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.keyvalue;  
018  
019import java.io.Serializable;  
020import java.util.Arrays;  
021  
022/\*\*  
023 \* A <code>MultiKey</code> allows multiple map keys to be merged together.  
024 \* <p>  
025 \* The purpose of this class is to avoid the need to write code to handle  
026 \* maps of maps. An example might be the need to look up a file name by  
027 \* key and locale. The typical solution might be nested maps. This class  
028 \* can be used instead by creating an instance passing in the key and locale.  
029 \* </p>  
030 \* <p>  
031 \* Example usage:  
032 \* </p>  
033 \* <pre>  
034 \* // populate map with data mapping key+locale to localizedText  
035 \* Map map = new HashMap();  
036 \* MultiKey multiKey = new MultiKey(key, locale);  
037 \* map.put(multiKey, localizedText);  
038 \*  
039 \* // later retrieve the localized text  
040 \* MultiKey multiKey = new MultiKey(key, locale);  
041 \* String localizedText = (String) map.get(multiKey);  
042 \* </pre>  
043 \*  
044 \* @param <K> the type of keys  
045 \* @since 3.0  
046 \*/  
047public class MultiKey<K> implements Serializable {  
048 // This class could implement List, but that would confuse it's purpose  
049  
050 /\*\* Serialisation version \*/  
051 private static final long serialVersionUID = 4465448607415788805L;  
052  
053 /\*\* The individual keys \*/  
054 private final K[] keys;  
055 /\*\* The cached hashCode \*/  
056 private transient int hashCode;  
057  
058 /\*\*  
059 \* Constructor taking two keys.  
060 \* <p>  
061 \* The keys should be immutable  
062 \* If they are not then they must not be changed after adding to the MultiKey.  
063 \*  
064 \* @param key1 the first key  
065 \* @param key2 the second key  
066 \*/  
067 @SuppressWarnings("unchecked")  
068 public MultiKey(final K key1, final K key2) {  
069 this((K[]) new Object[] { key1, key2 }, false);  
070 }  
071  
072 /\*\*  
073 \* Constructor taking three keys.  
074 \* <p>  
075 \* The keys should be immutable  
076 \* If they are not then they must not be changed after adding to the MultiKey.  
077 \*  
078 \* @param key1 the first key  
079 \* @param key2 the second key  
080 \* @param key3 the third key  
081 \*/  
082 @SuppressWarnings("unchecked")  
083 public MultiKey(final K key1, final K key2, final K key3) {  
084 this((K[]) new Object[] {key1, key2, key3}, false);  
085 }  
086  
087 /\*\*  
088 \* Constructor taking four keys.  
089 \* <p>  
090 \* The keys should be immutable  
091 \* If they are not then they must not be changed after adding to the MultiKey.  
092 \*  
093 \* @param key1 the first key  
094 \* @param key2 the second key  
095 \* @param key3 the third key  
096 \* @param key4 the fourth key  
097 \*/  
098 @SuppressWarnings("unchecked")  
099 public MultiKey(final K key1, final K key2, final K key3, final K key4) {  
100 this((K[]) new Object[] {key1, key2, key3, key4}, false);  
101 }  
102  
103 /\*\*  
104 \* Constructor taking five keys.  
105 \* <p>  
106 \* The keys should be immutable  
107 \* If they are not then they must not be changed after adding to the MultiKey.  
108 \*  
109 \* @param key1 the first key  
110 \* @param key2 the second key  
111 \* @param key3 the third key  
112 \* @param key4 the fourth key  
113 \* @param key5 the fifth key  
114 \*/  
115 @SuppressWarnings("unchecked")  
116 public MultiKey(final K key1, final K key2, final K key3, final K key4, final K key5) {  
117 this((K[]) new Object[] {key1, key2, key3, key4, key5}, false);  
118 }  
119  
120 /\*\*  
121 \* Constructor taking an array of keys which is cloned.  
122 \* <p>  
123 \* The keys should be immutable  
124 \* If they are not then they must not be changed after adding to the MultiKey.  
125 \* <p>  
126 \* This is equivalent to <code>new MultiKey(keys, true)</code>.  
127 \*  
128 \* @param keys the array of keys, not null  
129 \* @throws IllegalArgumentException if the key array is null  
130 \*/  
131 public MultiKey(final K[] keys) {  
132 this(keys, true);  
133 }  
134  
135 /\*\*  
136 \* Constructor taking an array of keys, optionally choosing whether to clone.  
