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.lang.ref.Reference;  
023import java.lang.ref.ReferenceQueue;  
024import java.lang.ref.SoftReference;  
025import java.lang.ref.WeakReference;  
026import java.util.ArrayList;  
027import java.util.Collection;  
028import java.util.ConcurrentModificationException;  
029import java.util.Iterator;  
030import java.util.List;  
031import java.util.Map;  
032import java.util.NoSuchElementException;  
033import java.util.Set;  
034  
035import org.apache.commons.collections4.MapIterator;  
036import org.apache.commons.collections4.keyvalue.DefaultMapEntry;  
037  
038/\*\*  
039 \* An abstract implementation of a hash-based map that allows the entries to  
040 \* be removed by the garbage collector.  
041 \* <p>  
042 \* This class implements all the features necessary for a subclass reference  
043 \* hash-based map. Key-value entries are stored in instances of the  
044 \* <code>ReferenceEntry</code> class which can be overridden and replaced.  
045 \* The iterators can similarly be replaced, without the need to replace the KeySet,  
046 \* EntrySet and Values view classes.  
047 \* </p>  
048 \* <p>  
049 \* Overridable methods are provided to change the default hashing behaviour, and  
050 \* to change how entries are added to and removed from the map. Hopefully, all you  
051 \* need for unusual subclasses is here.  
052 \* </p>  
053 \* <p>  
054 \* When you construct an <code>AbstractReferenceMap</code>, you can specify what  
055 \* kind of references are used to store the map's keys and values.  
056 \* If non-hard references are used, then the garbage collector can remove  
057 \* mappings if a key or value becomes unreachable, or if the JVM's memory is  
058 \* running low. For information on how the different reference types behave,  
059 \* see {@link Reference}.  
060 \* </p>  
061 \* <p>  
062 \* Different types of references can be specified for keys and values.  
063 \* The keys can be configured to be weak but the values hard,  
064 \* in which case this class will behave like a  
065 \* <a href="http://java.sun.com/j2se/1.4/docs/api/java/util/WeakHashMap.html">  
066 \* <code>WeakHashMap</code></a>. However, you can also specify hard keys and  
067 \* weak values, or any other combination. The default constructor uses  
068 \* hard keys and soft values, providing a memory-sensitive cache.  
069 \* </p>  
070 \* <p>  
071 \* This {@link Map} implementation does <i>not</i> allow null elements.  
072 \* Attempting to add a null key or value to the map will raise a  
073 \* <code>NullPointerException</code>.  
074 \* </p>  
075 \* <p>  
076 \* All the available iterators can be reset back to the start by casting to  
077 \* <code>ResettableIterator</code> and calling <code>reset()</code>.  
078 \* </p>  
079 \* <p>  
080 \* This implementation is not synchronized.  
081 \* You can use {@link java.util.Collections#synchronizedMap} to  
082 \* provide synchronized access to a <code>ReferenceMap</code>.  
083 \* </p>  
084 \*  
085 \* @param <K> the type of the keys in this map  
086 \* @param <V> the type of the values in this map  
087 \*  
088 \* @see java.lang.ref.Reference  
089 \* @since 3.1 (extracted from ReferenceMap in 3.0)  
090 \*/  
091public abstract class AbstractReferenceMap<K, V> extends AbstractHashedMap<K, V> {  
092  
093 /\*\*  
094 \* Reference type enum.  
095 \*/  
096 public enum ReferenceStrength {  
097 HARD(0), SOFT(1), WEAK(2);  
098  
099 /\*\* value \*/  
100 public final int value;  
101  
102 /\*\*  
103 \* Resolve enum from int.  
104 \* @param value the int value  
105 \* @return ReferenceType  
106 \* @throws IllegalArgumentException if the specified value is invalid.  
107 \*/  
108 public static ReferenceStrength resolve(final int value) {  
109 switch (value) {  
110 case 0:  
111 return HARD;  
112 case 1:  
113 return SOFT;  
114 case 2:  
115 return WEAK;  
116 default:  
117 throw new IllegalArgumentException();  
118 }  
119 }  
120  
121 ReferenceStrength(final int value) {  
122 this.value = value;  
123 }  
124  
125 }  
126  
127 /\*\*  
128 \* The reference type for keys.  
129 \*/  
130 private ReferenceStrength keyType;  
131  
132 /\*\*  
133 \* The reference type for values.  
134 \*/  
135 private ReferenceStrength valueType;  
136  
137 /\*\*  
138 \* Should the value be automatically purged when the associated key has been collected?  
