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;  
018  
019import org.apache.commons.collections4.bag.CollectionBag;  
020import org.apache.commons.collections4.bag.HashBag;  
021import org.apache.commons.collections4.bag.PredicatedBag;  
022import org.apache.commons.collections4.bag.PredicatedSortedBag;  
023import org.apache.commons.collections4.bag.SynchronizedBag;  
024import org.apache.commons.collections4.bag.SynchronizedSortedBag;  
025import org.apache.commons.collections4.bag.TransformedBag;  
026import org.apache.commons.collections4.bag.TransformedSortedBag;  
027import org.apache.commons.collections4.bag.TreeBag;  
028import org.apache.commons.collections4.bag.UnmodifiableBag;  
029import org.apache.commons.collections4.bag.UnmodifiableSortedBag;  
030  
031/\*\*  
032 \* Provides utility methods and decorators for {@link Bag} and {@link SortedBag} instances.  
033 \*  
034 \* @since 2.1  
035 \*/  
036public class BagUtils {  
037  
038 /\*\*  
039 \* An empty unmodifiable bag.  
040 \*/  
041 @SuppressWarnings("rawtypes") // OK, empty bag is compatible with any type  
042 public static final Bag EMPTY\_BAG = UnmodifiableBag.unmodifiableBag(new HashBag<>());  
043  
044 /\*\*  
045 \* An empty unmodifiable sorted bag.  
046 \*/  
047 @SuppressWarnings("rawtypes") // OK, empty bag is compatible with any type  
048 public static final Bag EMPTY\_SORTED\_BAG =  
049 UnmodifiableSortedBag.unmodifiableSortedBag(new TreeBag<>());  
050  
051 /\*\*  
052 \* Instantiation of BagUtils is not intended or required.  
053 \*/  
054 private BagUtils() {}  
055  
056 //-----------------------------------------------------------------------  
057 /\*\*  
058 \* Returns a synchronized (thread-safe) bag backed by the given bag. In  
059 \* order to guarantee serial access, it is critical that all access to the  
060 \* backing bag is accomplished through the returned bag.  
061 \* <p>  
062 \* It is imperative that the user manually synchronize on the returned bag  
063 \* when iterating over it:  
064 \* </p>  
065 \*  
066 \* <pre>  
067 \* Bag bag = BagUtils.synchronizedBag(new HashBag());  
068 \* ...  
069 \* synchronized(bag) {  
070 \* Iterator i = bag.iterator(); // Must be in synchronized block  
071 \* while (i.hasNext())  
072 \* foo(i.next());  
073 \* }  
074 \* }  
075 \* </pre>  
076 \*  
077 \* Failure to follow this advice may result in non-deterministic behavior.  
078 \*  
079 \* @param <E> the element type  
080 \* @param bag the bag to synchronize, must not be null  
081 \* @return a synchronized bag backed by that bag  
082 \* @throws NullPointerException if the Bag is null  
083 \*/  
084 public static <E> Bag<E> synchronizedBag(final Bag<E> bag) {  
085 return SynchronizedBag.synchronizedBag(bag);  
086 }  
087  
088 /\*\*  
089 \* Returns an unmodifiable view of the given bag. Any modification attempts  
090 \* to the returned bag will raise an {@link UnsupportedOperationException}.  
091 \*  
092 \* @param <E> the element type  
093 \* @param bag the bag whose unmodifiable view is to be returned, must not be null  
094 \* @return an unmodifiable view of that bag  
095 \* @throws NullPointerException if the Bag is null  
096 \*/  
097 public static <E> Bag<E> unmodifiableBag(final Bag<? extends E> bag) {  
098 return UnmodifiableBag.unmodifiableBag(bag);  
099 }  
100  
101 /\*\*  
102 \* Returns a predicated (validating) bag backed by the given bag.  
103 \* <p>  
104 \* Only objects that pass the test in the given predicate can be added to  
105 \* the bag. Trying to add an invalid object results in an  
106 \* IllegalArgumentException. It is important not to use the original bag  
107 \* after invoking this method, as it is a backdoor for adding invalid  
108 \* objects.  
109 \* </p>  
110 \*  
111 \* @param <E> the element type  
112 \* @param bag the bag to predicate, must not be null  
113 \* @param predicate the predicate for the bag, must not be null  
114 \* @return a predicated bag backed by the given bag  
115 \* @throws NullPointerException if the Bag or Predicate is null  
116 \*/  
117 public static <E> Bag<E> predicatedBag(final Bag<E> bag, final Predicate<? super E> predicate) {  
118 return PredicatedBag.predicatedBag(bag, predicate);  
119 }  
120  
121 /\*\*  
122 \* Returns a transformed bag backed by the given bag.  
123 \* <p>  
124 \* Each object is passed through the transformer as it is added to the Bag.  
125 \* It is important not to use the original bag after invoking this method,  
126 \* as it is a backdoor for adding untransformed objects.  
127 \* </p>  
128 \* <p>  
129 \* Existing entries in the specified bag will not be transformed.  
130 \* If you want that behaviour, see {@link TransformedBag#transformedBag(Bag, Transformer)}.  
131 \* </p>  
132 \*  
133 \* @param <E> the element type  
134 \* @param bag the bag to predicate, must not be null  
135 \* @param transformer the transformer for the bag, must not be null  
136 \* @return a transformed bag backed by the given bag  
137 \* @throws NullPointerException if the Bag or Transformer is null  
138 \*/  
139 public static <E> Bag<E> transformingBag(final Bag<E> bag, final Transformer<? super E, ? extends E> transformer) {  
140 return TransformedBag.transformingBag(bag, transformer);  
141 }  
142  
143 /\*\*  
144 \* Returns a bag that complies to the Collection contract, backed by the given bag.  
