class type  
146 \* @param map the map to decorate  
147 \* @param collectionFactory the collection factory which must return a Collection instance  
148 \*/  
149 @SuppressWarnings("unchecked")  
150 protected <C extends Collection<V>> MultiValueMap(final Map<K, ? super C> map,  
151 final Factory<C> collectionFactory) {  
152 super((Map<K, Object>) map);  
153 if (collectionFactory == null) {  
154 throw new IllegalArgumentException("The factory must not be null");  
155 }  
156 this.collectionFactory = collectionFactory;  
157 }  
158  
159 //-----------------------------------------------------------------------  
160 /\*\*  
161 \* Write the map out using a custom routine.  
162 \*  
163 \* @param out the output stream  
164 \* @throws IOException if an error occurs while writing to the stream  
165 \* @since 4.0  
166 \*/  
167 private void writeObject(final ObjectOutputStream out) throws IOException {  
168 out.defaultWriteObject();  
169 out.writeObject(map);  
170 }  
171  
172 /\*\*  
173 \* Read the map in using a custom routine.  
174 \*  
175 \* @param in the input stream  
176 \* @throws IOException if an error occurs while reading from the stream  
177 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
178 \* @since 4.0  
179 \*/  
180 @SuppressWarnings("unchecked") // (1) should only fail if input stream is incorrect  
181 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
182 in.defaultReadObject();  
183 map = (Map<K, Object>) in.readObject(); // (1)  
184 }  
185  
186 //-----------------------------------------------------------------------  
187 /\*\*  
188 \* Clear the map.  
189 \*/  
190 @Override  
191 public void clear() {  
192 // If you believe that you have GC issues here, try uncommenting this code  
193// Set pairs = getMap().entrySet();  
194// Iterator pairsIterator = pairs.iterator();  
195// while (pairsIterator.hasNext()) {  
196// Map.Entry keyValuePair = (Map.Entry) pairsIterator.next();  
197// Collection coll = (Collection) keyValuePair.getValue();  
198// coll.clear();  
199// }  
200 decorated().clear();  
201 }  
202  
203 /\*\*  
204 \* Removes a specific value from map.  
205 \* <p>  
206 \* The item is removed from the collection mapped to the specified key.  
207 \* Other values attached to that key are unaffected.  
208 \* <p>  
209 \* If the last value for a key is removed, <code>null</code> will be returned  
210 \* from a subsequent <code>get(key)</code>.  
211 \*  
212 \* @param key the key to remove from  
213 \* @param value the value to remove  
214 \* @return {@code true} if the mapping was removed, {@code false} otherwise  
215 \*/  
216 @Override  
217 public boolean removeMapping(final Object key, final Object value) {  
218 final Collection<V> valuesForKey = getCollection(key);  
219 if (valuesForKey == null) {  
220 return false;  
221 }  
222 final boolean removed = valuesForKey.remove(value);  
223 if (removed == false) {  
224 return false;  
225 }  
226 if (valuesForKey.isEmpty()) {  
227 remove(key);  
228 }  
229 return true;  
230 }  
231  
232 /\*\*  
233 \* Checks whether the map contains the value specified.  
234 \* <p>  
235 \* This checks all collections against all keys for the value, and thus could be slow.  
236 \*  
237 \* @param value the value to search for  
238 \* @return true if the map contains the value  
239 \*/  
240 @Override  
241 @SuppressWarnings("unchecked")  
242 public boolean containsValue(final Object value) {  
243 final Set<Map.Entry<K, Object>> pairs = decorated().entrySet();  
244 if (pairs != null) {  
245 for (final Map.Entry<K, Object> entry : pairs) {  
246 if (((Collection<V>) entry.getValue()).contains(value)) {  
247 return true;  
248 }  
249 }  
250 }  
251 return false;  
252 }  
253  
254 /\*\*  
255 \* Adds the value to the collection associated with the specified key.  
256 \* <p>  
257 \* Unlike a normal <code>Map</code> the previous value is not replaced.  
258 \* Instead the new value is added to the collection stored against the key.  
259 \*  
260 \* @param key the key to store against  
261 \* @param value the value to add to the collection at the key  
262 \* @return the value added if the map changed and null if the map did not change  
263 \*/  
264 @Override  
265 @SuppressWarnings("unchecked")  
266 public Object put(final K key, final Object value) {  
267 boolean result = false;  
268 Collection<V> coll = getCollection(key);  
269 if (coll == null) {  
270 coll = createCollection(1); // might produce a non-empty collection  
271 coll.add((V) value);  
272 if (coll.size() > 0) {  
273 // only add if non-zero size to maintain class state  
274 decorated().put(key, coll);  
275 result = true; // map definitely changed  
276 }  
277 } else {  
278 result = coll.add((V) value);  
279 }  
280 return result ? value : null;  
281 }  
282  
283 /\*\*  
284 \* Override superclass to ensure that MultiMap instances are  
285 \* correctly handled.  
