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.map;  
018  
019import java.io.IOException;  
020import java.io.ObjectInputStream;  
021import java.io.ObjectOutputStream;  
022import java.io.Serializable;  
023import java.util.Map;  
024  
025import org.apache.commons.collections4.MapIterator;  
026import org.apache.commons.collections4.keyvalue.MultiKey;  
027  
028/\*\*  
029 \* A <code>Map</code> implementation that uses multiple keys to map the value.  
030 \* <p>  
031 \* This class is the most efficient way to uses multiple keys to map to a value.  
032 \* The best way to use this class is via the additional map-style methods.  
033 \* These provide <code>get</code>, <code>containsKey</code>, <code>put</code> and  
034 \* <code>remove</code> for individual keys which operate without extra object creation.  
035 \* </p>  
036 \* <p>  
037 \* The additional methods are the main interface of this map.  
038 \* As such, you will not normally hold this map in a variable of type <code>Map</code>.  
039 \* </p>  
040 \* <p>  
041 \* The normal map methods take in and return a {@link MultiKey}.  
042 \* If you try to use <code>put()</code> with any other object type a  
043 \* <code>ClassCastException</code> is thrown. If you try to use <code>null</code> as  
044 \* the key in <code>put()</code> a <code>NullPointerException</code> is thrown.  
045 \* </p>  
046 \* <p>  
047 \* This map is implemented as a decorator of a <code>AbstractHashedMap</code> which  
048 \* enables extra behaviour to be added easily.  
049 \* </p>  
050 \* <ul>  
051 \* <li><code>MultiKeyMap.decorate(new LinkedMap())</code> creates an ordered map.  
052 \* <li><code>MultiKeyMap.decorate(new LRUMap())</code> creates an least recently used map.  
053 \* <li><code>MultiKeyMap.decorate(new ReferenceMap())</code> creates a garbage collector sensitive map.  
054 \* </ul>  
055 \* <p>  
056 \* Note that <code>IdentityMap</code> and <code>ReferenceIdentityMap</code> are unsuitable  
057 \* for use as the key comparison would work on the whole MultiKey, not the elements within.  
058 \* </p>  
059 \* <p>  
060 \* As an example, consider a least recently used cache that uses a String airline code  
061 \* and a Locale to lookup the airline's name:  
062 \* </p>  
063 \* <pre>  
064 \* private MultiKeyMap cache = MultiKeyMap.multiKeyMap(new LRUMap(50));  
065 \*  
066 \* public String getAirlineName(String code, String locale) {  
067 \* String name = (String) cache.get(code, locale);  
068 \* if (name == null) {  
069 \* name = getAirlineNameFromDB(code, locale);  
070 \* cache.put(code, locale, name);  
071 \* }  
072 \* return name;  
073 \* }  
074 \* </pre>  
075 \* <p>  
076 \* <strong>Note that MultiKeyMap is not synchronized and is not thread-safe.</strong>  
077 \* If you wish to use this map from multiple threads concurrently, you must use  
078 \* appropriate synchronization. This class may throw exceptions when accessed  
079 \* by concurrent threads without synchronization.  
080 \* </p>  
081 \*  
082 \* @param <K> the type of the keys in this map  
083 \* @param <V> the type of the values in this map  
084 \* @since 3.1  
085 \*/  
086public class MultiKeyMap<K, V> extends AbstractMapDecorator<MultiKey<? extends K>, V>  
087 implements Serializable, Cloneable {  
088  
089 /\*\* Serialisation version \*/  
090 private static final long serialVersionUID = -1788199231038721040L;  
091  
092 //-----------------------------------------------------------------------  
093 /\*\*  
094 \* Decorates the specified map to add the MultiKeyMap API and fast query.  
095 \* The map must not be null and must be empty.  
096 \*  
097 \* @param <K> the key type  
098 \* @param <V> the value type  
099 \* @param map the map to decorate, not null  
100 \* @return a new multi key map  
101 \* @throws NullPointerException if map is null  
102 \* @throws IllegalArgumentException if the map is not empty  
103 \* @since 4.0  
104 \*/  
105 public static <K, V> MultiKeyMap<K, V> multiKeyMap(final AbstractHashedMap<MultiKey<? extends K>, V> map) {  
106 if (map == null) {  
107 throw new NullPointerException("Map must not be null");  
108 }  
109 if (map.size() > 0) {  
110 throw new IllegalArgumentException("Map must be empty");  
111 }  
112 return new MultiKeyMap<>(map);  
113 }  
114  
115 //-----------------------------------------------------------------------  
116 /\*\*  
117 \* Constructs a new MultiKeyMap that decorates a <code>HashedMap</code>.  
