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.bag;  
018  
019import java.io.IOException;  
020import java.io.ObjectInputStream;  
021import java.io.ObjectOutputStream;  
022import java.lang.reflect.Array;  
023import java.util.Collection;  
024import java.util.ConcurrentModificationException;  
025import java.util.Iterator;  
026import java.util.Map;  
027import java.util.Map.Entry;  
028import java.util.Set;  
029  
030import org.apache.commons.collections4.Bag;  
031import org.apache.commons.collections4.set.UnmodifiableSet;  
032  
033/\*\*  
034 \* Abstract implementation of the {@link Bag} interface to simplify the creation  
035 \* of subclass implementations.  
036 \* <p>  
037 \* Subclasses specify a Map implementation to use as the internal storage. The  
038 \* map will be used to map bag elements to a number; the number represents the  
039 \* number of occurrences of that element in the bag.  
040 \* </p>  
041 \*  
042 \* @param <E> the type of elements in this bag  
043 \* @since 3.0 (previously DefaultMapBag v2.0)  
044 \*/  
045public abstract class AbstractMapBag<E> implements Bag<E> {  
046  
047 /\*\* The map to use to store the data \*/  
048 private transient Map<E, MutableInteger> map;  
049 /\*\* The current total size of the bag \*/  
050 private int size;  
051 /\*\* The modification count for fail fast iterators \*/  
052 private transient int modCount;  
053 /\*\* Unique view of the elements \*/  
054 private transient Set<E> uniqueSet;  
055  
056 /\*\*  
057 \* Constructor needed for subclass serialisation.  
058 \*/  
059 protected AbstractMapBag() {  
060 super();  
061 }  
062  
063 /\*\*  
064 \* Constructor that assigns the specified Map as the backing store. The map  
065 \* must be empty and non-null.  
066 \*  
067 \* @param map the map to assign  
068 \*/  
069 protected AbstractMapBag(final Map<E, MutableInteger> map) {  
070 super();  
071 this.map = map;  
072 }  
073  
074 /\*\*  
075 \* Utility method for implementations to access the map that backs this bag.  
076 \* Not intended for interactive use outside of subclasses.  
077 \*  
078 \* @return the map being used by the Bag  
079 \*/  
080 protected Map<E, MutableInteger> getMap() {  
081 return map;  
082 }  
083  
084 //-----------------------------------------------------------------------  
085 /\*\*  
086 \* Returns the number of elements in this bag.  
087 \*  
088 \* @return current size of the bag  
089 \*/  
090 @Override  
091 public int size() {  
092 return size;  
093 }  
094  
095 /\*\*  
096 \* Returns true if the underlying map is empty.  
097 \*  
098 \* @return true if bag is empty  
099 \*/  
100 @Override  
101 public boolean isEmpty() {  
102 return map.isEmpty();  
103 }  
104  
105 /\*\*  
106 \* Returns the number of occurrence of the given element in this bag by  
107 \* looking up its count in the underlying map.  
108 \*  
109 \* @param object the object to search for  
110 \* @return the number of occurrences of the object, zero if not found  
111 \*/  
112 @Override  
113 public int getCount(final Object object) {  
114 final MutableInteger count = map.get(object);  
115 if (count != null) {  
116 return count.value;  
117 }  
118 return 0;  
119 }  
120  
121 //-----------------------------------------------------------------------  
122 /\*\*  
123 \* Determines if the bag contains the given element by checking if the  
124 \* underlying map contains the element as a key.  
125 \*  
126 \* @param object the object to search for  
127 \* @return true if the bag contains the given element  
128 \*/  
129 @Override  
130 public boolean contains(final Object object) {  
131 return map.containsKey(object);  
132 }  
133  
134 /\*\*  
135 \* Determines if the bag contains the given elements.  
