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.util.AbstractCollection;  
023import java.util.AbstractSet;  
024import java.util.Collection;  
025import java.util.Iterator;  
026import java.util.Set;  
027  
028import org.apache.commons.collections4.IteratorUtils;  
029import org.apache.commons.collections4.MultiSet;  
030import org.apache.commons.collections4.Transformer;  
031  
032/\*\*  
033 \* Abstract implementation of the {@link MultiSet} interface to simplify the  
034 \* creation of subclass implementations.  
035 \*  
036 \* @param <E> the type held in the multiset  
037 \* @since 4.1  
038 \*/  
039public abstract class AbstractMultiSet<E> extends AbstractCollection<E> implements MultiSet<E> {  
040  
041 /\*\* View of the elements \*/  
042 private transient Set<E> uniqueSet;  
043 /\*\* View of the entries \*/  
044 private transient Set<Entry<E>> entrySet;  
045  
046 /\*\*  
047 \* Constructor needed for subclass serialisation.  
048 \*/  
049 protected AbstractMultiSet() {  
050 super();  
051 }  
052  
053 //-----------------------------------------------------------------------  
054 /\*\*  
055 \* Returns the number of elements in this multiset.  
056 \*  
057 \* @return current size of the multiset  
058 \*/  
059 @Override  
060 public int size() {  
061 int totalSize = 0;  
062 for (final Entry<E> entry : entrySet()) {  
063 totalSize += entry.getCount();  
064 }  
065 return totalSize;  
066 }  
067  
068 /\*\*  
069 \* Returns the number of occurrence of the given element in this multiset by  
070 \* iterating over its entrySet.  
071 \*  
072 \* @param object the object to search for  
073 \* @return the number of occurrences of the object, zero if not found  
074 \*/  
075 @Override  
076 public int getCount(final Object object) {  
077 for (final Entry<E> entry : entrySet()) {  
078 final E element = entry.getElement();  
079 if (element == object ||  
080 element != null && element.equals(object)) {  
081 return entry.getCount();  
082 }  
083 }  
084 return 0;  
085 }  
086  
087 @Override  
088 public int setCount(final E object, final int count) {  
089 if (count < 0) {  
090 throw new IllegalArgumentException("Count must not be negative.");  
091 }  
092  
093 final int oldCount = getCount(object);  
094 if (oldCount < count) {  
095 add(object, count - oldCount);  
096 } else {  
097 remove(object, oldCount - count);  
098 }  
099 return oldCount;  
100 }  
101  
102 //-----------------------------------------------------------------------  
103 /\*\*  
104 \* Determines if the multiset contains the given element.  
105 \*  
106 \* @param object the object to search for  
107 \* @return true if the multiset contains the given element  
108 \*/  
109 @Override  
110 public boolean contains(final Object object) {  
111 return getCount(object) > 0;  
112 }  
113  
114 //-----------------------------------------------------------------------  
115 /\*\*  
116 \* Gets an iterator over the multiset elements. Elements present in the  
117 \* MultiSet more than once will be returned repeatedly.  
118 \*  
119 \* @return the iterator  
120 \*/  
121 @Override  
122 public Iterator<E> iterator() {  
123 return new MultiSetIterator<>(this);  
124 }  
125  
126 /\*\*  
127 \* Inner class iterator for the MultiSet.  
128 \*/  
129 private static class MultiSetIterator<E> implements Iterator<E> {  
130 private final AbstractMultiSet<E> parent;  
131 private final Iterator<Entry<E>> entryIterator;  
132 private Entry<E> current;  
133 private int itemCount;  
134 private boolean canRemove;  
135  
136 /\*\*  
137 \* Constructor.  
138 \*  
139 \* @param parent the parent multiset  
140 \*/  
141 public MultiSetIterator(final AbstractMultiSet<E> parent) {  
142 this.parent = parent;  
143 this.entryIterator = parent.entrySet().iterator();  
144 this.current = null;  
145 this.canRemove = false;  
146 }  
147  
148 /\*\* {@inheritDoc} \*/  
149 @Override  
150 public boolean hasNext() {  
151 return itemCount > 0 || entryIterator.hasNext();  
152 }  
153  
154 /\*\* {@inheritDoc} \*/  
155 @Override  
156 public E next() {  
157 if (itemCount == 0) {  
158 current = entryIterator.next();  
159 itemCount = current.getCount();  
160 }  
161 canRemove = true;  
162 itemCount--;  
163 return current.getElement();  
164 }  
165  
166 /\*\* {@inheritDoc} \*/  
167 @Override  
168 public void remove() {  
169 if (canRemove == false) {  
170 throw new IllegalStateException();  
171 }  
172 final int count = current.getCount();  
173 if (count > 1) {  
174 parent.remove(current.getElement());  
175 } else {  
176 entryIterator.remove();  
177 }  
178 canRemove = false;  
179 }  
180 }  
181  
182 //-----------------------------------------------------------------------  
183 @Override  
184 public boolean add(final E object) {  
185 add(object, 1);  
186 return true;  
187 }  
188  
189 @Override  
190 public int add(final E object, final int occurrences) {  
191 throw new UnsupportedOperationException();  
192 }  
193  
194 //-----------------------------------------------------------------------  
195 /\*\*  
196 \* Clears the multiset removing all elements from the entrySet.  
