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.bidimap;  
018  
019import java.util.Collection;  
020import java.util.Iterator;  
021import java.util.Map;  
022import java.util.Set;  
023import java.util.Objects;  
024import java.util.function.Predicate;  
025  
026import org.apache.commons.collections4.BidiMap;  
027import org.apache.commons.collections4.MapIterator;  
028import org.apache.commons.collections4.ResettableIterator;  
029import org.apache.commons.collections4.collection.AbstractCollectionDecorator;  
030import org.apache.commons.collections4.iterators.AbstractIteratorDecorator;  
031import org.apache.commons.collections4.keyvalue.AbstractMapEntryDecorator;  
032  
033/\*\*  
034 \* Abstract {@link BidiMap} implemented using two maps.  
035 \* <p>  
036 \* An implementation can be written simply by implementing the  
037 \* {@link #createBidiMap(Map, Map, BidiMap)} method.  
038 \* </p>  
039 \*  
040 \* @param <K> the type of the keys in the map  
041 \* @param <V> the type of the values in the map  
042 \*  
043 \* @see DualHashBidiMap  
044 \* @see DualTreeBidiMap  
045 \* @since 3.0  
046 \*/  
047public abstract class AbstractDualBidiMap<K, V> implements BidiMap<K, V> {  
048  
049 /\*\*  
050 \* Normal delegate map.  
051 \*/  
052 transient Map<K, V> normalMap;  
053  
054 /\*\*  
055 \* Reverse delegate map.  
056 \*/  
057 transient Map<V, K> reverseMap;  
058  
059 /\*\*  
060 \* Inverse view of this map.  
061 \*/  
062 transient BidiMap<V, K> inverseBidiMap = null;  
063  
064 /\*\*  
065 \* View of the keys.  
066 \*/  
067 transient Set<K> keySet = null;  
068  
069 /\*\*  
070 \* View of the values.  
071 \*/  
072 transient Set<V> values = null;  
073  
074 /\*\*  
075 \* View of the entries.  
076 \*/  
077 transient Set<Map.Entry<K, V>> entrySet = null;  
078  
079 /\*\*  
080 \* Creates an empty map, initialised by <code>createMap</code>.  
081 \* <p>  
082 \* This constructor remains in place for deserialization.  
083 \* All other usage is deprecated in favour of  
084 \* {@link #AbstractDualBidiMap(Map, Map)}.  
085 \*/  
086 protected AbstractDualBidiMap() {  
087 super();  
088 }  
089  
090 /\*\*  
091 \* Creates an empty map using the two maps specified as storage.  
092 \* <p>  
093 \* The two maps must be a matching pair, normal and reverse.  
094 \* They will typically both be empty.  
095 \* <p>  
096 \* Neither map is validated, so nulls may be passed in.  
097 \* If you choose to do this then the subclass constructor must populate  
098 \* the <code>maps[]</code> instance variable itself.  
099 \*  
100 \* @param normalMap the normal direction map  
101 \* @param reverseMap the reverse direction map  
102 \* @since 3.1  
103 \*/  
104 protected AbstractDualBidiMap(final Map<K, V> normalMap, final Map<V, K> reverseMap) {  
105 super();  
106 this.normalMap = normalMap;  
107 this.reverseMap = reverseMap;  
108 }  
109  
110 /\*\*  
111 \* Constructs a map that decorates the specified maps,  
112 \* used by the subclass <code>createBidiMap</code> implementation.  
113 \*  
114 \* @param normalMap the normal direction map  
115 \* @param reverseMap the reverse direction map  
116 \* @param inverseBidiMap the inverse BidiMap  
117 \*/  
118 protected AbstractDualBidiMap(final Map<K, V> normalMap, final Map<V, K> reverseMap,  
119 final BidiMap<V, K> inverseBidiMap) {  
120 super();  
121 this.normalMap = normalMap;  
122 this.reverseMap = reverseMap;  
123 this.inverseBidiMap = inverseBidiMap;  
124 }  
125  
126 /\*\*  
127 \* Creates a new instance of the subclass.  