136 \*  
137 \* @param coll the collection to check against  
138 \* @return <code>true</code> if the Bag contains all the collection  
139 \*/  
140 @Override  
141 public boolean containsAll(final Collection<?> coll) {  
142 if (coll instanceof Bag) {  
143 return containsAll((Bag<?>) coll);  
144 }  
145 return containsAll(new HashBag<>(coll));  
146 }  
147  
148 /\*\*  
149 \* Returns <code>true</code> if the bag contains all elements in the given  
150 \* collection, respecting cardinality.  
151 \*  
152 \* @param other the bag to check against  
153 \* @return <code>true</code> if the Bag contains all the collection  
154 \*/  
155 boolean containsAll(final Bag<?> other) {  
156 final Iterator<?> it = other.uniqueSet().iterator();  
157 while (it.hasNext()) {  
158 final Object current = it.next();  
159 if (getCount(current) < other.getCount(current)) {  
160 return false;  
161 }  
162 }  
163 return true;  
164 }  
165  
166 //-----------------------------------------------------------------------  
167 /\*\*  
168 \* Gets an iterator over the bag elements. Elements present in the Bag more  
169 \* than once will be returned repeatedly.  
170 \*  
171 \* @return the iterator  
172 \*/  
173 @Override  
174 public Iterator<E> iterator() {  
175 return new BagIterator<>(this);  
176 }  
177  
178 /\*\*  
179 \* Inner class iterator for the Bag.  
180 \*/  
181 static class BagIterator<E> implements Iterator<E> {  
182 private final AbstractMapBag<E> parent;  
183 private final Iterator<Map.Entry<E, MutableInteger>> entryIterator;  
184 private Map.Entry<E, MutableInteger> current;  
185 private int itemCount;  
186 private final int mods;  
187 private boolean canRemove;  
188  
189 /\*\*  
190 \* Constructor.  
191 \*  
192 \* @param parent the parent bag  
193 \*/  
194 public BagIterator(final AbstractMapBag<E> parent) {  
195 this.parent = parent;  
196 this.entryIterator = parent.map.entrySet().iterator();  
197 this.current = null;  
198 this.mods = parent.modCount;  
199 this.canRemove = false;  
200 }  
201  
202 /\*\* {@inheritDoc} \*/  
203 @Override  
204 public boolean hasNext() {  
205 return itemCount > 0 || entryIterator.hasNext();  
206 }  
207  
208 /\*\* {@inheritDoc} \*/  
209 @Override  
210 public E next() {  
211 if (parent.modCount != mods) {  
212 throw new ConcurrentModificationException();  
213 }  
214 if (itemCount == 0) {  
215 current = entryIterator.next();  
216 itemCount = current.getValue().value;  
217 }  
218 canRemove = true;  
219 itemCount--;  
220 return current.getKey();  
221 }  
222  
223 /\*\* {@inheritDoc} \*/  
224 @Override  
225 public void remove() {  
226 if (parent.modCount != mods) {  
227 throw new ConcurrentModificationException();  
228 }  
229 if (canRemove == false) {  
230 throw new IllegalStateException();  
231 }  
232 final MutableInteger mut = current.getValue();  
233 if (mut.value > 1) {  
234 mut.value--;  
235 } else {  
236 entryIterator.remove();  
237 }  
238 parent.size--;  
239 canRemove = false;  
240 }  
241 }  
242  
243 //-----------------------------------------------------------------------  
244 /\*\*  
245 \* Adds a new element to the bag, incrementing its count in the underlying map.  
246 \*  
247 \* @param object the object to add  
248 \* @return <code>true</code> if the object was not already in the <code>uniqueSet</code>  
249 \*/  
250 @Override  
251 public boolean add(final E object) {  
252 return add(object, 1);  
253 }  
254  
255 /\*\*  
256 \* Adds a new element to the bag, incrementing its count in the map.  
