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 java.util.Collection;  
020import java.util.Comparator;  
021  
022import org.apache.commons.collections4.comparators.BooleanComparator;  
023import org.apache.commons.collections4.comparators.ComparableComparator;  
024import org.apache.commons.collections4.comparators.ComparatorChain;  
025import org.apache.commons.collections4.comparators.NullComparator;  
026import org.apache.commons.collections4.comparators.ReverseComparator;  
027import org.apache.commons.collections4.comparators.TransformingComparator;  
028  
029/\*\*  
030 \* Provides convenient static utility methods for <Code>Comparator</Code>  
031 \* objects.  
032 \* <p>  
033 \* Most of the functionality in this class can also be found in the  
034 \* <code>comparators</code> package. This class merely provides a  
035 \* convenient central place if you have use for more than one class  
036 \* in the <code>comparators</code> subpackage.  
037 \* </p>  
038 \*  
039 \* @since 2.1  
040 \*/  
041public class ComparatorUtils {  
042  
043 /\*\*  
044 \* ComparatorUtils should not normally be instantiated.  
045 \*/  
046 private ComparatorUtils() {}  
047  
048 /\*\*  
049 \* Comparator for natural sort order.  
050 \*  
051 \* @see ComparableComparator#comparableComparator()  
052 \*/  
053 @SuppressWarnings({ "rawtypes", "unchecked" }) // explicit type needed for Java 1.5 compilation  
054 public static final Comparator NATURAL\_COMPARATOR = ComparableComparator.<Comparable>comparableComparator();  
055  
056 /\*\*  
057 \* Gets a comparator that uses the natural order of the objects.  
058 \*  
059 \* @param <E> the object type to compare  
060 \* @return a comparator which uses natural order  
061 \*/  
062 @SuppressWarnings("unchecked")  
063 public static <E extends Comparable<? super E>> Comparator<E> naturalComparator() {  
064 return NATURAL\_COMPARATOR;  
065 }  
066  
067 /\*\*  
068 \* Gets a comparator that compares using an array of {@link Comparator}s, applied  
069 \* in sequence until one returns not equal or the array is exhausted.  
070 \*  
071 \* @param <E> the object type to compare  
072 \* @param comparators the comparators to use, not null or empty or containing nulls  
073 \* @return a {@link ComparatorChain} formed from the input comparators  
074 \* @throws NullPointerException if comparators array is null or contains a null  
075 \* @see ComparatorChain  
076 \*/  
077 public static <E> Comparator<E> chainedComparator(final Comparator<E>... comparators) {  
078 final ComparatorChain<E> chain = new ComparatorChain<>();  
079 for (final Comparator<E> comparator : comparators) {  
080 if (comparator == null) {  
081 throw new NullPointerException("Comparator cannot be null");  
082 }  
083 chain.addComparator(comparator);  
084 }  
085 return chain;  
086 }  
087  
088 /\*\*  
089 \* Gets a comparator that compares using a collection of {@link Comparator}s,  
090 \* applied in (default iterator) sequence until one returns not equal or the  
091 \* collection is exhausted.  
092 \*  
093 \* @param <E> the object type to compare  
094 \* @param comparators the comparators to use, not null or empty or containing nulls  
095 \* @return a {@link ComparatorChain} formed from the input comparators  
096 \* @throws NullPointerException if comparators collection is null or contains a null  
097 \* @throws ClassCastException if the comparators collection contains the wrong object type  
098 \* @see ComparatorChain  
099 \*/  
100 @SuppressWarnings("unchecked")  
101 public static <E> Comparator<E> chainedComparator(final Collection<Comparator<E>> comparators) {  
102 return chainedComparator(  
103 (Comparator<E>[]) comparators.toArray(new Comparator[comparators.size()])  
104 );  
105 }  
106  
107 /\*\*  
108 \* Gets a comparator that reverses the order of the given comparator.  
109 \*  
110 \* @param <E> the object type to compare  
111 \* @param comparator the comparator to reverse  
112 \* @return a comparator that reverses the order of the input comparator  
113 \* @see ReverseComparator  
114 \*/  
115 public static <E> Comparator<E> reversedComparator(final Comparator<E> comparator) {  
116 return new ReverseComparator<>(comparator);  
117 }  
118  
119 /\*\*  
120 \* Gets a Comparator that can sort Boolean objects.  
121 \* <p>  
122 \* The parameter specifies whether true or false is sorted first.  
123 \* </p>  
124 \* <p>  
125 \* The comparator throws NullPointerException if a null value is compared.  