118 \*/  
119 public MultiKeyMap() {  
120 this(new HashedMap<MultiKey<? extends K>, V>());  
121 }  
122  
123 /\*\*  
124 \* Constructor that decorates the specified map and is called from  
125 \* {@link #multiKeyMap(AbstractHashedMap)}.  
126 \* The map must not be null and should be empty or only contain valid keys.  
127 \* This constructor performs no validation.  
128 \*  
129 \* @param map the map to decorate  
130 \*/  
131 protected MultiKeyMap(final AbstractHashedMap<MultiKey<? extends K>, V> map) {  
132 super(map);  
133 this.map = map;  
134 }  
135  
136 //-----------------------------------------------------------------------  
137 /\*\*  
138 \* Gets the value mapped to the specified multi-key.  
139 \*  
140 \* @param key1 the first key  
141 \* @param key2 the second key  
142 \* @return the mapped value, null if no match  
143 \*/  
144 public V get(final Object key1, final Object key2) {  
145 final int hashCode = hash(key1, key2);  
146 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry =  
147 decorated().data[decorated().hashIndex(hashCode, decorated().data.length)];  
148 while (entry != null) {  
149 if (entry.hashCode == hashCode && isEqualKey(entry, key1, key2)) {  
150 return entry.getValue();  
151 }  
152 entry = entry.next;  
153 }  
154 return null;  
155 }  
156  
157 /\*\*  
158 \* Checks whether the map contains the specified multi-key.  
159 \*  
160 \* @param key1 the first key  
161 \* @param key2 the second key  
162 \* @return true if the map contains the key  
163 \*/  
164 public boolean containsKey(final Object key1, final Object key2) {  
165 final int hashCode = hash(key1, key2);  
166 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry =  
167 decorated().data[decorated().hashIndex(hashCode, decorated().data.length)];  
168 while (entry != null) {  
169 if (entry.hashCode == hashCode && isEqualKey(entry, key1, key2)) {  
170 return true;  
171 }  
172 entry = entry.next;  
173 }  
174 return false;  
175 }  
176  
177 /\*\*  
178 \* Stores the value against the specified multi-key.  
179 \*  
180 \* @param key1 the first key  
181 \* @param key2 the second key  
182 \* @param value the value to store  
183 \* @return the value previously mapped to this combined key, null if none  
184 \*/  
185 public V put(final K key1, final K key2, final V value) {  
186 final int hashCode = hash(key1, key2);  
187 final int index = decorated().hashIndex(hashCode, decorated().data.length);  
188 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry = decorated().data[index];  
189 while (entry != null) {  
190 if (entry.hashCode == hashCode && isEqualKey(entry, key1, key2)) {  
191 final V oldValue = entry.getValue();  
192 decorated().updateEntry(entry, value);  
193 return oldValue;  
194 }  
195 entry = entry.next;  
196 }  
197 decorated().addMapping(index, hashCode, new MultiKey<>(key1, key2), value);  
198 return null;  
199 }  
200  
201 /\*\*  
202 \* Removes the specified multi-key from this map.  
203 \*  
204 \* @param key1 the first key  
205 \* @param key2 the second key  
206 \* @return the value mapped to the removed key, null if key not in map  
207 \* @since 4.0 (previous name: remove(Object, Object))  
208 \*/  
209 public V removeMultiKey(final Object key1, final Object key2) {  
210 final int hashCode = hash(key1, key2);  
211 final int index = decorated().hashIndex(hashCode, decorated().data.length);  
212 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry = decorated().data[index];  
213 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> previous = null;  
214 while (entry != null) {  
215 if (entry.hashCode == hashCode && isEqualKey(entry, key1, key2)) {  
216 final V oldValue = entry.getValue();  
217 decorated().removeMapping(entry, index, previous);  
218 return oldValue;  
219 }  
220 previous = entry;  
221 entry = entry.next;  
222 }  
223 return null;  
224 }  
225  
226 /\*\*  
227 \* Gets the hash code for the specified multi-key.  