257 \*  
258 \* @param object the object to search for  
259 \* @param nCopies the number of copies to add  
260 \* @return <code>true</code> if the object was not already in the <code>uniqueSet</code>  
261 \*/  
262 @Override  
263 public boolean add(final E object, final int nCopies) {  
264 modCount++;  
265 if (nCopies > 0) {  
266 final MutableInteger mut = map.get(object);  
267 size += nCopies;  
268 if (mut == null) {  
269 map.put(object, new MutableInteger(nCopies));  
270 return true;  
271 }  
272 mut.value += nCopies;  
273 return false;  
274 }  
275 return false;  
276 }  
277  
278 /\*\*  
279 \* Invokes {@link #add(Object)} for each element in the given collection.  
280 \*  
281 \* @param coll the collection to add  
282 \* @return <code>true</code> if this call changed the bag  
283 \*/  
284 @Override  
285 public boolean addAll(final Collection<? extends E> coll) {  
286 boolean changed = false;  
287 final Iterator<? extends E> i = coll.iterator();  
288 while (i.hasNext()) {  
289 final boolean added = add(i.next());  
290 changed = changed || added;  
291 }  
292 return changed;  
293 }  
294  
295 //-----------------------------------------------------------------------  
296 /\*\*  
297 \* Clears the bag by clearing the underlying map.  
298 \*/  
299 @Override  
300 public void clear() {  
301 modCount++;  
302 map.clear();  
303 size = 0;  
304 }  
305  
306 /\*\*  
307 \* Removes all copies of the specified object from the bag.  
308 \*  
309 \* @param object the object to remove  
310 \* @return true if the bag changed  
311 \*/  
312 @Override  
313 public boolean remove(final Object object) {  
314 final MutableInteger mut = map.get(object);  
315 if (mut == null) {  
316 return false;  
317 }  
318 modCount++;  
319 map.remove(object);  
320 size -= mut.value;  
321 return true;  
322 }  
323  
324 /\*\*  
325 \* Removes a specified number of copies of an object from the bag.  
326 \*  
327 \* @param object the object to remove  
328 \* @param nCopies the number of copies to remove  
329 \* @return true if the bag changed  
330 \*/  
331 @Override  
332 public boolean remove(final Object object, final int nCopies) {  
333 final MutableInteger mut = map.get(object);  
334 if (mut == null) {  
335 return false;  
336 }  
337 if (nCopies <= 0) {  
338 return false;  
339 }  
340 modCount++;  
341 if (nCopies < mut.value) {  
342 mut.value -= nCopies;  
343 size -= nCopies;  
344 } else {  
345 map.remove(object);  
346 size -= mut.value;  
347 }  
348 return true;  
349 }  
350  
351 /\*\*  
352 \* Removes objects from the bag according to their count in the specified  
353 \* collection.  
354 \*  
355 \* @param coll the collection to use  
356 \* @return true if the bag changed  
357 \*/  
358 @Override  
359 public boolean removeAll(final Collection<?> coll) {  
360 boolean result = false;  
361 if (coll != null) {  
362 final Iterator<?> i = coll.iterator();  
363 while (i.hasNext()) {  
364 final boolean changed = remove(i.next(), 1);  
365 result = result || changed;  
366 }  
367 }  
368 return result;  
369 }  
370  
371 /\*\*  
372 \* Remove any members of the bag that are not in the given bag, respecting  
373 \* cardinality.  
374 \*  
375 \* @param coll the collection to retain  
376 \* @return true if this call changed the collection  
377 \*/  
378 @Override  
379 public boolean retainAll(final Collection<?> coll) {  
380 if (coll instanceof Bag) {  
381 return retainAll((Bag<?>) coll);  
382 }  
383 return retainAll(new HashBag<>(coll));  
384 }  
385  
386 /\*\*  
387 \* Remove any members of the bag that are not in the given bag, respecting  
388 \* cardinality.  