128 \*  
129 \* @param normalMap the normal direction map  
130 \* @param reverseMap the reverse direction map  
131 \* @param inverseMap this map, which is the inverse in the new map  
132 \* @return the inverse map  
133 \*/  
134 protected abstract BidiMap<V, K> createBidiMap(Map<V, K> normalMap, Map<K, V> reverseMap, BidiMap<K, V> inverseMap);  
135  
136 // Map delegation  
137 //-----------------------------------------------------------------------  
138  
139 @Override  
140 public V get(final Object key) {  
141 return normalMap.get(key);  
142 }  
143  
144 @Override  
145 public int size() {  
146 return normalMap.size();  
147 }  
148  
149 @Override  
150 public boolean isEmpty() {  
151 return normalMap.isEmpty();  
152 }  
153  
154 @Override  
155 public boolean containsKey(final Object key) {  
156 return normalMap.containsKey(key);  
157 }  
158  
159 @Override  
160 public boolean equals(final Object obj) {  
161 return normalMap.equals(obj);  
162 }  
163  
164 @Override  
165 public int hashCode() {  
166 return normalMap.hashCode();  
167 }  
168  
169 @Override  
170 public String toString() {  
171 return normalMap.toString();  
172 }  
173  
174 // BidiMap changes  
175 //-----------------------------------------------------------------------  
176  
177 @Override  
178 public V put(final K key, final V value) {  
179 if (normalMap.containsKey(key)) {  
180 reverseMap.remove(normalMap.get(key));  
181 }  
182 if (reverseMap.containsKey(value)) {  
183 normalMap.remove(reverseMap.get(value));  
184 }  
185 final V obj = normalMap.put(key, value);  
186 reverseMap.put(value, key);  
187 return obj;  
188 }  
189  
190 @Override  
191 public void putAll(final Map<? extends K, ? extends V> map) {  
192 for (final Map.Entry<? extends K, ? extends V> entry : map.entrySet()) {  
193 put(entry.getKey(), entry.getValue());  
194 }  
195 }  
196  
197 @Override  
198 public V remove(final Object key) {  
199 V value = null;  
200 if (normalMap.containsKey(key)) {  
201 value = normalMap.remove(key);  
202 reverseMap.remove(value);  
203 }  
204 return value;  
205 }  
206  
207 @Override  
208 public void clear() {  
209 normalMap.clear();  
210 reverseMap.clear();  
211 }  
212  
213 @Override  
214 public boolean containsValue(final Object value) {  
215 return reverseMap.containsKey(value);  
216 }  
217  
218 // BidiMap  
219 //-----------------------------------------------------------------------  
220 /\*\*  
221 \* Obtains a <code>MapIterator</code> over the map.  
222 \* The iterator implements <code>ResetableMapIterator</code>.  
223 \* This implementation relies on the entrySet iterator.  
224 \* <p>  
225 \* The setValue() methods only allow a new value to be set.  
226 \* If the value being set is already in the map, an IllegalArgumentException  
227 \* is thrown (as setValue cannot change the size of the map).  
228 \*  
229 \* @return a map iterator  
230 \*/  
231 @Override  
232 public MapIterator<K, V> mapIterator() {  
233 return new BidiMapIterator<>(this);  
234 }  
235  
236 @Override  
237 public K getKey(final Object value) {  
238 return reverseMap.get(value);  
239 }  
240  
241 @Override  
242 public K removeValue(final Object value) {  
243 K key = null;  
244 if (reverseMap.containsKey(value)) {  
245 key = reverseMap.remove(value);  
246 normalMap.remove(key);  
247 }  
248 return key;  
249 }  
250  
251 @Override  
252 public BidiMap<V, K> inverseBidiMap() {  
253 if (inverseBidiMap == null) {  
254 inverseBidiMap = createBidiMap(reverseMap, normalMap, this);  
255 }  
256 return inverseBidiMap;  
257 }  
258  
259 // Map views  
260 //-----------------------------------------------------------------------  
261 /\*\*  
262 \* Gets a keySet view of the map.  
263 \* Changes made on the view are reflected in the map.  
264 \* The set supports remove and clear but not add.  
265 \*  
266 \* @return the keySet view  
267 \*/  
268 @Override  
269 public Set<K> keySet() {  
270 if (keySet == null) {  
271 keySet = new KeySet<>(this);  
272 }  
273 return keySet;  
274 }  
275  
276 /\*\*  
277 \* Creates a key set iterator.  
278 \* Subclasses can override this to return iterators with different properties.  
279 \*  
280 \* @param iterator the iterator to decorate  
281 \* @return the keySet iterator  
282 \*/  
283 protected Iterator<K> createKeySetIterator(final Iterator<K> iterator) {  
284 return new KeySetIterator<>(iterator, this);  
285 }  
286  
287 /\*\*  
288 \* Gets a values view of the map.  