228 \*  
229 \* @param key1 the first key  
230 \* @param key2 the second key  
231 \* @return the hash code  
232 \*/  
233 protected int hash(final Object key1, final Object key2) {  
234 int h = 0;  
235 if (key1 != null) {  
236 h ^= key1.hashCode();  
237 }  
238 if (key2 != null) {  
239 h ^= key2.hashCode();  
240 }  
241 h += ~(h << 9);  
242 h ^= h >>> 14;  
243 h += h << 4;  
244 h ^= h >>> 10;  
245 return h;  
246 }  
247  
248 /\*\*  
249 \* Is the key equal to the combined key.  
250 \*  
251 \* @param entry the entry to compare to  
252 \* @param key1 the first key  
253 \* @param key2 the second key  
254 \* @return true if the key matches  
255 \*/  
256 protected boolean isEqualKey(final AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry,  
257 final Object key1, final Object key2) {  
258 final MultiKey<? extends K> multi = entry.getKey();  
259 return  
260 multi.size() == 2 &&  
261 (key1 == multi.getKey(0) || key1 != null && key1.equals(multi.getKey(0))) &&  
262 (key2 == multi.getKey(1) || key2 != null && key2.equals(multi.getKey(1)));  
263 }  
264  
265 //-----------------------------------------------------------------------  
266 /\*\*  
267 \* Gets the value mapped to the specified multi-key.  
268 \*  
269 \* @param key1 the first key  
270 \* @param key2 the second key  
271 \* @param key3 the third key  
272 \* @return the mapped value, null if no match  
273 \*/  
274 public V get(final Object key1, final Object key2, final Object key3) {  
275 final int hashCode = hash(key1, key2, key3);  
276 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry =  
277 decorated().data[decorated().hashIndex(hashCode, decorated().data.length)];  
278 while (entry != null) {  
279 if (entry.hashCode == hashCode && isEqualKey(entry, key1, key2, key3)) {  
280 return entry.getValue();  
281 }  
282 entry = entry.next;  
283 }  
284 return null;  
285 }  
286  
287 /\*\*  
288 \* Checks whether the map contains the specified multi-key.  
289 \*  
290 \* @param key1 the first key  
291 \* @param key2 the second key  
292 \* @param key3 the third key  
293 \* @return true if the map contains the key  
294 \*/  
295 public boolean containsKey(final Object key1, final Object key2, final Object key3) {  
296 final int hashCode = hash(key1, key2, key3);  
297 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry =  
298 decorated().data[decorated().hashIndex(hashCode, decorated().data.length)];  
299 while (entry != null) {  
300 if (entry.hashCode == hashCode && isEqualKey(entry, key1, key2, key3)) {  
301 return true;  
302 }  
303 entry = entry.next;  
304 }  
305 return false;  
306 }  
307  
308 /\*\*  
309 \* Stores the value against the specified multi-key.  
310 \*  
311 \* @param key1 the first key  
312 \* @param key2 the second key  
313 \* @param key3 the third key  
314 \* @param value the value to store  
315 \* @return the value previously mapped to this combined key, null if none  
316 \*/  
317 public V put(final K key1, final K key2, final K key3, final V value) {  
318 final int hashCode = hash(key1, key2, key3);  
319 final int index = decorated().hashIndex(hashCode, decorated().data.length);  
320 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry = decorated().data[index];  
321 while (entry != null) {  
322 if (entry.hashCode == hashCode && isEqualKey(entry, key1, key2, key3)) {  
323 final V oldValue = entry.getValue();  
324 decorated().updateEntry(entry, value);  
325 return oldValue;  
326 }  
327 entry = entry.next;  
328 }  
329 decorated().addMapping(index, hashCode, new MultiKey<>(key1, key2, key3), value);  
330 return null;  
331 }  
332  
333 /\*\*  
334 \* Removes the specified multi-key from this map.  
335 \*  
336 \* @param key1 the first key  
337 \* @param key2 the second key  
338 \* @param key3 the third key  
339 \* @return the value mapped to the removed key, null if key not in map  
340 \* @since 4.0 (previous name: remove(Object, Object, Object))  
341 \*/  
342 public V removeMultiKey(final Object key1, final Object key2, final Object key3) {  
343 final int hashCode = hash(key1, key2, key3);  
344 final int index = decorated().hashIndex(hashCode, decorated().data.length);  
345 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry = decorated().data[index];  
346 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> previous = null;  
347 while (entry != null) {  
348 if (entry.hashCode == hashCode && isEqualKey(entry, key1, key2, key3)) {  
349 final V oldValue = entry.getValue();  
350 decorated().removeMapping(entry, index, previous);  
351 return oldValue;  
352 }  
353 previous = entry;  
354 entry = entry.next;  
355 }  
356 return null;  
357 }  
358  
359 /\*\*  
360 \* Gets the hash code for the specified multi-key.  