126 \* </p>  
127 \*  
128 \* @param trueFirst when <code>true</code>, sort  
129 \* <code>true</code> {@link Boolean}s before  
130 \* <code>false</code> {@link Boolean}s.  
131 \* @return a comparator that sorts booleans  
132 \*/  
133 public static Comparator<Boolean> booleanComparator(final boolean trueFirst) {  
134 return BooleanComparator.booleanComparator(trueFirst);  
135 }  
136  
137 /\*\*  
138 \* Gets a Comparator that controls the comparison of <code>null</code> values.  
139 \* <p>  
140 \* The returned comparator will consider a null value to be less than  
141 \* any nonnull value, and equal to any other null value. Two nonnull  
142 \* values will be evaluated with the given comparator.  
143 \* </p>  
144 \*  
145 \* @param <E> the object type to compare  
146 \* @param comparator the comparator that wants to allow nulls  
147 \* @return a version of that comparator that allows nulls  
148 \* @see NullComparator  
149 \*/  
150 @SuppressWarnings("unchecked")  
151 public static <E> Comparator<E> nullLowComparator(Comparator<E> comparator) {  
152 if (comparator == null) {  
153 comparator = NATURAL\_COMPARATOR;  
154 }  
155 return new NullComparator<>(comparator, false);  
156 }  
157  
158 /\*\*  
159 \* Gets a Comparator that controls the comparison of <code>null</code> values.  
160 \* <p>  
161 \* The returned comparator will consider a null value to be greater than  
162 \* any nonnull value, and equal to any other null value. Two nonnull  
163 \* values will be evaluated with the given comparator.  
164 \* </p>  
165 \*  
166 \* @param <E> the object type to compare  
167 \* @param comparator the comparator that wants to allow nulls  
168 \* @return a version of that comparator that allows nulls  
169 \* @see NullComparator  
170 \*/  
171 @SuppressWarnings("unchecked")  
172 public static <E> Comparator<E> nullHighComparator(Comparator<E> comparator) {  
173 if (comparator == null) {  
174 comparator = NATURAL\_COMPARATOR;  
175 }  
176 return new NullComparator<>(comparator, true);  
177 }  
178  
179 /\*\*  
180 \* Gets a Comparator that passes transformed objects to the given comparator.  
181 \* <p>  
182 \* Objects passed to the returned comparator will first be transformed  
183 \* by the given transformer before they are compared by the given  
184 \* comparator.  
185 \* </p>  
186 \*  
187 \* @param <I> the input object type of the transformed comparator  
188 \* @param <O> the object type of the decorated comparator  
189 \* @param comparator the sort order to use  
190 \* @param transformer the transformer to use  
191 \* @return a comparator that transforms its input objects before comparing them  
192 \* @see TransformingComparator  
193 \*/  
194 @SuppressWarnings("unchecked")  
195 public static <I, O> Comparator<I> transformedComparator(Comparator<O> comparator,  
196 final Transformer<? super I, ? extends O> transformer) {  
197  
198 if (comparator == null) {  
199 comparator = NATURAL\_COMPARATOR;  
200 }  
201 return new TransformingComparator<>(transformer, comparator);  
202 }  
203  
204 /\*\*  
205 \* Returns the smaller of the given objects according to the given  
206 \* comparator, returning the second object if the comparator  
207 \* returns equal.  
208 \*  
209 \* @param <E> the object type to compare  
210 \* @param o1 the first object to compare  
211 \* @param o2 the second object to compare  
212 \* @param comparator the sort order to use  
213 \* @return the smaller of the two objects  
214 \*/  
215 @SuppressWarnings("unchecked")  
216 public static <E> E min(final E o1, final E o2, Comparator<E> comparator) {  
217 if (comparator == null) {  
218 comparator = NATURAL\_COMPARATOR;  
219 }  
220 final int c = comparator.compare(o1, o2);  
221 return c < 0 ? o1 : o2;  
222 }  
223  
224 /\*\*  
225 \* Returns the larger of the given objects according to the given  
226 \* comparator, returning the second object if the comparator  
227 \* returns equal.  
228 \*  
229 \* @param <E> the object type to compare  
230 \* @param o1 the first object to compare  
231 \* @param o2 the second object to compare  
232 \* @param comparator the sort order to use  
233 \* @return the larger of the two objects  
234 \*/  
235 @SuppressWarnings("unchecked")  
236 public static <E> E max(final E o1, final E o2, Comparator<E> comparator) {  
237 if (comparator == null) {  
238 comparator = NATURAL\_COMPARATOR;  
239 }  
240 final int c = comparator.compare(o1, o2);  
241 return c > 0 ? o1 : o2;  
242 }  
243  
244}