389 \* @see #retainAll(Collection)  
390 \*  
391 \* @param other the bag to retain  
392 \* @return <code>true</code> if this call changed the collection  
393 \*/  
394 boolean retainAll(final Bag<?> other) {  
395 boolean result = false;  
396 final Bag<E> excess = new HashBag<>();  
397 final Iterator<E> i = uniqueSet().iterator();  
398 while (i.hasNext()) {  
399 final E current = i.next();  
400 final int myCount = getCount(current);  
401 final int otherCount = other.getCount(current);  
402 if (1 <= otherCount && otherCount <= myCount) {  
403 excess.add(current, myCount - otherCount);  
404 } else {  
405 excess.add(current, myCount);  
406 }  
407 }  
408 if (!excess.isEmpty()) {  
409 result = removeAll(excess);  
410 }  
411 return result;  
412 }  
413  
414 //-----------------------------------------------------------------------  
415 /\*\*  
416 \* Mutable integer class for storing the data.  
417 \*/  
418 protected static class MutableInteger {  
419 /\*\* The value of this mutable. \*/  
420 protected int value;  
421  
422 /\*\*  
423 \* Constructor.  
424 \* @param value the initial value  
425 \*/  
426 MutableInteger(final int value) {  
427 this.value = value;  
428 }  
429  
430 @Override  
431 public boolean equals(final Object obj) {  
432 if (obj instanceof MutableInteger == false) {  
433 return false;  
434 }  
435 return ((MutableInteger) obj).value == value;  
436 }  
437  
438 @Override  
439 public int hashCode() {  
440 return value;  
441 }  
442 }  
443  
444 //-----------------------------------------------------------------------  
445 /\*\*  
446 \* Returns an array of all of this bag's elements.  
447 \*  
448 \* @return an array of all of this bag's elements  
449 \*/  
450 @Override  
451 public Object[] toArray() {  
452 final Object[] result = new Object[size()];  
453 int i = 0;  
454 final Iterator<E> it = map.keySet().iterator();  
455 while (it.hasNext()) {  
456 final E current = it.next();  
457 for (int index = getCount(current); index > 0; index--) {  
458 result[i++] = current;  
459 }  
460 }  
461 return result;  
462 }  
463  
464 /\*\*  
465 \* Returns an array of all of this bag's elements.  
466 \* If the input array has more elements than are in the bag,  
467 \* trailing elements will be set to null.  
468 \*  
469 \* @param <T> the type of the array elements  
470 \* @param array the array to populate  
471 \* @return an array of all of this bag's elements  
472 \* @throws ArrayStoreException if the runtime type of the specified array is not  
473 \* a supertype of the runtime type of the elements in this list  
474 \* @throws NullPointerException if the specified array is null  
475 \*/  
476 @Override  
477 public <T> T[] toArray(T[] array) {  
478 final int size = size();  
479 if (array.length < size) {  
480 @SuppressWarnings("unchecked") // safe as both are of type T  
481 final T[] unchecked = (T[]) Array.newInstance(array.getClass().getComponentType(), size);  
482 array = unchecked;  
483 }  
484  
485 int i = 0;  
486 final Iterator<E> it = map.keySet().iterator();  
487 while (it.hasNext()) {  
488 final E current = it.next();  
489 for (int index = getCount(current); index > 0; index--) {  
490 // unsafe, will throw ArrayStoreException if types are not compatible, see javadoc  
491 @SuppressWarnings("unchecked")  
492 final T unchecked = (T) current;  
493 array[i++] = unchecked;  
494 }  
495 }  
496 while (i < array.length) {  
497 array[i++] = null;  
498 }  
499 return array;  
500 }  
501  
502 /\*\*  
503 \* Returns an unmodifiable view of the underlying map's key set.  
504 \*  
505 \* @return the set of unique elements in this bag  
506 \*/  
507 @Override  
508 public Set<E> uniqueSet() {  
509 if (uniqueSet == null) {  
510 uniqueSet = UnmodifiableSet.<E> unmodifiableSet(map.keySet());  
511 }  
512 return uniqueSet;  
513 }  
514  
515 //-----------------------------------------------------------------------  
516 /\*\*  
517 \* Write the map out using a custom routine.  
