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.comparators;  
018  
019import java.io.Serializable;  
020import java.util.Comparator;  
021import java.util.HashMap;  
022import java.util.List;  
023import java.util.Map;  
024  
025/\*\*  
026 \* A Comparator which imposes a specific order on a specific set of Objects.  
027 \* Objects are presented to the FixedOrderComparator in a specified order and  
028 \* subsequent calls to {@link #compare(Object, Object) compare} yield that order.  
029 \* For example:  
030 \* <pre>  
031 \* String[] planets = {"Mercury", "Venus", "Earth", "Mars"};  
032 \* FixedOrderComparator distanceFromSun = new FixedOrderComparator(planets);  
033 \* Arrays.sort(planets); // Sort to alphabetical order  
034 \* Arrays.sort(planets, distanceFromSun); // Back to original order  
035 \* </pre>  
036 \* <p>  
037 \* Once <code>compare</code> has been called, the FixedOrderComparator is locked  
038 \* and attempts to modify it yield an UnsupportedOperationException.  
039 \* </p>  
040 \* <p>  
041 \* Instances of FixedOrderComparator are not synchronized. The class is not  
042 \* thread-safe at construction time, but it is thread-safe to perform  
043 \* multiple comparisons after all the setup operations are complete.  
044 \* </p>  
045 \* <p>  
046 \* This class is Serializable from Commons Collections 4.0.  
047 \* </p>  
048 \*  
049 \* @param <T> the type of objects compared by this comparator  
050 \* @since 3.0  
051 \*/  
052public class FixedOrderComparator<T> implements Comparator<T>, Serializable {  
053  
054 /\*\* Serialization version from Collections 4.0. \*/  
055 private static final long serialVersionUID = 82794675842863201L;  
056  
057 /\*\*  
058 \* Unknown object behavior enum.  
059 \* @since 4.0  
060 \*/  
061 public enum UnknownObjectBehavior {  
062 BEFORE, AFTER, EXCEPTION;  
063 }  
064  
065 /\*\* Internal map of object to position \*/  
066 private final Map<T, Integer> map = new HashMap<>();  
067  
068 /\*\* Counter used in determining the position in the map \*/  
069 private int counter = 0;  
070  
071 /\*\* Is the comparator locked against further change \*/  
072 private boolean isLocked = false;  
073  
074 /\*\* The behaviour in the case of an unknown object \*/  
075 private UnknownObjectBehavior unknownObjectBehavior = UnknownObjectBehavior.EXCEPTION;  
076  
077 // Constructors  
078 //-----------------------------------------------------------------------  
079 /\*\*  
080 \* Constructs an empty FixedOrderComparator.  
081 \*/  
082 public FixedOrderComparator() {  
083 super();  
084 }  
085  
086 /\*\*  
087 \* Constructs a FixedOrderComparator which uses the order of the given array  
088 \* to compare the objects.  
089 \* <p>  
090 \* The array is copied, so later changes will not affect the comparator.  
091 \*  
092 \* @param items the items that the comparator can compare in order  
093 \* @throws NullPointerException if the array is null  
094 \*/  
095 public FixedOrderComparator(final T... items) {  
096 super();  
097 if (items == null) {  
098 throw new NullPointerException("The list of items must not be null");  
099 }  
100 for (final T item : items) {  
101 add(item);  
102 }  
103 }  
104  
105 /\*\*  
106 \* Constructs a FixedOrderComparator which uses the order of the given list  
107 \* to compare the objects.  
108 \* <p>  
109 \* The list is copied, so later changes will not affect the comparator.  
110 \*  
111 \* @param items the items that the comparator can compare in order  
112 \* @throws NullPointerException if the list is null  
113 \*/  
114 public FixedOrderComparator(final List<T> items) {  
115 super();  
116 if (items == null) {  
117 throw new NullPointerException("The list of items must not be null");  
118 }  
119 for (final T t : items) {  
120 add(t);  
121 }  
122 }  
123  
124 // Bean methods / state querying methods  
125 //-----------------------------------------------------------------------  
126 /\*\*  
127 \* Returns true if modifications cannot be made to the FixedOrderComparator.  
128 \* FixedOrderComparators cannot be modified once they have performed a comparison.  
129 \*  
130 \* @return true if attempts to change the FixedOrderComparator yield an  
131 \* UnsupportedOperationException, false if it can be changed.  
132 \*/  
133 public boolean isLocked() {  
134 return isLocked;  
135 }  
136  
137 /\*\*  
138 \* Checks to see whether the comparator is now locked against further changes.  
139 \*  
140 \* @throws UnsupportedOperationException if the comparator is locked  
141 \*/  
142 protected void checkLocked() {  
143 if (isLocked()) {  
144 throw new UnsupportedOperationException("Cannot modify a FixedOrderComparator after a comparison");  
145 }  
146 }  
147  
148 /\*\*  
149 \* Gets the behavior for comparing unknown objects.  
150 \*  
151 \* @return {@link UnknownObjectBehavior}  
152 \*/  
153 public UnknownObjectBehavior getUnknownObjectBehavior() {  
154 return unknownObjectBehavior;  
155 }  
156  
157 /\*\*  
158 \* Sets the behavior for comparing unknown objects.  
