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.ArrayList;  
021import java.util.BitSet;  
022import java.util.Comparator;  
023import java.util.Iterator;  
024import java.util.List;  
025  
026/\*\*  
027 \* A ComparatorChain is a Comparator that wraps one or more Comparators in  
028 \* sequence. The ComparatorChain calls each Comparator in sequence until either  
029 \* 1) any single Comparator returns a non-zero result (and that result is then  
030 \* returned), or 2) the ComparatorChain is exhausted (and zero is returned).  
031 \* This type of sorting is very similar to multi-column sorting in SQL, and this  
032 \* class allows Java classes to emulate that kind of behaviour when sorting a  
033 \* List.  
034 \* <p>  
035 \* To further facilitate SQL-like sorting, the order of any single Comparator in  
036 \* the list can be reversed.  
037 \* </p>  
038 \* <p>  
039 \* Calling a method that adds new Comparators or changes the ascend/descend sort  
040 \* <i>after compare(Object, Object) has been called</i> will result in an  
041 \* UnsupportedOperationException. However, <i>take care</i> to not alter the  
042 \* underlying List of Comparators or the BitSet that defines the sort order.  
043 \* </p>  
044 \* <p>  
045 \* Instances of ComparatorChain are not synchronized. The class is not  
046 \* thread-safe at construction time, but it <i>is</i> thread-safe to perform  
047 \* multiple comparisons after all the setup operations are complete.  
048 \* </p>  
049 \*  
050 \* @param <E> the type of objects compared by this comparator  
051 \* @since 2.0  
052 \*/  
053public class ComparatorChain<E> implements Comparator<E>, Serializable {  
054  
055 /\*\* Serialization version from Collections 2.0. \*/  
056 private static final long serialVersionUID = -721644942746081630L;  
057  
058 /\*\* The list of comparators in the chain. \*/  
059 private final List<Comparator<E>> comparatorChain;  
060 /\*\* Order - false (clear) = ascend; true (set) = descend. \*/  
061 private BitSet orderingBits = null;  
062 /\*\* Whether the chain has been "locked". \*/  
063 private boolean isLocked = false;  
064  
065 //-----------------------------------------------------------------------  
066 /\*\*  
067 \* Construct a ComparatorChain with no Comparators.  
068 \* You must add at least one Comparator before calling  
069 \* the compare(Object,Object) method, or an  
070 \* UnsupportedOperationException is thrown  
071 \*/  
072 public ComparatorChain() {  
073 this(new ArrayList<Comparator<E>>(), new BitSet());  
074 }  
075  
076 /\*\*  
077 \* Construct a ComparatorChain with a single Comparator,  
078 \* sorting in the forward order  
079 \*  
080 \* @param comparator First comparator in the Comparator chain  
081 \*/  
082 public ComparatorChain(final Comparator<E> comparator) {  
083 this(comparator, false);  
084 }  
085  
086 /\*\*  
087 \* Construct a Comparator chain with a single Comparator,  
088 \* sorting in the given order  
089 \*  
090 \* @param comparator First Comparator in the ComparatorChain  
091 \* @param reverse false = forward sort; true = reverse sort  
092 \*/  
093 public ComparatorChain(final Comparator<E> comparator, final boolean reverse) {  
094 comparatorChain = new ArrayList<>(1);  
095 comparatorChain.add(comparator);  
096 orderingBits = new BitSet(1);  
097 if (reverse == true) {  
098 orderingBits.set(0);  
099 }  
100 }  
101  
102 /\*\*  
103 \* Construct a ComparatorChain from the Comparators in the  
104 \* List. All Comparators will default to the forward  
105 \* sort order.  
106 \*  
107 \* @param list List of Comparators  
108 \* @see #ComparatorChain(List,BitSet)  
109 \*/  
110 public ComparatorChain(final List<Comparator<E>> list) {  
111 this(list, new BitSet(list.size()));  
112 }  
113  
114 /\*\*  
115 \* Construct a ComparatorChain from the Comparators in the  
116 \* given List. The sort order of each column will be  
117 \* drawn from the given BitSet. When determining the sort  
118 \* order for Comparator at index <i>i</i> in the List,  
119 \* the ComparatorChain will call BitSet.get(<i>i</i>).  
