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.multiset;  
018  
019import java.io.IOException;  
020import java.io.ObjectInputStream;  
021import java.io.ObjectOutputStream;  
022import java.lang.reflect.Array;  
023import java.util.ConcurrentModificationException;  
024import java.util.Iterator;  
025import java.util.Map;  
026  
027import org.apache.commons.collections4.MultiSet;  
028import org.apache.commons.collections4.iterators.AbstractIteratorDecorator;  
029  
030/\*\*  
031 \* Abstract implementation of the {@link MultiSet} interface to simplify the  
032 \* creation of subclass implementations.  
033 \* <p>  
034 \* Subclasses specify a Map implementation to use as the internal storage. The  
035 \* map will be used to map multiset elements to a number; the number represents the  
036 \* number of occurrences of that element in the multiset.  
037 \* </p>  
038 \*  
039 \* @param <E> the type held in the multiset  
040 \* @since 4.1  
041 \*/  
042public abstract class AbstractMapMultiSet<E> extends AbstractMultiSet<E> {  
043  
044 /\*\* The map to use to store the data \*/  
045 private transient Map<E, MutableInteger> map;  
046 /\*\* The current total size of the multiset \*/  
047 private transient int size;  
048 /\*\* The modification count for fail fast iterators \*/  
049 private transient int modCount;  
050  
051 /\*\*  
052 \* Constructor needed for subclass serialisation.  
053 \*/  
054 protected AbstractMapMultiSet() {  
055 super();  
056 }  
057  
058 /\*\*  
059 \* Constructor that assigns the specified Map as the backing store. The map  
060 \* must be empty and non-null.  
061 \*  
062 \* @param map the map to assign  
063 \*/  
064 protected AbstractMapMultiSet(final Map<E, MutableInteger> map) {  
065 super();  
066 this.map = map;  
067 }  
068  
069 /\*\*  
070 \* Utility method for implementations to access the map that backs this multiset.  
071 \* Not intended for interactive use outside of subclasses.  
072 \*  
073 \* @return the map being used by the MultiSet  
074 \*/  
075 protected Map<E, MutableInteger> getMap() {  
076 return map;  
077 }  
078  
079 /\*\*  
080 \* Sets the map being wrapped.  
081 \* <p>  
082 \* <b>NOTE:</b> this method should only be used during deserialization  
083 \*  
084 \* @param map the map to wrap  
085 \*/  
086 protected void setMap(final Map<E, MutableInteger> map) {  
087 this.map = map;  
088 }  
089  
090 //-----------------------------------------------------------------------  
091 /\*\*  
092 \* Returns the number of elements in this multiset.  
093 \*  
094 \* @return current size of the multiset  
095 \*/  
096 @Override  
097 public int size() {  
098 return size;  
099 }  
100  
101 /\*\*  
102 \* Returns true if the underlying map is empty.  
103 \*  
104 \* @return true if multiset is empty  
105 \*/  
106 @Override  
107 public boolean isEmpty() {  
108 return map.isEmpty();  
109 }  
110  
111 /\*\*  
112 \* Returns the number of occurrence of the given element in this multiset by  
113 \* looking up its count in the underlying map.  
114 \*  
115 \* @param object the object to search for  
116 \* @return the number of occurrences of the object, zero if not found  
117 \*/  
118 @Override  
119 public int getCount(final Object object) {  
120 final MutableInteger count = map.get(object);  
121 if (count != null) {  
122 return count.value;  
123 }  
124 return 0;  
125 }  
126  
127 //-----------------------------------------------------------------------  
128 /\*\*  
129 \* Determines if the multiset contains the given element by checking if the  
130 \* underlying map contains the element as a key.  
131 \*  
132 \* @param object the object to search for  
133 \* @return true if the multiset contains the given element  
134 \*/  
135 @Override  
136 public boolean contains(final Object object) {  
137 return map.containsKey(object);  
138 }  
139  
140 //-----------------------------------------------------------------------  
141 /\*\*  
142 \* Gets an iterator over the multiset elements. Elements present in the  
143 \* MultiSet more than once will be returned repeatedly.  
144 \*  
145 \* @return the iterator  
146 \*/  
147 @Override  
148 public Iterator<E> iterator() {  
149 return new MapBasedMultiSetIterator<>(this);  
150 }  
151  
152 /\*\*  
153 \* Inner class iterator for the MultiSet.  