361 \*  
362 \* @param key1 the first key  
363 \* @param key2 the second key  
364 \* @param key3 the third key  
365 \* @return the hash code  
366 \*/  
367 protected int hash(final Object key1, final Object key2, final Object key3) {  
368 int h = 0;  
369 if (key1 != null) {  
370 h ^= key1.hashCode();  
371 }  
372 if (key2 != null) {  
373 h ^= key2.hashCode();  
374 }  
375 if (key3 != null) {  
376 h ^= key3.hashCode();  
377 }  
378 h += ~(h << 9);  
379 h ^= h >>> 14;  
380 h += h << 4;  
381 h ^= h >>> 10;  
382 return h;  
383 }  
384  
385 /\*\*  
386 \* Is the key equal to the combined key.  
387 \*  
388 \* @param entry the entry to compare to  
389 \* @param key1 the first key  
390 \* @param key2 the second key  
391 \* @param key3 the third key  
392 \* @return true if the key matches  
393 \*/  
394 protected boolean isEqualKey(final AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry,  
395 final Object key1, final Object key2, final Object key3) {  
396 final MultiKey<? extends K> multi = entry.getKey();  
397 return  
398 multi.size() == 3 &&  
399 (key1 == multi.getKey(0) || key1 != null && key1.equals(multi.getKey(0))) &&  
400 (key2 == multi.getKey(1) || key2 != null && key2.equals(multi.getKey(1))) &&  
401 (key3 == multi.getKey(2) || key3 != null && key3.equals(multi.getKey(2)));  
402 }  
403  
404 //-----------------------------------------------------------------------  
405 /\*\*  
406 \* Gets the value mapped to the specified multi-key.  
407 \*  
408 \* @param key1 the first key  
409 \* @param key2 the second key  
410 \* @param key3 the third key  
411 \* @param key4 the fourth key  
412 \* @return the mapped value, null if no match  
413 \*/  
414 public V get(final Object key1, final Object key2, final Object key3, final Object key4) {  
415 final int hashCode = hash(key1, key2, key3, key4);  
416 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry =  
417 decorated().data[decorated().hashIndex(hashCode, decorated().data.length)];  
418 while (entry != null) {  
419 if (entry.hashCode == hashCode && isEqualKey(entry, key1, key2, key3, key4)) {  
420 return entry.getValue();  
421 }  
422 entry = entry.next;  
423 }  
424 return null;  
425 }  
426  
427 /\*\*  
428 \* Checks whether the map contains the specified multi-key.  
429 \*  
430 \* @param key1 the first key  
431 \* @param key2 the second key  
432 \* @param key3 the third key  
433 \* @param key4 the fourth key  
434 \* @return true if the map contains the key  
435 \*/  
436 public boolean containsKey(final Object key1, final Object key2, final Object key3, final Object key4) {  
437 final int hashCode = hash(key1, key2, key3, key4);  
438 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry =  
439 decorated().data[decorated().hashIndex(hashCode, decorated().data.length)];  
440 while (entry != null) {  
441 if (entry.hashCode == hashCode && isEqualKey(entry, key1, key2, key3, key4)) {  
442 return true;  
443 }  
444 entry = entry.next;  
445 }  
446 return false;  
447 }  
448  
449 /\*\*  
450 \* Stores the value against the specified multi-key.  
451 \*  
452 \* @param key1 the first key  
453 \* @param key2 the second key  
454 \* @param key3 the third key  
455 \* @param key4 the fourth key  
456 \* @param value the value to store  
457 \* @return the value previously mapped to this combined key, null if none  
458 \*/  
459 public V put(final K key1, final K key2, final K key3, final K key4, final V value) {  
460 final int hashCode = hash(key1, key2, key3, key4);  
461 final int index = decorated().hashIndex(hashCode, decorated().data.length);  
462 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry = decorated().data[index];  
463 while (entry != null) {  
464 if (entry.hashCode == hashCode && isEqualKey(entry, key1, key2, key3, key4)) {  
465 final V oldValue = entry.getValue();  
466 decorated().updateEntry(entry, value);  
467 return oldValue;  
468 }  
469 entry = entry.next;  
470 }  
471 decorated().addMapping(index, hashCode, new MultiKey<>(key1, key2, key3, key4), value);  
472 return null;  
473 }  
474  
475 /\*\*  
476 \* Removes the specified multi-key from this map.  
