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.Serializable;  
020  
021import java.util.Collection;  
022import java.util.Map;  
023import java.util.Set;  
024  
025import org.apache.commons.collections4.set.CompositeSet;  
026import org.apache.commons.collections4.CollectionUtils;  
027import org.apache.commons.collections4.collection.CompositeCollection;  
028  
029/\*\*  
030 \* Decorates a map of other maps to provide a single unified view.  
031 \* <p>  
032 \* Changes made to this map will actually be made on the decorated map.  
033 \* Add and remove operations require the use of a pluggable strategy. If no  
034 \* strategy is provided then add and remove are unsupported.  
035 \* </p>  
036 \* <p>  
037 \* <strong>Note that CompositeMap is not synchronized and is not thread-safe.</strong>  
038 \* If you wish to use this map from multiple threads concurrently, you must use  
039 \* appropriate synchronization. The simplest approach is to wrap this map  
040 \* using {@link java.util.Collections#synchronizedMap(Map)}. This class may throw  
041 \* exceptions when accessed by concurrent threads without synchronization.  
042 \* </p>  
043 \*  
044 \* @param <K> the type of the keys in this map  
045 \* @param <V> the type of the values in this map  
046 \* @since 3.0  
047 \*/  
048public class CompositeMap<K, V> extends AbstractIterableMap<K, V> implements Serializable {  
049  
050 /\*\* Serialization version \*/  
051 private static final long serialVersionUID = -6096931280583808322L;  
052  
053 /\*\* Array of all maps in the composite \*/  
054 private Map<K, V>[] composite;  
055  
056 /\*\* Handle mutation operations \*/  
057 private MapMutator<K, V> mutator;  
058  
059 /\*\*  
060 \* Create a new, empty, CompositeMap.  
061 \*/  
062 @SuppressWarnings("unchecked")  
063 public CompositeMap() {  
064 this(new Map[] {}, null);  
065 }  
066  
067 /\*\*  
068 \* Create a new CompositeMap with two composited Map instances.  
069 \*  
070 \* @param one the first Map to be composited  
071 \* @param two the second Map to be composited  
072 \* @throws IllegalArgumentException if there is a key collision  
073 \*/  
074 @SuppressWarnings("unchecked")  
075 public CompositeMap(final Map<K, V> one, final Map<K, V> two) {  
076 this(new Map[] { one, two }, null);  
077 }  
078  
079 /\*\*  
080 \* Create a new CompositeMap with two composited Map instances.  
081 \*  
082 \* @param one the first Map to be composited  
083 \* @param two the second Map to be composited  
084 \* @param mutator MapMutator to be used for mutation operations  
085 \*/  
086 @SuppressWarnings("unchecked")  
087 public CompositeMap(final Map<K, V> one, final Map<K, V> two, final MapMutator<K, V> mutator) {  
088 this(new Map[] { one, two }, mutator);  
089 }  
090  
091 /\*\*  
092 \* Create a new CompositeMap which composites all of the Map instances in the  
093 \* argument. It copies the argument array, it does not use it directly.  
094 \*  
095 \* @param composite the Maps to be composited  
096 \* @throws IllegalArgumentException if there is a key collision  
097 \*/  
098 public CompositeMap(final Map<K, V>... composite) {  
099 this(composite, null);  
100 }  
101  
102 /\*\*  
103 \* Create a new CompositeMap which composites all of the Map instances in the  
104 \* argument. It copies the argument array, it does not use it directly.  
105 \*  
106 \* @param composite Maps to be composited  
107 \* @param mutator MapMutator to be used for mutation operations  
108 \*/  
109 @SuppressWarnings("unchecked")  
110 public CompositeMap(final Map<K, V>[] composite, final MapMutator<K, V> mutator) {  
111 this.mutator = mutator;  
112 this.composite = new Map[0];  
113 for (int i = composite.length - 1; i >= 0; --i) {  
114 this.addComposited(composite[i]);  
115 }  
116 }  
117  
118 //-----------------------------------------------------------------------  
119 /\*\*  
120 \* Specify the MapMutator to be used by mutation operations.  
121 \*  
122 \* @param mutator the MapMutator to be used for mutation delegation  
123 \*/  
124 public void setMutator(final MapMutator<K, V> mutator) {  
125 this.mutator = mutator;  
126 }  
127  
128 /\*\*  
129 \* Add an additional Map to the composite.  
130 \*  
131 \* @param map the Map to be added to the composite  
132 \* @throws IllegalArgumentException if there is a key collision and there is no  
133 \* MapMutator set to handle it.  