154 \*/  
155 private static class MapBasedMultiSetIterator<E> implements Iterator<E> {  
156 private final AbstractMapMultiSet<E> parent;  
157 private final Iterator<Map.Entry<E, MutableInteger>> entryIterator;  
158 private Map.Entry<E, MutableInteger> current;  
159 private int itemCount;  
160 private final int mods;  
161 private boolean canRemove;  
162  
163 /\*\*  
164 \* Constructor.  
165 \*  
166 \* @param parent the parent multiset  
167 \*/  
168 public MapBasedMultiSetIterator(final AbstractMapMultiSet<E> parent) {  
169 this.parent = parent;  
170 this.entryIterator = parent.map.entrySet().iterator();  
171 this.current = null;  
172 this.mods = parent.modCount;  
173 this.canRemove = false;  
174 }  
175  
176 /\*\* {@inheritDoc} \*/  
177 @Override  
178 public boolean hasNext() {  
179 return itemCount > 0 || entryIterator.hasNext();  
180 }  
181  
182 /\*\* {@inheritDoc} \*/  
183 @Override  
184 public E next() {  
185 if (parent.modCount != mods) {  
186 throw new ConcurrentModificationException();  
187 }  
188 if (itemCount == 0) {  
189 current = entryIterator.next();  
190 itemCount = current.getValue().value;  
191 }  
192 canRemove = true;  
193 itemCount--;  
194 return current.getKey();  
195 }  
196  
197 /\*\* {@inheritDoc} \*/  
198 @Override  
199 public void remove() {  
200 if (parent.modCount != mods) {  
201 throw new ConcurrentModificationException();  
202 }  
203 if (canRemove == false) {  
204 throw new IllegalStateException();  
205 }  
206 final MutableInteger mut = current.getValue();  
207 if (mut.value > 1) {  
208 mut.value--;  
209 } else {  
210 entryIterator.remove();  
211 }  
212 parent.size--;  
213 canRemove = false;  
214 }  
215 }  
216  
217 //-----------------------------------------------------------------------  
218 @Override  
219 public int add(final E object, final int occurrences) {  
220 if (occurrences < 0) {  
221 throw new IllegalArgumentException("Occurrences must not be negative.");  
222 }  
223  
224 final MutableInteger mut = map.get(object);  
225 final int oldCount = mut != null ? mut.value : 0;  
226  
227 if (occurrences > 0) {  
228 modCount++;  
229 size += occurrences;  
230 if (mut == null) {  
231 map.put(object, new MutableInteger(occurrences));  
232 } else {  
233 mut.value += occurrences;  
234 }  
235 }  
236 return oldCount;  
237 }  
238  
239 //-----------------------------------------------------------------------  
240 /\*\*  
241 \* Clears the multiset by clearing the underlying map.  
242 \*/  
243 @Override  
244 public void clear() {  
245 modCount++;  
246 map.clear();  
247 size = 0;  
248 }  
249  
250 @Override  
251 public int remove(final Object object, final int occurrences) {  
252 if (occurrences < 0) {  
253 throw new IllegalArgumentException("Occurrences must not be negative.");  
254 }  
255  
256 final MutableInteger mut = map.get(object);  
257 if (mut == null) {  
258 return 0;  
259 }  
260 final int oldCount = mut.value;  
261 if (occurrences > 0) {  
262 modCount++;  
263 if (occurrences < mut.value) {  
264 mut.value -= occurrences;  
265 size -= occurrences;  
266 } else {  
267 map.remove(object);  
268 size -= mut.value;  
269 mut.value = 0;  
270 }  
271 }  
272 return oldCount;  
273 }  
274  
275 //-----------------------------------------------------------------------  
276 /\*\*  
277 \* Mutable integer class for storing the data.  
278 \*/  
279 protected static class MutableInteger {  
280 /\*\* The value of this mutable. \*/  
281 protected int value;  
282  
283 /\*\*  
284 \* Constructor.  