477 \*  
478 \* @param key1 the first key  
479 \* @param key2 the second key  
480 \* @param key3 the third key  
481 \* @param key4 the fourth key  
482 \* @return the value mapped to the removed key, null if key not in map  
483 \* @since 4.0 (previous name: remove(Object, Object, Object, Object))  
484 \*/  
485 public V removeMultiKey(final Object key1, final Object key2, final Object key3, final Object key4) {  
486 final int hashCode = hash(key1, key2, key3, key4);  
487 final int index = decorated().hashIndex(hashCode, decorated().data.length);  
488 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry = decorated().data[index];  
489 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> previous = null;  
490 while (entry != null) {  
491 if (entry.hashCode == hashCode && isEqualKey(entry, key1, key2, key3, key4)) {  
492 final V oldValue = entry.getValue();  
493 decorated().removeMapping(entry, index, previous);  
494 return oldValue;  
495 }  
496 previous = entry;  
497 entry = entry.next;  
498 }  
499 return null;  
500 }  
501  
502 /\*\*  
503 \* Gets the hash code for the specified multi-key.  
504 \*  
505 \* @param key1 the first key  
506 \* @param key2 the second key  
507 \* @param key3 the third key  
508 \* @param key4 the fourth key  
509 \* @return the hash code  
510 \*/  
511 protected int hash(final Object key1, final Object key2, final Object key3, final Object key4) {  
512 int h = 0;  
513 if (key1 != null) {  
514 h ^= key1.hashCode();  
515 }  
516 if (key2 != null) {  
517 h ^= key2.hashCode();  
518 }  
519 if (key3 != null) {  
520 h ^= key3.hashCode();  
521 }  
522 if (key4 != null) {  
523 h ^= key4.hashCode();  
524 }  
525 h += ~(h << 9);  
526 h ^= h >>> 14;  
527 h += h << 4;  
528 h ^= h >>> 10;  
529 return h;  
530 }  
531  
532 /\*\*  
533 \* Is the key equal to the combined key.  
534 \*  
535 \* @param entry the entry to compare to  
536 \* @param key1 the first key  
537 \* @param key2 the second key  
538 \* @param key3 the third key  
539 \* @param key4 the fourth key  
540 \* @return true if the key matches  
541 \*/  
542 protected boolean isEqualKey(final AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry,  
543 final Object key1, final Object key2, final Object key3, final Object key4) {  
544 final MultiKey<? extends K> multi = entry.getKey();  
545 return  
546 multi.size() == 4 &&  
547 (key1 == multi.getKey(0) || key1 != null && key1.equals(multi.getKey(0))) &&  
548 (key2 == multi.getKey(1) || key2 != null && key2.equals(multi.getKey(1))) &&  
549 (key3 == multi.getKey(2) || key3 != null && key3.equals(multi.getKey(2))) &&  
550 (key4 == multi.getKey(3) || key4 != null && key4.equals(multi.getKey(3)));  
551 }  
552  
553 //-----------------------------------------------------------------------  
554 /\*\*  
555 \* Gets the value mapped to the specified multi-key.  
556 \*  
557 \* @param key1 the first key  
558 \* @param key2 the second key  
559 \* @param key3 the third key  
560 \* @param key4 the fourth key  
561 \* @param key5 the fifth key  
562 \* @return the mapped value, null if no match  
563 \*/  
564 public V get(final Object key1, final Object key2, final Object key3, final Object key4, final Object key5) {  
565 final int hashCode = hash(key1, key2, key3, key4, key5);  
566 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry =  
567 decorated().data[decorated().hashIndex(hashCode, decorated().data.length)];  
568 while (entry != null) {  
569 if (entry.hashCode == hashCode && isEqualKey(entry, key1, key2, key3, key4, key5)) {  
570 return entry.getValue();  
571 }  
572 entry = entry.next;  
573 }  
574 return null;  
575 }  
576  
577 /\*\*  
578 \* Checks whether the map contains the specified multi-key.  