159 \*  
160 \* @param unknownObjectBehavior the flag for unknown behaviour -  
161 \* UNKNOWN\_AFTER, UNKNOWN\_BEFORE or UNKNOWN\_THROW\_EXCEPTION  
162 \* @throws UnsupportedOperationException if a comparison has been performed  
163 \* @throws NullPointerException if unknownObjectBehavior is null  
164 \*/  
165 public void setUnknownObjectBehavior(final UnknownObjectBehavior unknownObjectBehavior) {  
166 checkLocked();  
167 if (unknownObjectBehavior == null) {  
168 throw new NullPointerException("Unknown object behavior must not be null");  
169 }  
170 this.unknownObjectBehavior = unknownObjectBehavior;  
171 }  
172  
173 // Methods for adding items  
174 //-----------------------------------------------------------------------  
175 /\*\*  
176 \* Adds an item, which compares as after all items known to the Comparator.  
177 \* If the item is already known to the Comparator, its old position is  
178 \* replaced with the new position.  
179 \*  
180 \* @param obj the item to be added to the Comparator.  
181 \* @return true if obj has been added for the first time, false if  
182 \* it was already known to the Comparator.  
183 \* @throws UnsupportedOperationException if a comparison has already been made  
184 \*/  
185 public boolean add(final T obj) {  
186 checkLocked();  
187 final Integer position = map.put(obj, Integer.valueOf(counter++));  
188 return position == null;  
189 }  
190  
191 /\*\*  
192 \* Adds a new item, which compares as equal to the given existing item.  
193 \*  
194 \* @param existingObj an item already in the Comparator's set of  
195 \* known objects  
196 \* @param newObj an item to be added to the Comparator's set of  
197 \* known objects  
198 \* @return true if newObj has been added for the first time, false if  
199 \* it was already known to the Comparator.  
200 \* @throws IllegalArgumentException if existingObject is not in the  
201 \* Comparator's set of known objects.  
202 \* @throws UnsupportedOperationException if a comparison has already been made  
203 \*/  
204 public boolean addAsEqual(final T existingObj, final T newObj) {  
205 checkLocked();  
206 final Integer position = map.get(existingObj);  
207 if (position == null) {  
208 throw new IllegalArgumentException(existingObj + " not known to " + this);  
209 }  
210 final Integer result = map.put(newObj, position);  
211 return result == null;  
212 }  
213  
214 // Comparator methods  
215 //-----------------------------------------------------------------------  
216 /\*\*  
217 \* Compares two objects according to the order of this Comparator.  
218 \* <p>  
219 \* It is important to note that this class will throw an IllegalArgumentException  
220 \* in the case of an unrecognized object. This is not specified in the  
221 \* Comparator interface, but is the most appropriate exception.  
222 \*  
223 \* @param obj1 the first object to compare  
224 \* @param obj2 the second object to compare  
225 \* @return negative if obj1 is less, positive if greater, zero if equal  
226 \* @throws IllegalArgumentException if obj1 or obj2 are not known  
227 \* to this Comparator and an alternative behavior has not been set  
228 \* via {@link #setUnknownObjectBehavior(UnknownObjectBehavior)}.  
229 \*/  
230 @Override  
231 public int compare(final T obj1, final T obj2) {  
232 isLocked = true;  
233 final Integer position1 = map.get(obj1);  
234 final Integer position2 = map.get(obj2);  
235 if (position1 == null || position2 == null) {  
236 switch (unknownObjectBehavior) {  
237 case BEFORE:  
238 return position1 == null ? position2 == null ? 0 : -1 : 1;  
239 case AFTER:  
240 return position1 == null ? position2 == null ? 0 : 1 : -1;  
241 case EXCEPTION:  
242 final Object unknownObj = position1 == null ? obj1 : obj2;  
243 throw new IllegalArgumentException("Attempting to compare unknown object "  
244 + unknownObj);  
245 default: //could be null  
246 throw new UnsupportedOperationException("Unknown unknownObjectBehavior: "  
247 + unknownObjectBehavior);  
248 }  
249 }  
250 return position1.compareTo(position2);  
251 }  
252  
253 //-----------------------------------------------------------------------  
254 /\*\*  
255 \* Implement a hash code for this comparator that is consistent with  
256 \* {@link #equals(Object) equals}.  
257 \*  
258 \* @return a hash code for this comparator.  
259 \*/  
260 @Override  
261 public int hashCode() {  
262 int total = 17;  
263 total = total\*37 + map.hashCode();  
264 total = total\*37 + (unknownObjectBehavior == null ? 0 : unknownObjectBehavior.hashCode());  
265 total = total\*37 + counter;  
266 total = total\*37 + (isLocked ? 0 : 1);  
267 return total;  
268 }  
269  
270 /\*\*  
271 \* Returns <code>true</code> iff <i>that</i> Object is  
272 \* is a {@link Comparator} whose ordering is known to be  
273 \* equivalent to mine.  
274 \* <p>  
275 \* This implementation returns <code>true</code>  
276 \* iff <code><i>that</i></code> is a {@link FixedOrderComparator}  
277 \* whose attributes are equal to mine.  
278 \*  
279 \* @param object the object to compare to  
280 \* @return true if equal  
281 \*/  
282 @Override  
283 public boolean equals(final Object object) {  
284 if (this == object) {  
285 return true;  
286 }  
287 if (null == object) {  
288 return false;  
289 }  
290 if (object.getClass().equals(this.getClass())) {  
291 final FixedOrderComparator<?> comp = (FixedOrderComparator<?>) object;  
292 return (null == map ? null == comp.map : map.equals(comp.map)) &&  
293 (null == unknownObjectBehavior ? null == comp.unknownObjectBehavior :  
294 unknownObjectBehavior == comp.unknownObjectBehavior &&  
295 counter == comp.counter &&  
296 isLocked == comp.isLocked &&  
297 unknownObjectBehavior == comp.unknownObjectBehavior);  
298 }  
299 return false;  
300 }  
301  
302}