285 \* @param value the initial value  
286 \*/  
287 MutableInteger(final int value) {  
288 this.value = value;  
289 }  
290  
291 @Override  
292 public boolean equals(final Object obj) {  
293 if (obj instanceof MutableInteger == false) {  
294 return false;  
295 }  
296 return ((MutableInteger) obj).value == value;  
297 }  
298  
299 @Override  
300 public int hashCode() {  
301 return value;  
302 }  
303 }  
304  
305 //-----------------------------------------------------------------------  
306 @Override  
307 protected Iterator<E> createUniqueSetIterator() {  
308 return new UniqueSetIterator<>(getMap().keySet().iterator(), this);  
309 }  
310  
311 @Override  
312 protected int uniqueElements() {  
313 return map.size();  
314 }  
315  
316 @Override  
317 protected Iterator<Entry<E>> createEntrySetIterator() {  
318 return new EntrySetIterator<>(map.entrySet().iterator(), this);  
319 }  
320  
321 //-----------------------------------------------------------------------  
322 /\*\*  
323 \* Inner class UniqueSetIterator.  
324 \*/  
325 protected static class UniqueSetIterator<E> extends AbstractIteratorDecorator<E> {  
326  
327 /\*\* The parent multiset \*/  
328 protected final AbstractMapMultiSet<E> parent;  
329  
330 /\*\* The last returned element \*/  
331 protected E lastElement = null;  
332  
333 /\*\* Whether remove is allowed at present \*/  
334 protected boolean canRemove = false;  
335  
336 /\*\*  
337 \* Constructor.  
338 \* @param iterator the iterator to decorate  
339 \* @param parent the parent multiset  
340 \*/  
341 protected UniqueSetIterator(final Iterator<E> iterator, final AbstractMapMultiSet<E> parent) {  
342 super(iterator);  
343 this.parent = parent;  
344 }  
345  
346 @Override  
347 public E next() {  
348 lastElement = super.next();  
349 canRemove = true;  
350 return lastElement;  
351 }  
352  
353 @Override  
354 public void remove() {  
355 if (canRemove == false) {  
356 throw new IllegalStateException("Iterator remove() can only be called once after next()");  
357 }  
358 final int count = parent.getCount(lastElement);  
359 super.remove();  
360 parent.remove(lastElement, count);  
361 lastElement = null;  
362 canRemove = false;  
363 }  
364 }  
365  
366 /\*\*  
367 \* Inner class EntrySetIterator.  
368 \*/  
369 protected static class EntrySetIterator<E> implements Iterator<Entry<E>> {  
370  
371 /\*\* The parent map \*/  
372 protected final AbstractMapMultiSet<E> parent;  
373  
374 protected final Iterator<Map.Entry<E, MutableInteger>> decorated;  
375  
376 /\*\* The last returned entry \*/  
377 protected Entry<E> last = null;  
378  
379 /\*\* Whether remove is allowed at present \*/  
380 protected boolean canRemove = false;  
381  
382 /\*\*  
383 \* Constructor.  
384 \* @param iterator the iterator to decorate  
385 \* @param parent the parent multiset  
386 \*/  
387 protected EntrySetIterator(final Iterator<Map.Entry<E, MutableInteger>> iterator,  
388 final AbstractMapMultiSet<E> parent) {  
389 this.decorated = iterator;  
390 this.parent = parent;  
391 }  
392  
393 @Override  
394 public boolean hasNext() {  
395 return decorated.hasNext();  
396 }  
397  
398 @Override  
399 public Entry<E> next() {  
400 last = new MultiSetEntry<>(decorated.next());  
401 canRemove = true;  
402 return last;  
403 }  
404  
405 @Override  
406 public void remove() {  
407 if (canRemove == false) {  
408 throw new IllegalStateException("Iterator remove() can only be called once after next()");  
409 }  
410 decorated.remove();  
411 last = null;  
412 canRemove = false;  
413 }  
414 }  
415  
416 /\*\*  
417 \* Inner class MultiSetEntry.  
418 \*/  
419 protected static class MultiSetEntry<E> extends AbstractEntry<E> {  
420  
421 protected final Map.Entry<E, MutableInteger> parentEntry;  
422  
423 /\*\*  
424 \* Constructor.  
425 \* @param parentEntry the entry to decorate  
426 \*/  
427 protected MultiSetEntry(final Map.Entry<E, MutableInteger> parentEntry) {  
428 this.parentEntry = parentEntry;  
429 }  
430  
431 @Override  
432 public E getElement() {  
433 return parentEntry.getKey();  
434 }  
435  
436 @Override  
437 public int getCount() {  
438 return parentEntry.getValue().value;  
439 }  
440 }  
441  
442 //-----------------------------------------------------------------------  
443 /\*\*  
444 \* Write the multiset out using a custom routine.  