579 \*  
580 \* @param key1 the first key  
581 \* @param key2 the second key  
582 \* @param key3 the third key  
583 \* @param key4 the fourth key  
584 \* @param key5 the fifth key  
585 \* @return true if the map contains the key  
586 \*/  
587 public boolean containsKey(final Object key1, final Object key2, final Object key3,  
588 final Object key4, final Object key5) {  
589 final int hashCode = hash(key1, key2, key3, key4, key5);  
590 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry =  
591 decorated().data[decorated().hashIndex(hashCode, decorated().data.length)];  
592 while (entry != null) {  
593 if (entry.hashCode == hashCode && isEqualKey(entry, key1, key2, key3, key4, key5)) {  
594 return true;  
595 }  
596 entry = entry.next;  
597 }  
598 return false;  
599 }  
600  
601 /\*\*  
602 \* Stores the value against the specified multi-key.  
603 \*  
604 \* @param key1 the first key  
605 \* @param key2 the second key  
606 \* @param key3 the third key  
607 \* @param key4 the fourth key  
608 \* @param key5 the fifth key  
609 \* @param value the value to store  
610 \* @return the value previously mapped to this combined key, null if none  
611 \*/  
612 public V put(final K key1, final K key2, final K key3, final K key4, final K key5, final V value) {  
613 final int hashCode = hash(key1, key2, key3, key4, key5);  
614 final int index = decorated().hashIndex(hashCode, decorated().data.length);  
615 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry = decorated().data[index];  
616 while (entry != null) {  
617 if (entry.hashCode == hashCode && isEqualKey(entry, key1, key2, key3, key4, key5)) {  
618 final V oldValue = entry.getValue();  
619 decorated().updateEntry(entry, value);  
620 return oldValue;  
621 }  
622 entry = entry.next;  
623 }  
624 decorated().addMapping(index, hashCode, new MultiKey<>(key1, key2, key3, key4, key5), value);  
625 return null;  
626 }  
627  
628 /\*\*  
629 \* Removes the specified multi-key from this map.  
630 \*  
631 \* @param key1 the first key  
632 \* @param key2 the second key  
633 \* @param key3 the third key  
634 \* @param key4 the fourth key  
635 \* @param key5 the fifth key  
636 \* @return the value mapped to the removed key, null if key not in map  
637 \* @since 4.0 (previous name: remove(Object, Object, Object, Object, Object))  
638 \*/  
639 public V removeMultiKey(final Object key1, final Object key2, final Object key3,  
640 final Object key4, final Object key5) {  
641 final int hashCode = hash(key1, key2, key3, key4, key5);  
642 final int index = decorated().hashIndex(hashCode, decorated().data.length);  
643 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry = decorated().data[index];  
644 AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> previous = null;  
645 while (entry != null) {  
646 if (entry.hashCode == hashCode && isEqualKey(entry, key1, key2, key3, key4, key5)) {  
647 final V oldValue = entry.getValue();  
648 decorated().removeMapping(entry, index, previous);  
649 return oldValue;  
650 }  
651 previous = entry;  
652 entry = entry.next;  
653 }  
654 return null;  
655 }  
656  
657 /\*\*  
658 \* Gets the hash code for the specified multi-key.  
659 \*  
660 \* @param key1 the first key  
661 \* @param key2 the second key  
662 \* @param key3 the third key  
663 \* @param key4 the fourth key  
664 \* @param key5 the fifth key  
665 \* @return the hash code  
666 \*/  
667 protected int hash(final Object key1, final Object key2, final Object key3, final Object key4, final Object key5) {  
668 int h = 0;  
669 if (key1 != null) {  
670 h ^= key1.hashCode();  
671 }  
672 if (key2 != null) {  
673 h ^= key2.hashCode();  
674 }  
675 if (key3 != null) {  
676 h ^= key3.hashCode();  
677 }  
678 if (key4 != null) {  
679 h ^= key4.hashCode();  
680 }  
681 if (key5 != null) {  
682 h ^= key5.hashCode();  
683 }  
684 h += ~(h << 9);  
685 h ^= h >>> 14;  
686 h += h << 4;  
687 h ^= h >>> 10;  
688 return h;  
689 }  
690  
691 /\*\*  
692 \* Is the key equal to the combined key.  