134 \*/  
135 @SuppressWarnings("unchecked")  
136 public synchronized void addComposited(final Map<K, V> map) throws IllegalArgumentException {  
137 if (map != null) {  
138 for (int i = composite.length - 1; i >= 0; --i) {  
139 final Collection<K> intersect = CollectionUtils.intersection(this.composite[i].keySet(), map.keySet());  
140 if (intersect.size() != 0) {  
141 if (this.mutator == null) {  
142 throw new IllegalArgumentException("Key collision adding Map to CompositeMap");  
143 }  
144 this.mutator.resolveCollision(this, this.composite[i], map, intersect);  
145 }  
146 }  
147 final Map<K, V>[] temp = new Map[this.composite.length + 1];  
148 System.arraycopy(this.composite, 0, temp, 0, this.composite.length);  
149 temp[temp.length - 1] = map;  
150 this.composite = temp;  
151 }  
152 }  
153  
154 /\*\*  
155 \* Remove a Map from the composite.  
156 \*  
157 \* @param map the Map to be removed from the composite  
158 \* @return The removed Map or <code>null</code> if map is not in the composite  
159 \*/  
160 @SuppressWarnings("unchecked")  
161 public synchronized Map<K, V> removeComposited(final Map<K, V> map) {  
162 final int size = this.composite.length;  
163 for (int i = 0; i < size; ++i) {  
164 if (this.composite[i].equals(map)) {  
165 final Map<K, V>[] temp = new Map[size - 1];  
166 System.arraycopy(this.composite, 0, temp, 0, i);  
167 System.arraycopy(this.composite, i + 1, temp, i, size - i - 1);  
168 this.composite = temp;  
169 return map;  
170 }  
171 }  
172 return null;  
173 }  
174  
175 //-----------------------------------------------------------------------  
176 /\*\*  
177 \* Calls <code>clear()</code> on all composited Maps.  
178 \*  
179 \* @throws UnsupportedOperationException if any of the composited Maps do not support clear()  
180 \*/  
181 @Override  
182 public void clear() {  
183 for (int i = this.composite.length - 1; i >= 0; --i) {  
184 this.composite[i].clear();  
185 }  
186 }  
187  
188 /\*\*  
189 \* Returns {@code true} if this map contains a mapping for the specified  
190 \* key. More formally, returns {@code true} if and only if  
191 \* this map contains at a mapping for a key {@code k} such that  
192 \* {@code (key==null ? k==null : key.equals(k))}. (There can be  
193 \* at most one such mapping.)  
194 \*  
195 \* @param key key whose presence in this map is to be tested.  
196 \* @return {@code true} if this map contains a mapping for the specified  
197 \* key.  
198 \*  
199 \* @throws ClassCastException if the key is of an inappropriate type for  
200 \* this map (optional).  
201 \* @throws NullPointerException if the key is {@code null} and this map  
202 \* does not not permit {@code null} keys (optional).  
203 \*/  
204 @Override  
205 public boolean containsKey(final Object key) {  
206 for (int i = this.composite.length - 1; i >= 0; --i) {  
207 if (this.composite[i].containsKey(key)) {  
208 return true;  
209 }  
210 }  
211 return false;  
212 }  
213  
214 /\*\*  
215 \* Returns {@code true} if this map maps one or more keys to the  
216 \* specified value. More formally, returns {@code true} if and only if  
217 \* this map contains at least one mapping to a value {@code v} such that  
218 \* {@code (value==null ? v==null : value.equals(v))}. This operation  
219 \* will probably require time linear in the map size for most  
220 \* implementations of the {@code Map} interface.  
221 \*  
222 \* @param value value whose presence in this map is to be tested.  
223 \* @return {@code true} if this map maps one or more keys to the  
224 \* specified value.  
225 \* @throws ClassCastException if the value is of an inappropriate type for  
226 \* this map (optional).  
227 \* @throws NullPointerException if the value is {@code null} and this map  
228 \* does not not permit {@code null} values (optional).  
229 \*/  
230 @Override  
231 public boolean containsValue(final Object value) {  
232 for (int i = this.composite.length - 1; i >= 0; --i) {  
233 if (this.composite[i].containsValue(value)) {  
234 return true;  
235 }  
236 }  
237 return false;  
238 }  
239  
240 /\*\*  
241 \* Returns a set view of the mappings contained in this map. Each element  
242 \* in the returned set is a <code>Map.Entry</code>. The set is backed by the  
243 \* map, so changes to the map are reflected in the set, and vice-versa.  