289 \* Changes made on the view are reflected in the map.  
290 \* The set supports remove and clear but not add.  
291 \*  
292 \* @return the values view  
293 \*/  
294 @Override  
295 public Set<V> values() {  
296 if (values == null) {  
297 values = new Values<>(this);  
298 }  
299 return values;  
300 }  
301  
302 /\*\*  
303 \* Creates a values iterator.  
304 \* Subclasses can override this to return iterators with different properties.  
305 \*  
306 \* @param iterator the iterator to decorate  
307 \* @return the values iterator  
308 \*/  
309 protected Iterator<V> createValuesIterator(final Iterator<V> iterator) {  
310 return new ValuesIterator<>(iterator, this);  
311 }  
312  
313 /\*\*  
314 \* Gets an entrySet view of the map.  
315 \* Changes made on the set are reflected in the map.  
316 \* The set supports remove and clear but not add.  
317 \* <p>  
318 \* The Map Entry setValue() method only allow a new value to be set.  
319 \* If the value being set is already in the map, an IllegalArgumentException  
320 \* is thrown (as setValue cannot change the size of the map).  
321 \*  
322 \* @return the entrySet view  
323 \*/  
324 @Override  
325 public Set<Map.Entry<K, V>> entrySet() {  
326 if (entrySet == null) {  
327 entrySet = new EntrySet<>(this);  
328 }  
329 return entrySet;  
330 }  
331  
332 /\*\*  
333 \* Creates an entry set iterator.  
334 \* Subclasses can override this to return iterators with different properties.  
335 \*  
336 \* @param iterator the iterator to decorate  
337 \* @return the entrySet iterator  
338 \*/  
339 protected Iterator<Map.Entry<K, V>> createEntrySetIterator(final Iterator<Map.Entry<K, V>> iterator) {  
340 return new EntrySetIterator<>(iterator, this);  
341 }  
342  
343 //-----------------------------------------------------------------------  
344 /\*\*  
345 \* Inner class View.  
346 \*/  
347 protected static abstract class View<K, V, E> extends AbstractCollectionDecorator<E> {  
348  
349 /\*\* Generated serial version ID. \*/  
350 private static final long serialVersionUID = 4621510560119690639L;  
351  
352 /\*\* The parent map \*/  
353 protected final AbstractDualBidiMap<K, V> parent;  
354  
355 /\*\*  
356 \* Constructs a new view of the BidiMap.  
357 \*  
358 \* @param coll the collection view being decorated  
359 \* @param parent the parent BidiMap  
360 \*/  
361 protected View(final Collection<E> coll, final AbstractDualBidiMap<K, V> parent) {  
362 super(coll);  
363 this.parent = parent;  
364 }  
365  
366 @Override  
367 public boolean equals(final Object object) {  
368 return object == this || decorated().equals(object);  
369 }  
370  
371 @Override  
372 public int hashCode() {  
373 return decorated().hashCode();  
374 }  
375  
376 /\*\*  
377 \* @since 4.4  
378 \*/  
379 @Override  
380 public boolean removeIf(Predicate<? super E> filter) {  
381 if (parent.isEmpty() || Objects.isNull(filter)) {  
382 return false;  
383 }  
384 boolean modified = false;  
385 final Iterator<?> it = iterator();  
386 while (it.hasNext()) {  
387 @SuppressWarnings("unchecked")  
388 final E e = (E) it.next();  
389 if (filter.test(e)) {  
390 it.remove();  
391 modified = true;  
392 }  
393 }  
394 return modified;  
395 }  
396  
397 @Override  
398 public boolean removeAll(final Collection<?> coll) {  
399 if (parent.isEmpty() || coll.isEmpty()) {  
400 return false;  
401 }  
402 boolean modified = false;  
403 final Iterator<?> it = coll.iterator();  
404 while (it.hasNext()) {  
405 modified |= remove(it.next());  
406 }  
407 return modified;  
408 }  
409  
410 /\*\*  
411 \* {@inheritDoc}  
412 \* <p>  
413 \* This implementation iterates over the elements of this bidi map, checking each element in  
414 \* turn to see if it's contained in <code>coll</code>. If it's not contained, it's removed  
415 \* from this bidi map. As a consequence, it is advised to use a collection type for  
416 \* <code>coll</code> that provides a fast (e.g. O(1)) implementation of  
417 \* {@link Collection#contains(Object)}.  