693 \*  
694 \* @param entry the entry to compare to  
695 \* @param key1 the first key  
696 \* @param key2 the second key  
697 \* @param key3 the third key  
698 \* @param key4 the fourth key  
699 \* @param key5 the fifth key  
700 \* @return true if the key matches  
701 \*/  
702 protected boolean isEqualKey(final AbstractHashedMap.HashEntry<MultiKey<? extends K>, V> entry,  
703 final Object key1, final Object key2, final Object key3, final Object key4, final Object key5) {  
704 final MultiKey<? extends K> multi = entry.getKey();  
705 return  
706 multi.size() == 5 &&  
707 (key1 == multi.getKey(0) || key1 != null && key1.equals(multi.getKey(0))) &&  
708 (key2 == multi.getKey(1) || key2 != null && key2.equals(multi.getKey(1))) &&  
709 (key3 == multi.getKey(2) || key3 != null && key3.equals(multi.getKey(2))) &&  
710 (key4 == multi.getKey(3) || key4 != null && key4.equals(multi.getKey(3))) &&  
711 (key5 == multi.getKey(4) || key5 != null && key5.equals(multi.getKey(4)));  
712 }  
713  
714 //-----------------------------------------------------------------------  
715 /\*\*  
716 \* Removes all mappings where the first key is that specified.  
717 \* <p>  
718 \* This method removes all the mappings where the <code>MultiKey</code>  
719 \* has one or more keys, and the first matches that specified.  
720 \*  
721 \* @param key1 the first key  
722 \* @return true if any elements were removed  
723 \*/  
724 public boolean removeAll(final Object key1) {  
725 boolean modified = false;  
726 final MapIterator<MultiKey<? extends K>, V> it = mapIterator();  
727 while (it.hasNext()) {  
728 final MultiKey<? extends K> multi = it.next();  
729 if (multi.size() >= 1 &&  
730 (key1 == null ? multi.getKey(0) == null : key1.equals(multi.getKey(0)))) {  
731 it.remove();  
732 modified = true;  
733 }  
734 }  
735 return modified;  
736 }  
737  
738 /\*\*  
739 \* Removes all mappings where the first two keys are those specified.  
740 \* <p>  
741 \* This method removes all the mappings where the <code>MultiKey</code>  
742 \* has two or more keys, and the first two match those specified.  
743 \*  
744 \* @param key1 the first key  
745 \* @param key2 the second key  
746 \* @return true if any elements were removed  
747 \*/  
748 public boolean removeAll(final Object key1, final Object key2) {  
749 boolean modified = false;  
750 final MapIterator<MultiKey<? extends K>, V> it = mapIterator();  
751 while (it.hasNext()) {  
752 final MultiKey<? extends K> multi = it.next();  
753 if (multi.size() >= 2 &&  
754 (key1 == null ? multi.getKey(0) == null : key1.equals(multi.getKey(0))) &&  
755 (key2 == null ? multi.getKey(1) == null : key2.equals(multi.getKey(1)))) {  
756 it.remove();  
757 modified = true;  
758 }  
759 }  
760 return modified;  
761 }  
762  
763 /\*\*  
764 \* Removes all mappings where the first three keys are those specified.  
765 \* <p>  
766 \* This method removes all the mappings where the <code>MultiKey</code>  
767 \* has three or more keys, and the first three match those specified.  
768 \*  
769 \* @param key1 the first key  
770 \* @param key2 the second key  
771 \* @param key3 the third key  
772 \* @return true if any elements were removed  
773 \*/  
774 public boolean removeAll(final Object key1, final Object key2, final Object key3) {  
775 boolean modified = false;  
776 final MapIterator<MultiKey<? extends K>, V> it = mapIterator();  
777 while (it.hasNext()) {  
778 final MultiKey<? extends K> multi = it.next();  
779 if (multi.size() >= 3 &&  
780 (key1 == null ? multi.getKey(0) == null : key1.equals(multi.getKey(0))) &&  
781 (key2 == null ? multi.getKey(1) == null : key2.equals(multi.getKey(1))) &&  
782 (key3 == null ? multi.getKey(2) == null : key3.equals(multi.getKey(2)))) {  
783 it.remove();  
784 modified = true;  
785 }  
786 }  
787 return modified;  
788 }  
789  
790 /\*\*  
791 \* Removes all mappings where the first four keys are those specified.  
792 \* <p>  
793 \* This method removes all the mappings where the <code>MultiKey</code>  
794 \* has four or more keys, and the first four match those specified.  
