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.collection;  
018  
019import java.util.Collection;  
020import java.util.HashMap;  
021import java.util.Iterator;  
022import java.util.Objects;  
023import java.util.function.Predicate;  
024  
025import org.apache.commons.collections4.MultiMap;  
026import org.apache.commons.collections4.Transformer;  
027import org.apache.commons.collections4.map.MultiValueMap;  
028  
029/\*\*  
030 \* An IndexedCollection is a Map-like view onto a Collection. It accepts a  
031 \* keyTransformer to define how the keys are converted from the values.  
032 \* <p>  
033 \* Modifications made to this decorator modify the index as well as the  
034 \* decorated {@link Collection}. However, modifications to the underlying  
035 \* {@link Collection} will not update the index and it will get out of sync.  
036 \* </p>  
037 \* <p>  
038 \* If modification of the decorated {@link Collection} is unavoidable, then a  
039 \* call to {@link #reindex()} will update the index to the current contents of  
040 \* the {@link Collection}.  
041 \* </p>  
042 \*  
043 \* @param <K> the type of object in the index.  
044 \* @param <C> the type of object in the collection.  
045 \*  
046 \* @since 4.0  
047 \*/  
048public class IndexedCollection<K, C> extends AbstractCollectionDecorator<C> {  
049  
050 // TODO: replace with MultiValuedMap  
051  
052 /\*\* Serialization version \*/  
053 private static final long serialVersionUID = -5512610452568370038L;  
054  
055 /\*\* The {@link Transformer} for generating index keys. \*/  
056 private final Transformer<C, K> keyTransformer;  
057  
058 /\*\* The map of indexes to collected objects. \*/  
059 private final MultiMap<K, C> index;  
060  
061 /\*\* The uniqueness constraint for the index. \*/  
062 private final boolean uniqueIndex;  
063  
064 /\*\*  
065 \* Create an {@link IndexedCollection} for a unique index.  
066 \* <p>  
067 \* If an element is added, which maps to an existing key, an {@link IllegalArgumentException}  
068 \* will be thrown.  
069 \*  
070 \* @param <K> the index object type.  
071 \* @param <C> the collection type.  
072 \* @param coll the decorated {@link Collection}.  
073 \* @param keyTransformer the {@link Transformer} for generating index keys.  
074 \* @return the created {@link IndexedCollection}.  
075 \*/  
076 public static <K, C> IndexedCollection<K, C> uniqueIndexedCollection(final Collection<C> coll,  
077 final Transformer<C, K> keyTransformer) {  
078 return new IndexedCollection<>(coll, keyTransformer,  
079 MultiValueMap.<K, C>multiValueMap(new HashMap<K, Collection<C>>()),  
080 true);  
081 }  
082  
083 /\*\*  
084 \* Create an {@link IndexedCollection} for a non-unique index.  
085 \*  
086 \* @param <K> the index object type.  
087 \* @param <C> the collection type.  
088 \* @param coll the decorated {@link Collection}.  
089 \* @param keyTransformer the {@link Transformer} for generating index keys.  
090 \* @return the created {@link IndexedCollection}.  
091 \*/  
092 public static <K, C> IndexedCollection<K, C> nonUniqueIndexedCollection(final Collection<C> coll,  
093 final Transformer<C, K> keyTransformer) {  
094 return new IndexedCollection<>(coll, keyTransformer,  
095 MultiValueMap.<K, C>multiValueMap(new HashMap<K, Collection<C>>()),  
096 false);  
097 }  
098  
099 /\*\*  
100 \* Create a {@link IndexedCollection}.  