145 \*  
146 \* @param <E> the element type  
147 \* @param bag the bag to decorate, must not be null  
148 \* @return a Bag that complies to the Collection contract  
149 \* @throws NullPointerException if bag is null  
150 \* @since 4.0  
151 \*/  
152 public static <E> Bag<E> collectionBag(final Bag<E> bag) {  
153 return CollectionBag.collectionBag(bag);  
154 }  
155  
156 //-----------------------------------------------------------------------  
157 /\*\*  
158 \* Returns a synchronized (thread-safe) sorted bag backed by the given  
159 \* sorted bag. In order to guarantee serial access, it is critical that all  
160 \* access to the backing bag is accomplished through the returned bag.  
161 \* <p>  
162 \* It is imperative that the user manually synchronize on the returned bag  
163 \* when iterating over it:  
164 \* </p>  
165 \*  
166 \* <pre>  
167 \* SortedBag bag = BagUtils.synchronizedSortedBag(new TreeBag());  
168 \* ...  
169 \* synchronized(bag) {  
170 \* Iterator i = bag.iterator(); // Must be in synchronized block  
171 \* while (i.hasNext())  
172 \* foo(i.next());  
173 \* }  
174 \* }  
175 \* </pre>  
176 \*  
177 \* Failure to follow this advice may result in non-deterministic behavior.  
178 \*  
179 \* @param <E> the element type  
180 \* @param bag the bag to synchronize, must not be null  
181 \* @return a synchronized bag backed by that bag  
182 \* @throws NullPointerException if the SortedBag is null  
183 \*/  
184 public static <E> SortedBag<E> synchronizedSortedBag(final SortedBag<E> bag) {  
185 return SynchronizedSortedBag.synchronizedSortedBag(bag);  
186 }  
187  
188 /\*\*  
189 \* Returns an unmodifiable view of the given sorted bag. Any modification  
190 \* attempts to the returned bag will raise an  
191 \* {@link UnsupportedOperationException}.  
192 \*  
193 \* @param <E> the element type  
194 \* @param bag the bag whose unmodifiable view is to be returned, must not be null  
195 \* @return an unmodifiable view of that bag  
196 \* @throws NullPointerException if the SortedBag is null  
197 \*/  
198 public static <E> SortedBag<E> unmodifiableSortedBag(final SortedBag<E> bag) {  
199 return UnmodifiableSortedBag.unmodifiableSortedBag(bag);  
200 }  
201  
202 /\*\*  
203 \* Returns a predicated (validating) sorted bag backed by the given sorted  
204 \* bag.  
205 \* <p>  
206 \* Only objects that pass the test in the given predicate can be added to  
207 \* the bag. Trying to add an invalid object results in an  
208 \* IllegalArgumentException. It is important not to use the original bag  
209 \* after invoking this method, as it is a backdoor for adding invalid  
210 \* objects.  
211 \* </p>  
212 \*  
213 \* @param <E> the element type  
214 \* @param bag the sorted bag to predicate, must not be null  
215 \* @param predicate the predicate for the bag, must not be null  
216 \* @return a predicated bag backed by the given bag  
217 \* @throws NullPointerException if the SortedBag or Predicate is null  
218 \*/  
219 public static <E> SortedBag<E> predicatedSortedBag(final SortedBag<E> bag,  
220 final Predicate<? super E> predicate) {  
221 return PredicatedSortedBag.predicatedSortedBag(bag, predicate);  
222 }  
223  
224 /\*\*  
225 \* Returns a transformed sorted bag backed by the given bag.  
226 \* <p>  
227 \* Each object is passed through the transformer as it is added to the Bag.  
228 \* It is important not to use the original bag after invoking this method,  
229 \* as it is a backdoor for adding untransformed objects.  
230 \* </p>  
231 \* <p>  
232 \* Existing entries in the specified bag will not be transformed.  
233 \* If you want that behaviour, see  
234 \* {@link TransformedSortedBag#transformedSortedBag(SortedBag, Transformer)}.  
235 \* </p>  
236 \*  
237 \* @param <E> the element type  
238 \* @param bag the bag to predicate, must not be null  
239 \* @param transformer the transformer for the bag, must not be null  
240 \* @return a transformed bag backed by the given bag  
241 \* @throws NullPointerException if the Bag or Transformer is null  
242 \*/  
243 public static <E> SortedBag<E> transformingSortedBag(final SortedBag<E> bag,  
244 final Transformer<? super E, ? extends E> transformer) {  
245 return TransformedSortedBag.transformingSortedBag(bag, transformer);  
246 }  
247  
248 /\*\*  
249 \* Get an empty <code>Bag</code>.  
250 \*  
251 \* @param <E> the element type  
252 \* @return an empty Bag  
253 \*/  
254 @SuppressWarnings("unchecked") // OK, empty bag is compatible with any type  
255 public static <E> Bag<E> emptyBag() {  
256 return EMPTY\_BAG;  
257 }  
258  
259 /\*\*  
260 \* Get an empty <code>SortedBag</code>.  
261 \*  
262 \* @param <E> the element type  
263 \* @return an empty sorted Bag  
264 \*/  
265 @SuppressWarnings("unchecked") // OK, empty bag is compatible with any type  
266 public static <E> SortedBag<E> emptySortedBag() {  
267 return (SortedBag<E>) EMPTY\_SORTED\_BAG;  
268 }  
269}