418 \*/  
419 @Override  
420 public boolean retainAll(final Collection<?> coll) {  
421 if (parent.isEmpty()) {  
422 return false;  
423 }  
424 if (coll.isEmpty()) {  
425 parent.clear();  
426 return true;  
427 }  
428 boolean modified = false;  
429 final Iterator<E> it = iterator();  
430 while (it.hasNext()) {  
431 if (coll.contains(it.next()) == false) {  
432 it.remove();  
433 modified = true;  
434 }  
435 }  
436 return modified;  
437 }  
438  
439 @Override  
440 public void clear() {  
441 parent.clear();  
442 }  
443 }  
444  
445 //-----------------------------------------------------------------------  
446 /\*\*  
447 \* Inner class KeySet.  
448 \*/  
449 protected static class KeySet<K> extends View<K, Object, K> implements Set<K> {  
450  
451 /\*\* Serialization version \*/  
452 private static final long serialVersionUID = -7107935777385040694L;  
453  
454 /\*\*  
455 \* Constructs a new view of the BidiMap.  
456 \*  
457 \* @param parent the parent BidiMap  
458 \*/  
459 @SuppressWarnings("unchecked")  
460 protected KeySet(final AbstractDualBidiMap<K, ?> parent) {  
461 super(parent.normalMap.keySet(), (AbstractDualBidiMap<K, Object>) parent);  
462 }  
463  
464 @Override  
465 public Iterator<K> iterator() {  
466 return parent.createKeySetIterator(super.iterator());  
467 }  
468  
469 @Override  
470 public boolean contains(final Object key) {  
471 return parent.normalMap.containsKey(key);  
472 }  
473  
474 @Override  
475 public boolean remove(final Object key) {  
476 if (parent.normalMap.containsKey(key)) {  
477 final Object value = parent.normalMap.remove(key);  
478 parent.reverseMap.remove(value);  
479 return true;  
480 }  
481 return false;  
482 }  
483 }  
484  
485 /\*\*  
486 \* Inner class KeySetIterator.  
487 \*/  
488 protected static class KeySetIterator<K> extends AbstractIteratorDecorator<K> {  
489  
490 /\*\* The parent map \*/  
491 protected final AbstractDualBidiMap<K, ?> parent;  
492  
493 /\*\* The last returned key \*/  
494 protected K lastKey = null;  
495  
496 /\*\* Whether remove is allowed at present \*/  
497 protected boolean canRemove = false;  
498  
499 /\*\*  
500 \* Constructor.  
501 \* @param iterator the iterator to decorate  
502 \* @param parent the parent map  
503 \*/  
504 protected KeySetIterator(final Iterator<K> iterator, final AbstractDualBidiMap<K, ?> parent) {  
505 super(iterator);  
506 this.parent = parent;  
507 }  
508  
509 @Override  
510 public K next() {  
511 lastKey = super.next();  
512 canRemove = true;  
513 return lastKey;  
514 }  
515  
516 @Override  
517 public void remove() {  
518 if (canRemove == false) {  
519 throw new IllegalStateException("Iterator remove() can only be called once after next()");  
520 }  
521 final Object value = parent.normalMap.get(lastKey);  
522 super.remove();  
523 parent.reverseMap.remove(value);  
524 lastKey = null;  
525 canRemove = false;  
526 }  
527 }  
528  
529 //-----------------------------------------------------------------------  
530 /\*\*  
531 \* Inner class Values.  
532 \*/  
533 protected static class Values<V> extends View<Object, V, V> implements Set<V> {  
534  
535 /\*\* Serialization version \*/  
536 private static final long serialVersionUID = 4023777119829639864L;  
537  
538 /\*\*  
539 \* Constructs a new view of the BidiMap.  
540 \*  
541 \* @param parent the parent BidiMap  
542 \*/  
543 @SuppressWarnings("unchecked")  
544 protected Values(final AbstractDualBidiMap<?, V> parent) {  
545 super(parent.normalMap.values(), (AbstractDualBidiMap<Object, V>) parent);  
546 }  
547  
548 @Override  
549 public Iterator<V> iterator() {  
550 return parent.createValuesIterator(super.iterator());  
551 }  
552  
553 @Override  
554 public boolean contains(final Object value) {  
555 return parent.reverseMap.containsKey(value);  
556 }  
557  
558 @Override  
559 public boolean remove(final Object value) {  
560 if (parent.reverseMap.containsKey(value)) {  
561 final Object key = parent.reverseMap.remove(value);  
562 parent.normalMap.remove(key);  
563 return true;  
564 }  
565 return false;  
566 }  
567 }  
568  
569 /\*\*  
570 \* Inner class ValuesIterator.  
