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.util.Collection;  
023import java.util.Iterator;  
024  
025import org.apache.commons.collections4.SortedBag;  
026  
027/\*\*  
028 \* Decorates another {@link SortedBag} to comply with the Collection contract.  
029 \*  
030 \* @param <E> the type of elements in this bag  
031 \* @since 4.0  
032 \*/  
033public final class CollectionSortedBag<E> extends AbstractSortedBagDecorator<E> {  
034  
035 /\*\* Serialization version \*/  
036 private static final long serialVersionUID = -2560033712679053143L;  
037  
038 /\*\*  
039 \* Factory method to create a sorted bag that complies to the Collection contract.  
040 \*  
041 \* @param <E> the type of the elements in the bag  
042 \* @param bag the sorted bag to decorate, must not be null  
043 \* @return a SortedBag that complies to the Collection contract  
044 \* @throws NullPointerException if bag is null  
045 \*/  
046 public static <E> SortedBag<E> collectionSortedBag(final SortedBag<E> bag) {  
047 return new CollectionSortedBag<>(bag);  
048 }  
049  
050 //-----------------------------------------------------------------------  
051 /\*\*  
052 \* Constructor that wraps (not copies).  
053 \*  
054 \* @param bag the sorted bag to decorate, must not be null  
055 \* @throws NullPointerException if bag is null  
056 \*/  
057 public CollectionSortedBag(final SortedBag<E> bag) {  
058 super(bag);  
059 }  
060  
061 //-----------------------------------------------------------------------  
062 /\*\*  
063 \* Write the collection out using a custom routine.  
064 \*  
065 \* @param out the output stream  
066 \* @throws IOException if an error occurs while writing to the stream  
067 \*/  
068 private void writeObject(final ObjectOutputStream out) throws IOException {  
069 out.defaultWriteObject();  
070 out.writeObject(decorated());  
071 }  
072  
073 /\*\*  
074 \* Read the collection in using a custom routine.  
075 \*  
076 \* @param in the input stream  
077 \* @throws IOException if an error occurs while reading from the stream  
078 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
079 \* @throws ClassCastException if deserialised object has wrong type  
080 \*/  
081 @SuppressWarnings("unchecked") // will throw CCE, see Javadoc  
082 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
083 in.defaultReadObject();  
084 setCollection((Collection<E>) in.readObject());  
085 }  
086  
087 //-----------------------------------------------------------------------  
088 // Collection interface  
089 //-----------------------------------------------------------------------  
090  
091 @Override  
092 public boolean containsAll(final Collection<?> coll) {  
093 final Iterator<?> e = coll.iterator();  
094 while (e.hasNext()) {  
095 if(!contains(e.next())) {  
096 return false;  
097 }  
098 }  
099 return true;  
100 }  
101  
102 @Override  
103 public boolean add(final E object) {  
104 return add(object, 1);  
105 }  
106  
107 @Override  
108 public boolean addAll(final Collection<? extends E> coll) {  
109 boolean changed = false;  
110 final Iterator<? extends E> i = coll.iterator();  
111 while (i.hasNext()) {  
112 final boolean added = add(i.next(), 1);  
113 changed = changed || added;  
114 }  
115 return changed;  
116 }  
117  
118 @Override  
119 public boolean remove(final Object object) {  
120 return remove(object, 1);  
121 }  
122  
123 @Override  
124 public boolean removeAll(final Collection<?> coll) {  
125 if (coll != null) {  
126 boolean result = false;  
127 final Iterator<?> i = coll.iterator();  
128 while (i.hasNext()) {  
129 final Object obj = i.next();  
130 final boolean changed = remove(obj, getCount(obj));  
131 result = result || changed;  
132 }  
133 return result;  
134 }  
135 // let the decorated bag handle the case of null argument  
136 return decorated().removeAll(null);  
137 }  
138  
139 @Override  
140 public boolean retainAll(final Collection<?> coll) {  
141 if (coll != null) {  
142 boolean modified = false;  
143 final Iterator<E> e = iterator();  
144 while (e.hasNext()) {  
145 if (!coll.contains(e.next())) {  
146 e.remove();  
147 modified = true;  
148 }  
149 }  
150 return modified;  
151 }  
152 // let the decorated bag handle the case of null argument  
153 return decorated().retainAll(null);  
154 }  
155  
156 //-----------------------------------------------------------------------  
157 // Bag interface  
158 //-----------------------------------------------------------------------  
159  
160 @Override  
161 public boolean add(final E object, final int count) {  
162 decorated().add(object, count);  
163 return true;  
164 }  
165  
166}