518 \* @param out the output stream  
519 \* @throws IOException any of the usual I/O related exceptions  
520 \*/  
521 protected void doWriteObject(final ObjectOutputStream out) throws IOException {  
522 out.writeInt(map.size());  
523 for (final Entry<E, MutableInteger> entry : map.entrySet()) {  
524 out.writeObject(entry.getKey());  
525 out.writeInt(entry.getValue().value);  
526 }  
527 }  
528  
529 /\*\*  
530 \* Read the map in using a custom routine.  
531 \* @param map the map to use  
532 \* @param in the input stream  
533 \* @throws IOException any of the usual I/O related exceptions  
534 \* @throws ClassNotFoundException if the stream contains an object which class can not be loaded  
535 \* @throws ClassCastException if the stream does not contain the correct objects  
536 \*/  
537 protected void doReadObject(final Map<E, MutableInteger> map, final ObjectInputStream in)  
538 throws IOException, ClassNotFoundException {  
539 this.map = map;  
540 final int entrySize = in.readInt();  
541 for (int i = 0; i < entrySize; i++) {  
542 @SuppressWarnings("unchecked") // This will fail at runtime if the stream is incorrect  
543 final E obj = (E) in.readObject();  
544 final int count = in.readInt();  
545 map.put(obj, new MutableInteger(count));  
546 size += count;  
547 }  
548 }  
549  
550 //-----------------------------------------------------------------------  
551 /\*\*  
552 \* Compares this Bag to another. This Bag equals another Bag if it contains  
553 \* the same number of occurrences of the same elements.  
554 \*  
555 \* @param object the Bag to compare to  
556 \* @return true if equal  
557 \*/  
558 @Override  
559 public boolean equals(final Object object) {  
560 if (object == this) {  
561 return true;  
562 }  
563 if (object instanceof Bag == false) {  
564 return false;  
565 }  
566 final Bag<?> other = (Bag<?>) object;  
567 if (other.size() != size()) {  
568 return false;  
569 }  
570 for (final E element : map.keySet()) {  
571 if (other.getCount(element) != getCount(element)) {  
572 return false;  
573 }  
574 }  
575 return true;  
576 }  
577  
578 /\*\*  
579 \* Gets a hash code for the Bag compatible with the definition of equals.  
580 \* The hash code is defined as the sum total of a hash code for each  
581 \* element. The per element hash code is defined as  
582 \* <code>(e==null ? 0 : e.hashCode()) ^ noOccurances)</code>. This hash code  
583 \* is compatible with the Set interface.  
584 \*  
585 \* @return the hash code of the Bag  
586 \*/  
587 @Override  
588 public int hashCode() {  
589 int total = 0;  
590 for (final Entry<E, MutableInteger> entry : map.entrySet()) {  
591 final E element = entry.getKey();  
592 final MutableInteger count = entry.getValue();  
593 total += (element == null ? 0 : element.hashCode()) ^ count.value;  
594 }  
595 return total;  
596 }  
597  
598 /\*\*  
599 \* Implement a toString() method suitable for debugging.  
600 \*  
601 \* @return a debugging toString  
602 \*/  
603 @Override  
604 public String toString() {  
605 if (size() == 0) {  
606 return "[]";  
607 }  
608 final StringBuilder buf = new StringBuilder();  
609 buf.append('[');  
610 final Iterator<E> it = uniqueSet().iterator();  
611 while (it.hasNext()) {  
612 final Object current = it.next();  
613 final int count = getCount(current);  
614 buf.append(count);  
615 buf.append(':');  
616 buf.append(current);  
617 if (it.hasNext()) {  
618 buf.append(',');  
619 }  
620 }  
621 buf.append(']');  
622 return buf.toString();  
623 }  
624  
625}