795 \*  
796 \* @param key1 the first key  
797 \* @param key2 the second key  
798 \* @param key3 the third key  
799 \* @param key4 the fourth key  
800 \* @return true if any elements were removed  
801 \*/  
802 public boolean removeAll(final Object key1, final Object key2, final Object key3, final Object key4) {  
803 boolean modified = false;  
804 final MapIterator<MultiKey<? extends K>, V> it = mapIterator();  
805 while (it.hasNext()) {  
806 final MultiKey<? extends K> multi = it.next();  
807 if (multi.size() >= 4 &&  
808 (key1 == null ? multi.getKey(0) == null : key1.equals(multi.getKey(0))) &&  
809 (key2 == null ? multi.getKey(1) == null : key2.equals(multi.getKey(1))) &&  
810 (key3 == null ? multi.getKey(2) == null : key3.equals(multi.getKey(2))) &&  
811 (key4 == null ? multi.getKey(3) == null : key4.equals(multi.getKey(3)))) {  
812 it.remove();  
813 modified = true;  
814 }  
815 }  
816 return modified;  
817 }  
818  
819 //-----------------------------------------------------------------------  
820 /\*\*  
821 \* Check to ensure that input keys are valid MultiKey objects.  
822 \*  
823 \* @param key the key to check  
824 \*/  
825 protected void checkKey(final MultiKey<?> key) {  
826 if (key == null) {  
827 throw new NullPointerException("Key must not be null");  
828 }  
829 }  
830  
831 /\*\*  
832 \* Clones the map without cloning the keys or values.  
833 \*  
834 \* @return a shallow clone  
835 \*/  
836 @SuppressWarnings("unchecked")  
837 @Override  
838 public MultiKeyMap<K, V> clone() {  
839 try {  
840 return (MultiKeyMap<K, V>) super.clone();  
841 } catch (final CloneNotSupportedException e) {  
842 throw new InternalError();  
843 }  
844 }  
845  
846 /\*\*  
847 \* Puts the key and value into the map, where the key must be a non-null  
848 \* MultiKey object.  
849 \*  
850 \* @param key the non-null MultiKey object  
851 \* @param value the value to store  
852 \* @return the previous value for the key  
853 \* @throws NullPointerException if the key is null  
854 \* @throws ClassCastException if the key is not a MultiKey  
855 \*/  
856 @Override  
857 public V put(final MultiKey<? extends K> key, final V value) {  
858 checkKey(key);  
859 return super.put(key, value);  
860 }  
861  
862 /\*\*  
863 \* Copies all of the keys and values from the specified map to this map.  
864 \* Each key must be non-null and a MultiKey object.  
865 \*  
866 \* @param mapToCopy to this map  
867 \* @throws NullPointerException if the mapToCopy or any key within is null  
868 \* @throws ClassCastException if any key in mapToCopy is not a MultiKey  
869 \*/  
870 @Override  
871 public void putAll(final Map<? extends MultiKey<? extends K>, ? extends V> mapToCopy) {  
872 for (final MultiKey<? extends K> key : mapToCopy.keySet()) {  
873 checkKey(key);  
874 }  
875 super.putAll(mapToCopy);  
876 }  
877  
878 //-----------------------------------------------------------------------  
879 @Override  
880 public MapIterator<MultiKey<? extends K>, V> mapIterator() {  
881 return decorated().mapIterator();  
882 }  
883  
884 /\*\*  
885 \* {@inheritDoc}  
886 \*/  
887 @Override  
888 protected AbstractHashedMap<MultiKey<? extends K>, V> decorated() {  
889 return (AbstractHashedMap<MultiKey<? extends K>, V>) super.decorated();  
890 }  
891  
892 //-----------------------------------------------------------------------  
893 /\*\*  
894 \* Write the map out using a custom routine.  
895 \*  
896 \* @param out the output stream  
897 \* @throws IOException if an error occurs while writing to the stream  
898 \*/  
899 private void writeObject(final ObjectOutputStream out) throws IOException {  
900 out.defaultWriteObject();  
901 out.writeObject(map);  
902 }  
903  
904 /\*\*  
905 \* Read the map in using a custom routine.  
906 \*  
907 \* @param in the input stream  
908 \* @throws IOException if an error occurs while reading from the stream  
909 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
910 \*/  
911 @SuppressWarnings("unchecked")  
912 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
913 in.defaultReadObject();  
914 map = (Map<MultiKey<? extends K>, V>) in.readObject();  
915 }  
916  
917}