244 \* If the map is modified while an iteration over the set is in progress,  
245 \* the results of the iteration are undefined. The set supports element  
246 \* removal, which removes the corresponding mapping from the map, via the  
247 \* {@code Iterator.remove}, {@code Set.remove}, {@code removeAll},  
248 \* {@code retainAll} and {@code clear} operations. It does not support  
249 \* the {@code add} or {@code addAll} operations.  
250 \* <p>  
251 \* This implementation returns a <code>CompositeSet</code> which  
252 \* composites the entry sets from all of the composited maps.  
253 \*  
254 \* @see CompositeSet  
255 \* @return a set view of the mappings contained in this map.  
256 \*/  
257 @Override  
258 public Set<Map.Entry<K, V>> entrySet() {  
259 final CompositeSet<Map.Entry<K, V>> entries = new CompositeSet<>();  
260 for (int i = composite.length - 1; i >= 0; --i) {  
261 entries.addComposited(composite[i].entrySet());  
262 }  
263 return entries;  
264 }  
265  
266 /\*\*  
267 \* Returns the value to which this map maps the specified key. Returns  
268 \* {@code null} if the map contains no mapping for this key. A return  
269 \* value of {@code null} does not <i>necessarily</i> indicate that the  
270 \* map contains no mapping for the key; it's also possible that the map  
271 \* explicitly maps the key to {@code null}. The {@code containsKey}  
272 \* operation may be used to distinguish these two cases.  
273 \*  
274 \* <p>More formally, if this map contains a mapping from a key  
275 \* {@code k} to a value {@code v} such that <code>(key==null ? k==null :  
276 \* key.equals(k))</code>, then this method returns {@code v}; otherwise  
277 \* it returns {@code null}. (There can be at most one such mapping.)  
278 \*  
279 \* @param key key whose associated value is to be returned.  
280 \* @return the value to which this map maps the specified key, or  
281 \* {@code null} if the map contains no mapping for this key.  
282 \*  
283 \* @throws ClassCastException if the key is of an inappropriate type for  
284 \* this map (optional).  
285 \* @throws NullPointerException key is {@code null} and this map does not  
286 \* not permit {@code null} keys (optional).  
287 \*  
288 \* @see #containsKey(Object)  
289 \*/  
290 @Override  
291 public V get(final Object key) {  
292 for (int i = this.composite.length - 1; i >= 0; --i) {  
293 if (this.composite[i].containsKey(key)) {  
294 return this.composite[i].get(key);  
295 }  
296 }  
297 return null;  
298 }  
299  
300 /\*\*  
301 \* Returns {@code true} if this map contains no key-value mappings.  
302 \*  
303 \* @return {@code true} if this map contains no key-value mappings.  
304 \*/  
305 @Override  
306 public boolean isEmpty() {  
307 for (int i = this.composite.length - 1; i >= 0; --i) {  
308 if (!this.composite[i].isEmpty()) {  
309 return false;  
310 }  
311 }  
312 return true;  
313 }  
314  
315 /\*\*  
316 \* Returns a set view of the keys contained in this map. The set is  
317 \* backed by the map, so changes to the map are reflected in the set, and  
318 \* vice-versa. If the map is modified while an iteration over the set is  
319 \* in progress, the results of the iteration are undefined. The set  
320 \* supports element removal, which removes the corresponding mapping from  
321 \* the map, via the {@code Iterator.remove}, {@code Set.remove},  
322 \* {@code removeAll} {@code retainAll}, and {@code clear} operations.  
323 \* It does not support the add or {@code addAll} operations.  
324 \* <p>  
325 \* This implementation returns a <code>CompositeSet</code> which  
326 \* composites the key sets from all of the composited maps.  
327 \*  
328 \* @return a set view of the keys contained in this map.  
329 \*/  
330 @Override  
331 public Set<K> keySet() {  
332 final CompositeSet<K> keys = new CompositeSet<>();  
333 for (int i = this.composite.length - 1; i >= 0; --i) {  
334 keys.addComposited(this.composite[i].keySet());  
335 }  
336 return keys;  
337 }  
338  
339 /\*\*  
340 \* Associates the specified value with the specified key in this map  
341 \* (optional operation). If the map previously contained a mapping for  
342 \* this key, the old value is replaced by the specified value. (A map  
343 \* {@code m} is said to contain a mapping for a key {@code k} if and only  
344 \* if {@link #containsKey(Object) m.containsKey(k)} would return  
345 \* {@code true}.))  
346 \*  
347 \* @param key key with which the specified value is to be associated.  
348 \* @param value value to be associated with the specified key.  