571 \*/  
572 protected static class ValuesIterator<V> extends AbstractIteratorDecorator<V> {  
573  
574 /\*\* The parent map \*/  
575 protected final AbstractDualBidiMap<Object, V> parent;  
576  
577 /\*\* The last returned value \*/  
578 protected V lastValue = null;  
579  
580 /\*\* Whether remove is allowed at present \*/  
581 protected boolean canRemove = false;  
582  
583 /\*\*  
584 \* Constructor.  
585 \* @param iterator the iterator to decorate  
586 \* @param parent the parent map  
587 \*/  
588 @SuppressWarnings("unchecked")  
589 protected ValuesIterator(final Iterator<V> iterator, final AbstractDualBidiMap<?, V> parent) {  
590 super(iterator);  
591 this.parent = (AbstractDualBidiMap<Object, V>) parent;  
592 }  
593  
594 @Override  
595 public V next() {  
596 lastValue = super.next();  
597 canRemove = true;  
598 return lastValue;  
599 }  
600  
601 @Override  
602 public void remove() {  
603 if (canRemove == false) {  
604 throw new IllegalStateException("Iterator remove() can only be called once after next()");  
605 }  
606 super.remove(); // removes from maps[0]  
607 parent.reverseMap.remove(lastValue);  
608 lastValue = null;  
609 canRemove = false;  
610 }  
611 }  
612  
613 //-----------------------------------------------------------------------  
614 /\*\*  
615 \* Inner class EntrySet.  
616 \*/  
617 protected static class EntrySet<K, V> extends View<K, V, Map.Entry<K, V>> implements Set<Map.Entry<K, V>> {  
618  
619 /\*\* Serialization version \*/  
620 private static final long serialVersionUID = 4040410962603292348L;  
621  
622 /\*\*  
623 \* Constructs a new view of the BidiMap.  
624 \*  
625 \* @param parent the parent BidiMap  
626 \*/  
627 protected EntrySet(final AbstractDualBidiMap<K, V> parent) {  
628 super(parent.normalMap.entrySet(), parent);  
629 }  
630  
631 @Override  
632 public Iterator<Map.Entry<K, V>> iterator() {  
633 return parent.createEntrySetIterator(super.iterator());  
634 }  
635  
636 @Override  
637 public boolean remove(final Object obj) {  
638 if (obj instanceof Map.Entry == false) {  
639 return false;  
640 }  
641 final Map.Entry<?, ?> entry = (Map.Entry<?, ?>) obj;  
642 final Object key = entry.getKey();  
643 if (parent.containsKey(key)) {  
644 final V value = parent.normalMap.get(key);  
645 if (value == null ? entry.getValue() == null : value.equals(entry.getValue())) {  
646 parent.normalMap.remove(key);  
647 parent.reverseMap.remove(value);  
648 return true;  
649 }  
650 }  
651 return false;  
652 }  
653 }  
654  
655 /\*\*  
656 \* Inner class EntrySetIterator.  
657 \*/  
658 protected static class EntrySetIterator<K, V> extends AbstractIteratorDecorator<Map.Entry<K, V>> {  
659  
660 /\*\* The parent map \*/  
661 protected final AbstractDualBidiMap<K, V> parent;  
662  
663 /\*\* The last returned entry \*/  
664 protected Map.Entry<K, V> last = null;  
665  
666 /\*\* Whether remove is allowed at present \*/  
667 protected boolean canRemove = false;  
668  
669 /\*\*  
670 \* Constructor.  
671 \* @param iterator the iterator to decorate  
672 \* @param parent the parent map  
673 \*/  
674 protected EntrySetIterator(final Iterator<Map.Entry<K, V>> iterator, final AbstractDualBidiMap<K, V> parent) {  
675 super(iterator);  
676 this.parent = parent;  
677 }  
678  
679 @Override  
680 public Map.Entry<K, V> next() {  
681 last = new MapEntry<>(super.next(), parent);  
682 canRemove = true;  
683 return last;  
684 }  
685  
686 @Override  
687 public void remove() {  
688 if (canRemove == false) {  
689 throw new IllegalStateException("Iterator remove() can only be called once after next()");  
690 }  
691 // store value as remove may change the entry in the decorator (eg.TreeMap)  
692 final Object value = last.getValue();  
693 super.remove();  
694 parent.reverseMap.remove(value);  
695 last = null;  
696 canRemove = false;  
697 }  
698 }  
699  
700 /\*\*  
701 \* Inner class MapEntry.  