197 \*/  
198 @Override  
199 public void clear() {  
200 final Iterator<Entry<E>> it = entrySet().iterator();  
201 while (it.hasNext()) {  
202 it.next();  
203 it.remove();  
204 }  
205 }  
206  
207 @Override  
208 public boolean remove(final Object object) {  
209 return remove(object, 1) != 0;  
210 }  
211  
212 @Override  
213 public int remove(final Object object, final int occurrences) {  
214 throw new UnsupportedOperationException();  
215 }  
216  
217 @Override  
218 public boolean removeAll(final Collection<?> coll) {  
219 boolean result = false;  
220 final Iterator<?> i = coll.iterator();  
221 while (i.hasNext()) {  
222 final Object obj = i.next();  
223 final boolean changed = remove(obj, getCount(obj)) != 0;  
224 result = result || changed;  
225 }  
226 return result;  
227 }  
228  
229 //-----------------------------------------------------------------------  
230 /\*\*  
231 \* Returns a view of the unique elements of this multiset.  
232 \*  
233 \* @return the set of unique elements in this multiset  
234 \*/  
235 @Override  
236 public Set<E> uniqueSet() {  
237 if (uniqueSet == null) {  
238 uniqueSet = createUniqueSet();  
239 }  
240 return uniqueSet;  
241 }  
242  
243 /\*\*  
244 \* Create a new view for the set of unique elements in this multiset.  
245 \*  
246 \* @return a view of the set of unique elements  
247 \*/  
248 protected Set<E> createUniqueSet() {  
249 return new UniqueSet<>(this);  
250 }  
251  
252 /\*\*  
253 \* Creates a unique set iterator.  
254 \* Subclasses can override this to return iterators with different properties.  
255 \*  
256 \* @return the uniqueSet iterator  
257 \*/  
258 protected Iterator<E> createUniqueSetIterator() {  
259 final Transformer<Entry<E>, E> transformer = new Transformer<Entry<E>, E>() {  
260 @Override  
261 public E transform(final Entry<E> entry) {  
262 return entry.getElement();  
263 }  
264 };  
265 return IteratorUtils.transformedIterator(entrySet().iterator(), transformer);  
266 }  
267  
268 /\*\*  
269 \* Returns an unmodifiable view of the entries of this multiset.  
270 \*  
271 \* @return the set of entries in this multiset  
272 \*/  
273 @Override  
274 public Set<Entry<E>> entrySet() {  
275 if (entrySet == null) {  
276 entrySet = createEntrySet();  
277 }  
278 return entrySet;  
279 }  
280  
281 /\*\*  
282 \* Create a new view for the set of entries in this multiset.  
283 \*  
284 \* @return a view of the set of entries  
285 \*/  
286 protected Set<Entry<E>> createEntrySet() {  
287 return new EntrySet<>(this);  
288 }  
289  
290 /\*\*  
291 \* Returns the number of unique elements in this multiset.  
292 \*  
293 \* @return the number of unique elements  
294 \*/  
295 protected abstract int uniqueElements();  
296  
297 /\*\*  
298 \* Creates an entry set iterator.  
299 \* Subclasses can override this to return iterators with different properties.  
300 \*  
301 \* @return the entrySet iterator  
302 \*/  
303 protected abstract Iterator<Entry<E>> createEntrySetIterator();  
304  
305 //-----------------------------------------------------------------------  
306 /\*\*  
307 \* Inner class UniqueSet.  
308 \*/  
309 protected static class UniqueSet<E> extends AbstractSet<E> {  
310  
311 /\*\* The parent multiset \*/  
312 protected final AbstractMultiSet<E> parent;  
313  
314 /\*\*  
315 \* Constructs a new unique element view of the MultiSet.  
316 \*  
317 \* @param parent the parent MultiSet  
318 \*/  
319 protected UniqueSet(final AbstractMultiSet<E> parent) {  
320 this.parent = parent;  
321 }  
322  
323 @Override  
324 public Iterator<E> iterator() {  
325 return parent.createUniqueSetIterator();  
326 }  
327  
328 @Override  
329 public boolean contains(final Object key) {  
330 return parent.contains(key);  
331 }  
332  
333 @Override  
334 public boolean containsAll(final Collection<?> coll) {  
335 return parent.containsAll(coll);  
336 }  
337  
338 @Override  
339 public boolean remove(final Object key) {  
340 return parent.remove(key, parent.getCount(key)) != 0;  
341 }  
342  
343 @Override  
344 public int size() {  
345 return parent.uniqueElements();  
346 }  
347  
348 @Override  
349 public void clear() {  
350 parent.clear();  
351 }  
352 }  
353  
354 //-----------------------------------------------------------------------  
355 /\*\*  
356 \* Inner class EntrySet.  