349 \* @return previous value associated with specified key, or {@code null}  
350 \* if there was no mapping for key. A {@code null} return can  
351 \* also indicate that the map previously associated {@code null}  
352 \* with the specified key, if the implementation supports  
353 \* {@code null} values.  
354 \*  
355 \* @throws UnsupportedOperationException if no MapMutator has been specified  
356 \* @throws ClassCastException if the class of the specified key or value  
357 \* prevents it from being stored in this map.  
358 \* @throws IllegalArgumentException if some aspect of this key or value  
359 \* prevents it from being stored in this map.  
360 \* @throws NullPointerException this map does not permit {@code null}  
361 \* keys or values, and the specified key or value is  
362 \* {@code null}.  
363 \*/  
364 @Override  
365 public V put(final K key, final V value) {  
366 if (this.mutator == null) {  
367 throw new UnsupportedOperationException("No mutator specified");  
368 }  
369 return this.mutator.put(this, this.composite, key, value);  
370 }  
371  
372 /\*\*  
373 \* Copies all of the mappings from the specified map to this map  
374 \* (optional operation). The effect of this call is equivalent to that  
375 \* of calling {@link #put(Object,Object) put(k, v)} on this map once  
376 \* for each mapping from key {@code k} to value {@code v} in the  
377 \* specified map. The behavior of this operation is unspecified if the  
378 \* specified map is modified while the operation is in progress.  
379 \*  
380 \* @param map Mappings to be stored in this map.  
381 \*  
382 \* @throws UnsupportedOperationException if the {@code putAll} method is  
383 \* not supported by this map.  
384 \*  
385 \* @throws ClassCastException if the class of a key or value in the  
386 \* specified map prevents it from being stored in this map.  
387 \*  
388 \* @throws IllegalArgumentException some aspect of a key or value in the  
389 \* specified map prevents it from being stored in this map.  
390 \* @throws NullPointerException the specified map is {@code null}, or if  
391 \* this map does not permit {@code null} keys or values, and the  
392 \* specified map contains {@code null} keys or values.  
393 \*/  
394 @Override  
395 public void putAll(final Map<? extends K, ? extends V> map) {  
396 if (this.mutator == null) {  
397 throw new UnsupportedOperationException("No mutator specified");  
398 }  
399 this.mutator.putAll(this, this.composite, map);  
400 }  
401  
402 /\*\*  
403 \* Removes the mapping for this key from this map if it is present  
404 \* (optional operation). More formally, if this map contains a mapping  
405 \* from key {@code k} to value {@code v} such that  
406 \* <code>(key==null ? k==null : key.equals(k))</code>, that mapping  
407 \* is removed. (The map can contain at most one such mapping.)  
408 \*  
409 \* <p>Returns the value to which the map previously associated the key, or  
410 \* {@code null} if the map contained no mapping for this key. (A  
411 \* {@code null} return can also indicate that the map previously  
412 \* associated {@code null} with the specified key if the implementation  
413 \* supports {@code null} values.) The map will not contain a mapping for  
414 \* the specified key once the call returns.  
415 \*  
416 \* @param key key whose mapping is to be removed from the map.  
417 \* @return previous value associated with specified key, or {@code null}  
418 \* if there was no mapping for key.  
419 \*  
420 \* @throws ClassCastException if the key is of an inappropriate type for  
421 \* the composited map (optional).  
422 \* @throws NullPointerException if the key is {@code null} and the composited map  
423 \* does not not permit {@code null} keys (optional).  
424 \* @throws UnsupportedOperationException if the {@code remove} method is  
425 \* not supported by the composited map containing the key  
426 \*/  
427 @Override  
428 public V remove(final Object key) {  
429 for (int i = this.composite.length - 1; i >= 0; --i) {  
430 if (this.composite[i].containsKey(key)) {  
431 return this.composite[i].remove(key);  
432 }  
433 }  
434 return null;  
435 }  
436  
437 /\*\*  
438 \* Returns the number of key-value mappings in this map. If the  
439 \* map contains more than {@code Integer.MAX\_VALUE} elements, returns  
440 \* {@code Integer.MAX\_VALUE}.  
441 \*  
442 \* @return the number of key-value mappings in this map.  
443 \*/  
444 @Override  
445 public int size() {  
446 int size = 0;  
447 for (int i = this.composite.length - 1; i >= 0; --i) {  
448 size += this.composite[i].size();  
449 }  
450 return size;  
451 }  
452  
453 /\*\*  
454 \* Returns a collection view of the values contained in this map. The  
455 \* collection is backed by the map, so changes to the map are reflected in  
456 \* the collection, and vice-versa. If the map is modified while an  
457 \* iteration over the collection is in progress, the results of the  
458 \* iteration are undefined. The collection supports element removal,  
459 \* which removes the corresponding mapping from the map, via the  
460 \* {@code Iterator.remove}, {@code Collection.remove},  
461 \* {@code removeAll}, {@code retainAll} and {@code clear} operations.  