702 \*/  
703 protected static class MapEntry<K, V> extends AbstractMapEntryDecorator<K, V> {  
704  
705 /\*\* The parent map \*/  
706 protected final AbstractDualBidiMap<K, V> parent;  
707  
708 /\*\*  
709 \* Constructor.  
710 \* @param entry the entry to decorate  
711 \* @param parent the parent map  
712 \*/  
713 protected MapEntry(final Map.Entry<K, V> entry, final AbstractDualBidiMap<K, V> parent) {  
714 super(entry);  
715 this.parent = parent;  
716 }  
717  
718 @Override  
719 public V setValue(final V value) {  
720 final K key = MapEntry.this.getKey();  
721 if (parent.reverseMap.containsKey(value) &&  
722 parent.reverseMap.get(value) != key) {  
723 throw new IllegalArgumentException(  
724 "Cannot use setValue() when the object being set is already in the map");  
725 }  
726 parent.put(key, value);  
727 return super.setValue(value);  
728 }  
729 }  
730  
731 /\*\*  
732 \* Inner class MapIterator.  
733 \*/  
734 protected static class BidiMapIterator<K, V> implements MapIterator<K, V>, ResettableIterator<K> {  
735  
736 /\*\* The parent map \*/  
737 protected final AbstractDualBidiMap<K, V> parent;  
738  
739 /\*\* The iterator being wrapped \*/  
740 protected Iterator<Map.Entry<K, V>> iterator;  
741  
742 /\*\* The last returned entry \*/  
743 protected Map.Entry<K, V> last = null;  
744  
745 /\*\* Whether remove is allowed at present \*/  
746 protected boolean canRemove = false;  
747  
748 /\*\*  
749 \* Constructor.  
750 \* @param parent the parent map  
751 \*/  
752 protected BidiMapIterator(final AbstractDualBidiMap<K, V> parent) {  
753 super();  
754 this.parent = parent;  
755 this.iterator = parent.normalMap.entrySet().iterator();  
756 }  
757  
758 @Override  
759 public boolean hasNext() {  
760 return iterator.hasNext();  
761 }  
762  
763 @Override  
764 public K next() {  
765 last = iterator.next();  
766 canRemove = true;  
767 return last.getKey();  
768 }  
769  
770 @Override  
771 public void remove() {  
772 if (canRemove == false) {  
773 throw new IllegalStateException("Iterator remove() can only be called once after next()");  
774 }  
775 // store value as remove may change the entry in the decorator (eg.TreeMap)  
776 final V value = last.getValue();  
777 iterator.remove();  
778 parent.reverseMap.remove(value);  
779 last = null;  
780 canRemove = false;  
781 }  
782  
783 @Override  
784 public K getKey() {  
785 if (last == null) {  
786 throw new IllegalStateException(  
787 "Iterator getKey() can only be called after next() and before remove()");  
788 }  
789 return last.getKey();  
790 }  
791  
792 @Override  
793 public V getValue() {  
794 if (last == null) {  
795 throw new IllegalStateException(  
796 "Iterator getValue() can only be called after next() and before remove()");  
797 }  
798 return last.getValue();  
799 }  
800  
801 @Override  
802 public V setValue(final V value) {  
803 if (last == null) {  
804 throw new IllegalStateException(  
805 "Iterator setValue() can only be called after next() and before remove()");  
806 }  
807 if (parent.reverseMap.containsKey(value) &&  
808 parent.reverseMap.get(value) != last.getKey()) {  
809 throw new IllegalArgumentException(  
810 "Cannot use setValue() when the object being set is already in the map");  
811 }  
812 return parent.put(last.getKey(), value);  
813 }  
814  
815 @Override  
816 public void reset() {  
817 iterator = parent.normalMap.entrySet().iterator();  
818 last = null;  
819 canRemove = false;  
820 }  
821  
822 @Override  
823 public String toString() {  
824 if (last != null) {  
825 return "MapIterator[" + getKey() + "=" + getValue() + "]";  
826 }  
827 return "MapIterator[]";  
828 }  
829 }  
830  
831}