120 \* If that method returns <i>false</i>, the forward  
121 \* sort order is used; a return value of <i>true</i>  
122 \* indicates reverse sort order.  
123 \*  
124 \* @param list List of Comparators. NOTE: This constructor does not perform a  
125 \* defensive copy of the list  
126 \* @param bits Sort order for each Comparator. Extra bits are ignored,  
127 \* unless extra Comparators are added by another method.  
128 \*/  
129 public ComparatorChain(final List<Comparator<E>> list, final BitSet bits) {  
130 comparatorChain = list;  
131 orderingBits = bits;  
132 }  
133  
134 //-----------------------------------------------------------------------  
135 /\*\*  
136 \* Add a Comparator to the end of the chain using the  
137 \* forward sort order  
138 \*  
139 \* @param comparator Comparator with the forward sort order  
140 \*/  
141 public void addComparator(final Comparator<E> comparator) {  
142 addComparator(comparator, false);  
143 }  
144  
145 /\*\*  
146 \* Add a Comparator to the end of the chain using the  
147 \* given sort order  
148 \*  
149 \* @param comparator Comparator to add to the end of the chain  
150 \* @param reverse false = forward sort order; true = reverse sort order  
151 \*/  
152 public void addComparator(final Comparator<E> comparator, final boolean reverse) {  
153 checkLocked();  
154  
155 comparatorChain.add(comparator);  
156 if (reverse == true) {  
157 orderingBits.set(comparatorChain.size() - 1);  
158 }  
159 }  
160  
161 /\*\*  
162 \* Replace the Comparator at the given index, maintaining  
163 \* the existing sort order.  
164 \*  
165 \* @param index index of the Comparator to replace  
166 \* @param comparator Comparator to place at the given index  
167 \* @throws IndexOutOfBoundsException  
168 \* if index < 0 or index >= size()  
169 \*/  
170 public void setComparator(final int index, final Comparator<E> comparator) throws IndexOutOfBoundsException {  
171 setComparator(index, comparator, false);  
172 }  
173  
174 /\*\*  
175 \* Replace the Comparator at the given index in the  
176 \* ComparatorChain, using the given sort order  
177 \*  
178 \* @param index index of the Comparator to replace  
179 \* @param comparator Comparator to set  
180 \* @param reverse false = forward sort order; true = reverse sort order  
181 \*/  
182 public void setComparator(final int index, final Comparator<E> comparator, final boolean reverse) {  
183 checkLocked();  
184  
185 comparatorChain.set(index,comparator);  
186 if (reverse == true) {  
187 orderingBits.set(index);  
188 } else {  
189 orderingBits.clear(index);  
190 }  
191 }  
192  
193 /\*\*  
194 \* Change the sort order at the given index in the  
195 \* ComparatorChain to a forward sort.  
196 \*  
197 \* @param index Index of the ComparatorChain  
198 \*/  
199 public void setForwardSort(final int index) {  
200 checkLocked();  
201 orderingBits.clear(index);  
202 }  
203  
204 /\*\*  
205 \* Change the sort order at the given index in the  
206 \* ComparatorChain to a reverse sort.  
207 \*  
208 \* @param index Index of the ComparatorChain  
209 \*/  
210 public void setReverseSort(final int index) {  
211 checkLocked();  
212 orderingBits.set(index);  
213 }  
214  
215 /\*\*  
216 \* Number of Comparators in the current ComparatorChain.  
217 \*  
218 \* @return Comparator count  
219 \*/  
220 public int size() {  
221 return comparatorChain.size();  
222 }  
223  
224 /\*\*  
225 \* Determine if modifications can still be made to the  
226 \* ComparatorChain. ComparatorChains cannot be modified  
227 \* once they have performed a comparison.  
228 \*  
229 \* @return true = ComparatorChain cannot be modified; false =  
230 \* ComparatorChain can still be modified.  
231 \*/  
232 public boolean isLocked() {  
233 return isLocked;  
234 }  
235  
236 /\*\*  
237 \* Throws an exception if the {@link ComparatorChain} is locked.  