462 \* It does not support the add or {@code addAll} operations.  
463 \*  
464 \* @return a collection view of the values contained in this map.  
465 \*/  
466 @Override  
467 public Collection<V> values() {  
468 final CompositeCollection<V> values = new CompositeCollection<>();  
469 for (int i = composite.length - 1; i >= 0; --i) {  
470 values.addComposited(composite[i].values());  
471 }  
472 return values;  
473 }  
474  
475 /\*\*  
476 \* Checks if this Map equals another as per the Map specification.  
477 \*  
478 \* @param obj the object to compare to  
479 \* @return true if the maps are equal  
480 \*/  
481 @Override  
482 public boolean equals(final Object obj) {  
483 if (obj instanceof Map) {  
484 final Map<?, ?> map = (Map<?, ?>) obj;  
485 return this.entrySet().equals(map.entrySet());  
486 }  
487 return false;  
488 }  
489  
490 /\*\*  
491 \* Gets a hash code for the Map as per the Map specification.  
492 \* {@inheritDoc}  
493 \*/  
494 @Override  
495 public int hashCode() {  
496 int code = 0;  
497 for (final Map.Entry<K, V> entry : entrySet()) {  
498 code += entry.hashCode();  
499 }  
500 return code;  
501 }  
502  
503 /\*\*  
504 \* This interface allows definition for all of the indeterminate  
505 \* mutators in a CompositeMap, as well as providing a hook for  
506 \* callbacks on key collisions.  
507 \*  
508 \* @param <K> the type of the keys in the map  
509 \* @param <V> the type of the values in the map  
510 \*/  
511 public interface MapMutator<K, V> extends Serializable {  
512 /\*\*  
513 \* Called when adding a new Composited Map results in a  
514 \* key collision.  
515 \*  
516 \* @param composite the CompositeMap with the collision  
517 \* @param existing the Map already in the composite which contains the  
518 \* offending key  
519 \* @param added the Map being added  
520 \* @param intersect the intersection of the keysets of the existing and added maps  
521 \*/  
522 void resolveCollision(CompositeMap<K, V> composite, Map<K, V> existing,  
523 Map<K, V> added, Collection<K> intersect);  
524  
525 /\*\*  
526 \* Called when the CompositeMap.put() method is invoked.  
527 \*  
528 \* @param map the CompositeMap which is being modified  
529 \* @param composited array of Maps in the CompositeMap being modified  
530 \* @param key key with which the specified value is to be associated.  
531 \* @param value value to be associated with the specified key.  
532 \* @return previous value associated with specified key, or {@code null}  
533 \* if there was no mapping for key. A {@code null} return can  
534 \* also indicate that the map previously associated {@code null}  
535 \* with the specified key, if the implementation supports  
536 \* {@code null} values.  
537 \*  
538 \* @throws UnsupportedOperationException if not defined  
539 \* @throws ClassCastException if the class of the specified key or value  
540 \* prevents it from being stored in this map.  
541 \* @throws IllegalArgumentException if some aspect of this key or value  
542 \* prevents it from being stored in this map.  
543 \* @throws NullPointerException this map does not permit {@code null}  
544 \* keys or values, and the specified key or value is  
545 \* {@code null}.  
546 \*/  
547 V put(CompositeMap<K, V> map, Map<K, V>[] composited, K key, V value);  
548  
549 /\*\*  
550 \* Called when the CompositeMap.putAll() method is invoked.  
551 \*  
552 \* @param map the CompositeMap which is being modified  
553 \* @param composited array of Maps in the CompositeMap being modified  
554 \* @param mapToAdd Mappings to be stored in this CompositeMap  
555 \*  
556 \* @throws UnsupportedOperationException if not defined  
557 \* @throws ClassCastException if the class of the specified key or value  
558 \* prevents it from being stored in this map.  
559 \* @throws IllegalArgumentException if some aspect of this key or value  
560 \* prevents it from being stored in this map.  
561 \* @throws NullPointerException this map does not permit {@code null}  
562 \* keys or values, and the specified key or value is  
563 \* {@code null}.  
564 \*/  
565 void putAll(CompositeMap<K, V> map, Map<K, V>[] composited,  
566 Map<? extends K, ? extends V> mapToAdd);  
567 }  
568}