445 \* @param out the output stream  
446 \* @throws IOException any of the usual I/O related exceptions  
447 \*/  
448 @Override  
449 protected void doWriteObject(final ObjectOutputStream out) throws IOException {  
450 out.writeInt(map.size());  
451 for (final Map.Entry<E, MutableInteger> entry : map.entrySet()) {  
452 out.writeObject(entry.getKey());  
453 out.writeInt(entry.getValue().value);  
454 }  
455 }  
456  
457 /\*\*  
458 \* Read the multiset in using a custom routine.  
459 \* @param in the input stream  
460 \* @throws IOException any of the usual I/O related exceptions  
461 \* @throws ClassNotFoundException if the stream contains an object which class can not be loaded  
462 \* @throws ClassCastException if the stream does not contain the correct objects  
463 \*/  
464 @Override  
465 protected void doReadObject(final ObjectInputStream in)  
466 throws IOException, ClassNotFoundException {  
467 final int entrySize = in.readInt();  
468 for (int i = 0; i < entrySize; i++) {  
469 @SuppressWarnings("unchecked") // This will fail at runtime if the stream is incorrect  
470 final E obj = (E) in.readObject();  
471 final int count = in.readInt();  
472 map.put(obj, new MutableInteger(count));  
473 size += count;  
474 }  
475 }  
476  
477 //-----------------------------------------------------------------------  
478 /\*\*  
479 \* Returns an array of all of this multiset's elements.  
480 \*  
481 \* @return an array of all of this multiset's elements  
482 \*/  
483 @Override  
484 public Object[] toArray() {  
485 final Object[] result = new Object[size()];  
486 int i = 0;  
487 for (final Map.Entry<E, MutableInteger> entry : map.entrySet()) {  
488 final E current = entry.getKey();  
489 final MutableInteger count = entry.getValue();  
490 for (int index = count.value; index > 0; index--) {  
491 result[i++] = current;  
492 }  
493 }  
494 return result;  
495 }  
496  
497 /\*\*  
498 \* Returns an array of all of this multiset's elements.  
499 \* If the input array has more elements than are in the multiset,  
500 \* trailing elements will be set to null.  
501 \*  
502 \* @param <T> the type of the array elements  
503 \* @param array the array to populate  
504 \* @return an array of all of this multiset's elements  
505 \* @throws ArrayStoreException if the runtime type of the specified array is not  
506 \* a supertype of the runtime type of the elements in this list  
507 \* @throws NullPointerException if the specified array is null  
508 \*/  
509 @Override  
510 public <T> T[] toArray(T[] array) {  
511 final int size = size();  
512 if (array.length < size) {  
513 @SuppressWarnings("unchecked") // safe as both are of type T  
514 final T[] unchecked = (T[]) Array.newInstance(array.getClass().getComponentType(), size);  
515 array = unchecked;  
516 }  
517  
518 int i = 0;  
519 for (final Map.Entry<E, MutableInteger> entry : map.entrySet()) {  
520 final E current = entry.getKey();  
521 final MutableInteger count = entry.getValue();  
522 for (int index = count.value; index > 0; index--) {  
523 // unsafe, will throw ArrayStoreException if types are not compatible, see javadoc  
524 @SuppressWarnings("unchecked")  
525 final T unchecked = (T) current;  
526 array[i++] = unchecked;  
527 }  
528 }  
529 while (i < array.length) {  
530 array[i++] = null;  
531 }  
532 return array;  
533 }  
534  
535 //-----------------------------------------------------------------------  
536 @Override  
537 public boolean equals(final Object object) {  
538 if (object == this) {  
539 return true;  
540 }  
541 if (object instanceof MultiSet == false) {  
542 return false;  
543 }  
544 final MultiSet<?> other = (MultiSet<?>) object;  
545 if (other.size() != size()) {  
546 return false;  
547 }  
548 for (final E element : map.keySet()) {  
549 if (other.getCount(element) != getCount(element)) {  
550 return false;  
551 }  
552 }  
553 return true;  
554 }  
555  
556 @Override  
557 public int hashCode() {  
558 int total = 0;  
559 for (final Map.Entry<E, MutableInteger> entry : map.entrySet()) {  
560 final E element = entry.getKey();  
561 final MutableInteger count = entry.getValue();  
562 total += (element == null ? 0 : element.hashCode()) ^ count.value;  
563 }  
564 return total;  
565 }  
566}