357 \*/  
358 protected static class EntrySet<E> extends AbstractSet<Entry<E>> {  
359  
360 private final AbstractMultiSet<E> parent;  
361  
362 /\*\*  
363 \* Constructs a new view of the MultiSet.  
364 \*  
365 \* @param parent the parent MultiSet  
366 \*/  
367 protected EntrySet(final AbstractMultiSet<E> parent) {  
368 this.parent = parent;  
369 }  
370  
371 @Override  
372 public int size() {  
373 return parent.uniqueElements();  
374 }  
375  
376 @Override  
377 public Iterator<Entry<E>> iterator() {  
378 return parent.createEntrySetIterator();  
379 }  
380  
381 @Override  
382 public boolean contains(final Object obj) {  
383 if (obj instanceof Entry<?> == false) {  
384 return false;  
385 }  
386 final Entry<?> entry = (Entry<?>) obj;  
387 final Object element = entry.getElement();  
388 return parent.getCount(element) == entry.getCount();  
389 }  
390  
391 @Override  
392 public boolean remove(final Object obj) {  
393 if (obj instanceof Entry<?> == false) {  
394 return false;  
395 }  
396 final Entry<?> entry = (Entry<?>) obj;  
397 final Object element = entry.getElement();  
398 if (parent.contains(element)) {  
399 final int count = parent.getCount(element);  
400 if (entry.getCount() == count) {  
401 parent.remove(element, count);  
402 return true;  
403 }  
404 }  
405 return false;  
406 }  
407 }  
408  
409 /\*\*  
410 \* Inner class AbstractEntry.  
411 \*/  
412 protected static abstract class AbstractEntry<E> implements Entry<E> {  
413  
414 @Override  
415 public boolean equals(final Object object) {  
416 if (object instanceof Entry) {  
417 final Entry<?> other = (Entry<?>) object;  
418 final E element = this.getElement();  
419 final Object otherElement = other.getElement();  
420  
421 return this.getCount() == other.getCount() &&  
422 (element == otherElement ||  
423 element != null && element.equals(otherElement));  
424 }  
425 return false;  
426 }  
427  
428 @Override  
429 public int hashCode() {  
430 final E element = getElement();  
431 return ((element == null) ? 0 : element.hashCode()) ^ getCount();  
432 }  
433  
434 @Override  
435 public String toString() {  
436 return String.format("%s:%d", getElement(), getCount());  
437 }  
438  
439 }  
440  
441 //-----------------------------------------------------------------------  
442 /\*\*  
443 \* Write the multiset out using a custom routine.  
444 \* @param out the output stream  
445 \* @throws IOException any of the usual I/O related exceptions  
446 \*/  
447 protected void doWriteObject(final ObjectOutputStream out) throws IOException {  
448 out.writeInt(entrySet().size());  
449 for (final Entry<E> entry : entrySet()) {  
450 out.writeObject(entry.getElement());  
451 out.writeInt(entry.getCount());  
452 }  
453 }  
454  
455 /\*\*  
456 \* Read the multiset in using a custom routine.  
457 \* @param in the input stream  
458 \* @throws IOException any of the usual I/O related exceptions  
459 \* @throws ClassNotFoundException if the stream contains an object which class can not be loaded  
460 \* @throws ClassCastException if the stream does not contain the correct objects  
461 \*/  
462 protected void doReadObject(final ObjectInputStream in)  
463 throws IOException, ClassNotFoundException {  
464 final int entrySize = in.readInt();  
465 for (int i = 0; i < entrySize; i++) {  
466 @SuppressWarnings("unchecked") // This will fail at runtime if the stream is incorrect  
467 final E obj = (E) in.readObject();  
468 final int count = in.readInt();  
469 setCount(obj, count);  
470 }  
471 }  
472  
473 //-----------------------------------------------------------------------  
474 @Override  
475 public boolean equals(final Object object) {  
476 if (object == this) {  
477 return true;  
478 }  
479 if (object instanceof MultiSet == false) {  
480 return false;  
481 }  
482 final MultiSet<?> other = (MultiSet<?>) object;  
483 if (other.size() != size()) {  
484 return false;  
485 }  
486 for (final Entry<E> entry : entrySet()) {  
487 if (other.getCount(entry.getElement()) != getCount(entry.getElement())) {  
488 return false;  
489 }  
490 }  
491 return true;  
492 }  
493  
494 @Override  
495 public int hashCode() {  
496 return entrySet().hashCode();  
497 }  
498  
499 /\*\*  
500 \* Implement a toString() method suitable for debugging.  
501 \*  
502 \* @return a debugging toString  
503 \*/  
504 @Override  
505 public String toString() {  
506 return entrySet().toString();  
507 }  
508  
509}