101 \*  
102 \* @param coll decorated {@link Collection}  
103 \* @param keyTransformer {@link Transformer} for generating index keys  
104 \* @param map map to use as index  
105 \* @param uniqueIndex if the index shall enforce uniqueness of index keys  
106 \*/  
107 public IndexedCollection(final Collection<C> coll, final Transformer<C, K> keyTransformer,  
108 final MultiMap<K, C> map, final boolean uniqueIndex) {  
109 super(coll);  
110 this.keyTransformer = keyTransformer;  
111 this.index = map;  
112 this.uniqueIndex = uniqueIndex;  
113 reindex();  
114 }  
115  
116 /\*\*  
117 \* {@inheritDoc}  
118 \*  
119 \* @throws IllegalArgumentException if the object maps to an existing key and the index  
120 \* enforces a uniqueness constraint  
121 \*/  
122 @Override  
123 public boolean add(final C object) {  
124 final boolean added = super.add(object);  
125 if (added) {  
126 addToIndex(object);  
127 }  
128 return added;  
129 }  
130  
131 @Override  
132 public boolean addAll(final Collection<? extends C> coll) {  
133 boolean changed = false;  
134 for (final C c: coll) {  
135 changed |= add(c);  
136 }  
137 return changed;  
138 }  
139  
140 @Override  
141 public void clear() {  
142 super.clear();  
143 index.clear();  
144 }  
145  
146 /\*\*  
147 \* {@inheritDoc}  
148 \* <p>  
149 \* Note: uses the index for fast lookup  
150 \*/  
151 @SuppressWarnings("unchecked")  
152 @Override  
153 public boolean contains(final Object object) {  
154 return index.containsKey(keyTransformer.transform((C) object));  
155 }  
156  
157 /\*\*  
158 \* {@inheritDoc}  
159 \* <p>  
160 \* Note: uses the index for fast lookup  
161 \*/  
162 @Override  
163 public boolean containsAll(final Collection<?> coll) {  
164 for (final Object o : coll) {  
165 if (!contains(o)) {  
166 return false;  
167 }  
168 }  
169 return true;  
170 }  
171  
172 /\*\*  
173 \* Get the element associated with the given key.  
174 \* <p>  
175 \* In case of a non-unique index, this method will return the first  
176 \* value associated with the given key. To retrieve all elements associated  
177 \* with a key, use {@link #values(Object)}.  
178 \*  
179 \* @param key key to look up  
180 \* @return element found  
181 \* @see #values(Object)  
182 \*/  
183 public C get(final K key) {  
184 @SuppressWarnings("unchecked") // index is a MultiMap which returns a Collection  
185 final Collection<C> coll = (Collection<C>) index.get(key);  
186 return coll == null ? null : coll.iterator().next();  
187 }  
188  
189 /\*\*  
190 \* Get all elements associated with the given key.  
191 \*  
192 \* @param key key to look up  
193 \* @return a collection of elements found, or null if {@code contains(key) == false}  
194 \*/  
195 @SuppressWarnings("unchecked") // index is a MultiMap which returns a Collection  
196 public Collection<C> values(final K key) {  
197 return (Collection<C>) index.get(key);  
198 }  
199  
200 /\*\*  
201 \* Clears the index and re-indexes the entire decorated {@link Collection}.  
202 \*/  
203 public void reindex() {  
204 index.clear();  
205 for (final C c : decorated()) {  
206 addToIndex(c);  
207 }  
208 }  
209  
210 @SuppressWarnings("unchecked")  
211 @Override  
212 public boolean remove(final Object object) {  
213 final boolean removed = super.remove(object);  
214 if (removed) {  
215 removeFromIndex((C) object);  
216 }  
217 return removed;  
218 }  
219  
220 /\*\*  
221 \* @since 4.4  
222 \*/  
223 @Override  
224 public boolean removeIf(final Predicate<? super C> filter) {  
225 if (Objects.isNull(filter)) {  
226 return false;  
227 }  
228 boolean changed = false;  
229 final Iterator<C> it = iterator();  
230 while (it.hasNext()) {  
231 if (filter.test(it.next())) {  
232 it.remove();  
233 changed = true;  
234 }  
235 }  
236 if (changed) {  
237 reindex();  
238 }  
239 return changed;  
240 }  
241  
242 @Override  
243 public boolean removeAll(final Collection<?> coll) {  
244 boolean changed = false;  
245 for (final Object o : coll) {  
246 changed |= remove(o);  
247 }  
248 return changed;  
249 }  
250  
251 @Override  
252 public boolean retainAll(final Collection<?> coll) {  
253 final boolean changed = super.retainAll(coll);  
254 if (changed) {  
255 reindex();  
256 }  
257 return changed;  
258 }  
259  
260 //-----------------------------------------------------------------------  
261  
262 /\*\*  
263 \* Provides checking for adding the index.  
264 \*  
265 \* @param object the object to index  
266 \* @throws IllegalArgumentException if the object maps to an existing key and the index  
267 \* enforces a uniqueness constraint  
268 \*/  
269 private void addToIndex(final C object) {  
270 final K key = keyTransformer.transform(object);  
271 if (uniqueIndex && index.containsKey(key)) {  
272 throw new IllegalArgumentException("Duplicate key in uniquely indexed collection.");  
273 }  
274 index.put(key, object);  
275 }  
276  
277 /\*\*  
278 \* Removes an object from the index.  
279 \*  
280 \* @param object the object to remove  
281 \*/  
282 private void removeFromIndex(final C object) {  
283 index.remove(keyTransformer.transform(object));  
284 }  
285  
286}