137 \* <p>  
138 \* <b>If the array is not cloned, then it must not be modified.</b>  
139 \* <p>  
140 \* This method is public for performance reasons only, to avoid a clone.  
141 \* The hashcode is calculated once here in this method.  
142 \* Therefore, changing the array passed in would not change the hashcode but  
143 \* would change the equals method, which is a bug.  
144 \* <p>  
145 \* This is the only fully safe usage of this constructor, as the object array  
146 \* is never made available in a variable:  
147 \* <pre>  
148 \* new MultiKey(new Object[] {...}, false);  
149 \* </pre>  
150 \* <p>  
151 \* The keys should be immutable  
152 \* If they are not then they must not be changed after adding to the MultiKey.  
153 \*  
154 \* @param keys the array of keys, not null  
155 \* @param makeClone true to clone the array, false to assign it  
156 \* @throws IllegalArgumentException if the key array is null  
157 \* @since 3.1  
158 \*/  
159 public MultiKey(final K[] keys, final boolean makeClone) {  
160 super();  
161 if (keys == null) {  
162 throw new IllegalArgumentException("The array of keys must not be null");  
163 }  
164 if (makeClone) {  
165 this.keys = keys.clone();  
166 } else {  
167 this.keys = keys;  
168 }  
169  
170 calculateHashCode(keys);  
171 }  
172  
173 //-----------------------------------------------------------------------  
174 /\*\*  
175 \* Gets a clone of the array of keys.  
176 \* <p>  
177 \* The keys should be immutable  
178 \* If they are not then they must not be changed.  
179 \*  
180 \* @return the individual keys  
181 \*/  
182 public K[] getKeys() {  
183 return keys.clone();  
184 }  
185  
186 /\*\*  
187 \* Gets the key at the specified index.  
188 \* <p>  
189 \* The key should be immutable.  
190 \* If it is not then it must not be changed.  
191 \*  
192 \* @param index the index to retrieve  
193 \* @return the key at the index  
194 \* @throws IndexOutOfBoundsException if the index is invalid  
195 \* @since 3.1  
196 \*/  
197 public K getKey(final int index) {  
198 return keys[index];  
199 }  
200  
201 /\*\*  
202 \* Gets the size of the list of keys.  
203 \*  
204 \* @return the size of the list of keys  
205 \* @since 3.1  
206 \*/  
207 public int size() {  
208 return keys.length;  
209 }  
210  
211 //-----------------------------------------------------------------------  
212 /\*\*  
213 \* Compares this object to another.  
214 \* <p>  
215 \* To be equal, the other object must be a <code>MultiKey</code> with the  
216 \* same number of keys which are also equal.  
217 \*  
218 \* @param other the other object to compare to  
219 \* @return true if equal  
220 \*/  
221 @Override  
222 public boolean equals(final Object other) {  
223 if (other == this) {  
224 return true;  
225 }  
226 if (other instanceof MultiKey) {  
227 final MultiKey<?> otherMulti = (MultiKey<?>) other;  
228 return Arrays.equals(keys, otherMulti.keys);  
229 }  
230 return false;  
231 }  
232  
233 /\*\*  
234 \* Gets the combined hash code that is computed from all the keys.  
235 \* <p>  
236 \* This value is computed once and then cached, so elements should not  
237 \* change their hash codes once created (note that this is the same  
238 \* constraint that would be used if the individual keys elements were  
239 \* themselves {@link java.util.Map Map} keys.  
240 \*  
241 \* @return the hash code  
242 \*/  
243 @Override  
244 public int hashCode() {  
245 return hashCode;  
246 }  
247  
248 /\*\*  
249 \* Gets a debugging string version of the key.  
250 \*  
251 \* @return a debugging string  
252 \*/  
253 @Override  
254 public String toString() {  
255 return "MultiKey" + Arrays.toString(keys);  
256 }  
257  
258 /\*\*  
259 \* Calculate the hash code of the instance using the provided keys.  
260 \* @param keys the keys to calculate the hash code for  
261 \*/  
262 private void calculateHashCode(final Object[] keys)  
263 {  
264 int total = 0;  
265 for (final Object key : keys) {  
266 if (key != null) {  
267 total ^= key.hashCode();  
268 }  
269 }  
270 hashCode = total;  
271 }  
272  
273 /\*\*  
274 \* Recalculate the hash code after deserialization. The hash code of some  
275 \* keys might have change (hash codes based on the system hash code are  
276 \* only stable for the same process).  
277 \* @return the instance with recalculated hash code  
278 \*/  
279 protected Object readResolve() {  
280 calculateHashCode(keys);  
281 return this;  
282 }  
283}