286 \* <p>  
287 \* If you call this method with a normal map, each entry is  
288 \* added using <code>put(Object,Object)</code>.  
289 \* If you call this method with a multi map, each entry is  
290 \* added using <code>putAll(Object,Collection)</code>.  
291 \*  
292 \* @param map the map to copy (either a normal or multi map)  
293 \*/  
294 @Override  
295 @SuppressWarnings("unchecked")  
296 public void putAll(final Map<? extends K, ?> map) {  
297 if (map instanceof MultiMap) {  
298 for (final Map.Entry<? extends K, Object> entry : ((MultiMap<? extends K, V>) map).entrySet()) {  
299 putAll(entry.getKey(), (Collection<V>) entry.getValue());  
300 }  
301 } else {  
302 for (final Map.Entry<? extends K, ?> entry : map.entrySet()) {  
303 put(entry.getKey(), entry.getValue());  
304 }  
305 }  
306 }  
307  
308 /\*\*  
309 \* {@inheritDoc}  
310 \* <p>  
311 \* NOTE: the returned Entry objects will contain as value a {@link Collection}  
312 \* of all values that are mapped to the given key. To get a "flattened" version  
313 \* of all mappings contained in this map, use {@link #iterator()}.  
314 \*  
315 \* @see #iterator()  
316 \*/  
317 @Override  
318 public Set<Entry<K, Object>> entrySet() { // NOPMD  
319 return super.entrySet();  
320 }  
321  
322 /\*\*  
323 \* Gets a collection containing all the values in the map.  
324 \* <p>  
325 \* This returns a collection containing the combination of values from all keys.  
326 \*  
327 \* @return a collection view of the values contained in this map  
328 \*/  
329 @Override  
330 @SuppressWarnings("unchecked")  
331 public Collection<Object> values() {  
332 final Collection<V> vs = valuesView;  
333 return (Collection<Object>) (vs != null ? vs : (valuesView = new Values()));  
334 }  
335  
336 /\*\*  
337 \* Checks whether the collection at the specified key contains the value.  
338 \*  
339 \* @param key the key to search for  
340 \* @param value the value to search for  
341 \* @return true if the map contains the value  
342 \*/  
343 public boolean containsValue(final Object key, final Object value) {  
344 final Collection<V> coll = getCollection(key);  
345 if (coll == null) {  
346 return false;  
347 }  
348 return coll.contains(value);  
349 }  
350  
351 /\*\*  
352 \* Gets the collection mapped to the specified key.  
353 \* This method is a convenience method to typecast the result of <code>get(key)</code>.  
354 \*  
355 \* @param key the key to retrieve  
356 \* @return the collection mapped to the key, null if no mapping  
357 \*/  
358 @SuppressWarnings("unchecked")  
359 public Collection<V> getCollection(final Object key) {  
360 return (Collection<V>) decorated().get(key);  
361 }  
362  
363 /\*\*  
364 \* Gets the size of the collection mapped to the specified key.  
365 \*  
366 \* @param key the key to get size for  
367 \* @return the size of the collection at the key, zero if key not in map  
368 \*/  
369 public int size(final Object key) {  
370 final Collection<V> coll = getCollection(key);  
371 if (coll == null) {  
372 return 0;  
373 }  
374 return coll.size();  
375 }  
376  
377 /\*\*  
378 \* Adds a collection of values to the collection associated with  
379 \* the specified key.  
380 \*  
381 \* @param key the key to store against  
382 \* @param values the values to add to the collection at the key, null ignored  
383 \* @return true if this map changed  
384 \*/  
385 public boolean putAll(final K key, final Collection<V> values) {  
386 if (values == null || values.size() == 0) {  
387 return false;  
388 }  
389 boolean result = false;  
390 Collection<V> coll = getCollection(key);  
391 if (coll == null) {  
392 coll = createCollection(values.size()); // might produce a non-empty collection  
393 coll.addAll(values);  
394 if (coll.size() > 0) {  
395 // only add if non-zero size to maintain class state  
396 decorated().put(key, coll);  
397 result = true; // map definitely changed  
398 }  
399 } else {  
400 result = coll.addAll(values);  
401 }  
402 return result;  
403 }  
404  
405 /\*\*  
406 \* Gets an iterator for the collection mapped to the specified key.  
407 \*  
408 \* @param key the key to get an iterator for  
409 \* @return the iterator of the collection at the key, empty iterator if key not in map  
410 \*/  
411 public Iterator<V> iterator(final Object key) {  
412 if (!containsKey(key)) {  
413 return EmptyIterator.<V>emptyIterator();  
414 }  
415 return new ValuesIterator(key);  
416 }  
417  
418 /\*\*  
419 \* Gets an iterator for all mappings stored in this {@link MultiValueMap}.  
420 \* <p>  
421 \* The iterator will return multiple Entry objects with the same key  
422 \* if there are multiple values mapped to this key.  