238 \*  
239 \* @throws UnsupportedOperationException if the {@link ComparatorChain} is locked  
240 \*/  
241 private void checkLocked() {  
242 if (isLocked == true) {  
243 throw new UnsupportedOperationException(  
244 "Comparator ordering cannot be changed after the first comparison is performed");  
245 }  
246 }  
247  
248 /\*\*  
249 \* Throws an exception if the {@link ComparatorChain} is empty.  
250 \*  
251 \* @throws UnsupportedOperationException if the {@link ComparatorChain} is empty  
252 \*/  
253 private void checkChainIntegrity() {  
254 if (comparatorChain.size() == 0) {  
255 throw new UnsupportedOperationException("ComparatorChains must contain at least one Comparator");  
256 }  
257 }  
258  
259 //-----------------------------------------------------------------------  
260 /\*\*  
261 \* Perform comparisons on the Objects as per  
262 \* Comparator.compare(o1,o2).  
263 \*  
264 \* @param o1 the first object to compare  
265 \* @param o2 the second object to compare  
266 \* @return -1, 0, or 1  
267 \* @throws UnsupportedOperationException if the ComparatorChain does not contain at least one Comparator  
268 \*/  
269 @Override  
270 public int compare(final E o1, final E o2) throws UnsupportedOperationException {  
271 if (isLocked == false) {  
272 checkChainIntegrity();  
273 isLocked = true;  
274 }  
275  
276 // iterate over all comparators in the chain  
277 final Iterator<Comparator<E>> comparators = comparatorChain.iterator();  
278 for (int comparatorIndex = 0; comparators.hasNext(); ++comparatorIndex) {  
279  
280 final Comparator<? super E> comparator = comparators.next();  
281 int retval = comparator.compare(o1,o2);  
282 if (retval != 0) {  
283 // invert the order if it is a reverse sort  
284 if (orderingBits.get(comparatorIndex) == true) {  
285 if (retval > 0) {  
286 retval = -1;  
287 } else {  
288 retval = 1;  
289 }  
290 }  
291 return retval;  
292 }  
293 }  
294  
295 // if comparators are exhausted, return 0  
296 return 0;  
297 }  
298  
299 //-----------------------------------------------------------------------  
300 /\*\*  
301 \* Implement a hash code for this comparator that is consistent with  
302 \* {@link #equals(Object) equals}.  
303 \*  
304 \* @return a suitable hash code  
305 \* @since 3.0  
306 \*/  
307 @Override  
308 public int hashCode() {  
309 int hash = 0;  
310 if (null != comparatorChain) {  
311 hash ^= comparatorChain.hashCode();  
312 }  
313 if (null != orderingBits) {  
314 hash ^= orderingBits.hashCode();  
315 }  
316 return hash;  
317 }  
318  
319 /\*\*  
320 \* Returns <code>true</code> iff <i>that</i> Object is  
321 \* is a {@link Comparator} whose ordering is known to be  
322 \* equivalent to mine.  
323 \* <p>  
324 \* This implementation returns <code>true</code>  
325 \* iff <code><i>object</i>.{@link Object#getClass() getClass()}</code>  
326 \* equals <code>this.getClass()</code>, and the underlying  
327 \* comparators and order bits are equal.  
328 \* Subclasses may want to override this behavior to remain consistent  
329 \* with the {@link Comparator#equals(Object)} contract.  
330 \*  
331 \* @param object the object to compare with  
332 \* @return true if equal  
333 \* @since 3.0  
334 \*/  
335 @Override  
336 public boolean equals(final Object object) {  
337 if (this == object) {  
338 return true;  
339 }  
340 if (null == object) {  
341 return false;  
342 }  
343 if (object.getClass().equals(this.getClass())) {  
344 final ComparatorChain<?> chain = (ComparatorChain<?>) object;  
345 return (null == orderingBits ? null == chain.orderingBits : orderingBits.equals(chain.orderingBits)) &&  
346 (null == comparatorChain ? null == chain.comparatorChain :  
347 comparatorChain.equals(chain.comparatorChain));  
348 }  
349 return false;  
350 }  
351  
352}