423 \* <p>  
424 \* NOTE: calling {@link java.util.Map.Entry#setValue(Object)} on any of the returned  
425 \* elements will result in a {@link UnsupportedOperationException}.  
426 \*  
427 \* @return the iterator of all mappings in this map  
428 \* @since 4.0  
429 \*/  
430 public Iterator<Entry<K, V>> iterator() {  
431 final Collection<K> allKeys = new ArrayList<>(keySet());  
432 final Iterator<K> keyIterator = allKeys.iterator();  
433  
434 return new LazyIteratorChain<Entry<K, V>>() {  
435 @Override  
436 protected Iterator<? extends Entry<K, V>> nextIterator(final int count) {  
437 if ( ! keyIterator.hasNext() ) {  
438 return null;  
439 }  
440 final K key = keyIterator.next();  
441 final Transformer<V, Entry<K, V>> transformer = new Transformer<V, Entry<K, V>>() {  
442 @Override  
443 public Entry<K, V> transform(final V input) {  
444 return new Entry<K, V>() {  
445 @Override  
446 public K getKey() {  
447 return key;  
448 }  
449 @Override  
450 public V getValue() {  
451 return input;  
452 }  
453 @Override  
454 public V setValue(final V value) {  
455 throw new UnsupportedOperationException();  
456 }  
457 };  
458 }  
459 };  
460 return new TransformIterator<>(new ValuesIterator(key), transformer);  
461 }  
462 };  
463 }  
464  
465 /\*\*  
466 \* Gets the total size of the map by counting all the values.  
467 \*  
468 \* @return the total size of the map counting all values  
469 \*/  
470 public int totalSize() {  
471 int total = 0;  
472 for (final Object v : decorated().values()) {  
473 total += CollectionUtils.size(v);  
474 }  
475 return total;  
476 }  
477  
478 /\*\*  
479 \* Creates a new instance of the map value Collection container  
480 \* using the factory.  
481 \* <p>  
482 \* This method can be overridden to perform your own processing  
483 \* instead of using the factory.  
484 \*  
485 \* @param size the collection size that is about to be added  
486 \* @return the new collection  
487 \*/  
488 protected Collection<V> createCollection(final int size) {  
489 return collectionFactory.create();  
490 }  
491  
492 //-----------------------------------------------------------------------  
493 /\*\*  
494 \* Inner class that provides the values view.  
495 \*/  
496 private class Values extends AbstractCollection<V> {  
497 @Override  
498 public Iterator<V> iterator() {  
499 final IteratorChain<V> chain = new IteratorChain<>();  
500 for (final K k : keySet()) {  
501 chain.addIterator(new ValuesIterator(k));  
502 }  
503 return chain;  
504 }  
505  
506 @Override  
507 public int size() {  
508 return totalSize();  
509 }  
510  
511 @Override  
512 public void clear() {  
513 MultiValueMap.this.clear();  
514 }  
515 }  
516  
517 /\*\*  
518 \* Inner class that provides the values iterator.  
519 \*/  
520 private class ValuesIterator implements Iterator<V> {  
521 private final Object key;  
522 private final Collection<V> values;  
523 private final Iterator<V> iterator;  
524  
525 public ValuesIterator(final Object key) {  
526 this.key = key;  
527 this.values = getCollection(key);  
528 this.iterator = values.iterator();  
529 }  
530  
531 @Override  
532 public void remove() {  
533 iterator.remove();  
534 if (values.isEmpty()) {  
535 MultiValueMap.this.remove(key);  
536 }  
537 }  
538  
539 @Override  
540 public boolean hasNext() {  
541 return iterator.hasNext();  
542 }  
543  
544 @Override  
545 public V next() {  
546 return iterator.next();  
547 }  
548 }  
549  
550 /\*\*  
551 \* Inner class that provides a simple reflection factory.  
552 \*/  
553 private static class ReflectionFactory<T extends Collection<?>> implements Factory<T>, Serializable {  
554  
555 /\*\* Serialization version \*/  
556 private static final long serialVersionUID = 2986114157496788874L;  
557  
558 private final Class<T> clazz;  
559  
560 public ReflectionFactory(final Class<T> clazz) {  
561 this.clazz = clazz;  
562 }  
563  
564 @Override  
565 public T create() {  
566 try {  
567 return clazz.getDeclaredConstructor().newInstance();  
568 } catch (final Exception ex) {  
569 throw new FunctorException("Cannot instantiate class: " + clazz, ex);  
570 }  
571 }  
572  
573 private void readObject(final ObjectInputStream is) throws IOException, ClassNotFoundException {  
574 is.defaultReadObject();  
575 // ensure that the de-serialized class is a Collection, COLLECTIONS-580  
576 if (clazz != null && !Collection.class.isAssignableFrom(clazz)) {  
577 throw new UnsupportedOperationException();  
578 }  
579 }  
580 }  
581  
582}