139 \*/  
140 private boolean purgeValues;  
141  
142 /\*\*  
143 \* ReferenceQueue used to eliminate stale mappings.  
144 \* See purge.  
145 \*/  
146 private transient ReferenceQueue<Object> queue;  
147  
148 //-----------------------------------------------------------------------  
149 /\*\*  
150 \* Constructor used during deserialization.  
151 \*/  
152 protected AbstractReferenceMap() {  
153 super();  
154 }  
155  
156 /\*\*  
157 \* Constructs a new empty map with the specified reference types,  
158 \* load factor and initial capacity.  
159 \*  
160 \* @param keyType the type of reference to use for keys;  
161 \* must be {@link ReferenceStrength#HARD HARD},  
162 \* {@link ReferenceStrength#SOFT SOFT},  
163 \* {@link ReferenceStrength#WEAK WEAK}  
164 \* @param valueType the type of reference to use for values;  
165 \* must be {@link ReferenceStrength#HARD},  
166 \* {@link ReferenceStrength#SOFT SOFT},  
167 \* {@link ReferenceStrength#WEAK WEAK}  
168 \* @param capacity the initial capacity for the map  
169 \* @param loadFactor the load factor for the map  
170 \* @param purgeValues should the value be automatically purged when the  
171 \* key is garbage collected  
172 \*/  
173 protected AbstractReferenceMap(  
174 final ReferenceStrength keyType, final ReferenceStrength valueType, final int capacity,  
175 final float loadFactor, final boolean purgeValues) {  
176 super(capacity, loadFactor);  
177 this.keyType = keyType;  
178 this.valueType = valueType;  
179 this.purgeValues = purgeValues;  
180 }  
181  
182 /\*\*  
183 \* Initialise this subclass during construction, cloning or deserialization.  
184 \*/  
185 @Override  
186 protected void init() {  
187 queue = new ReferenceQueue<>();  
188 }  
189  
190 //-----------------------------------------------------------------------  
191 /\*\*  
192 \* Gets the size of the map.  
193 \*  
194 \* @return the size  
195 \*/  
196 @Override  
197 public int size() {  
198 purgeBeforeRead();  
199 return super.size();  
200 }  
201  
202 /\*\*  
203 \* Checks whether the map is currently empty.  
204 \*  
205 \* @return true if the map is currently size zero  
206 \*/  
207 @Override  
208 public boolean isEmpty() {  
209 purgeBeforeRead();  
210 return super.isEmpty();  
211 }  
212  
213 /\*\*  
214 \* Checks whether the map contains the specified key.  
215 \*  
216 \* @param key the key to search for  
217 \* @return true if the map contains the key  
218 \*/  
219 @Override  
220 public boolean containsKey(final Object key) {  
221 purgeBeforeRead();  
222 final Entry<K, V> entry = getEntry(key);  
223 if (entry == null) {  
224 return false;  
225 }  
226 return entry.getValue() != null;  
227 }  
228  
229 /\*\*  
230 \* Checks whether the map contains the specified value.  
231 \*  
232 \* @param value the value to search for  
233 \* @return true if the map contains the value  
234 \*/  
235 @Override  
236 public boolean containsValue(final Object value) {  
237 purgeBeforeRead();  
238 if (value == null) {  
239 return false;  
240 }  
241 return super.containsValue(value);  
242 }  
243  
244 /\*\*  
245 \* Gets the value mapped to the key specified.  
246 \*  
247 \* @param key the key  
248 \* @return the mapped value, null if no match  
249 \*/  
250 @Override  
251 public V get(final Object key) {  
252 purgeBeforeRead();  
253 final Entry<K, V> entry = getEntry(key);  
254 if (entry == null) {  
255 return null;  
256 }  
257 return entry.getValue();  
258 }  
259  
260  
261 /\*\*  
262 \* Puts a key-value mapping into this map.  
263 \* Neither the key nor the value may be null.  
264 \*  
265 \* @param key the key to add, must not be null  
266 \* @param value the value to add, must not be null  
267 \* @return the value previously mapped to this key, null if none  
268 \* @throws NullPointerException if either the key or value is null  
269 \*/  
270 @Override  
271 public V put(final K key, final V value) {  
272 if (key == null) {  
273 throw new NullPointerException("null keys not allowed");  
274 }  
275 if (value == null) {  
276 throw new NullPointerException("null values not allowed");  
277 }  
278  
279 purgeBeforeWrite();  
280 return super.put(key, value);  
281 }  
282  
283 /\*\*  
284 \* Removes the specified mapping from this map.  
285 \*  
286 \* @param key the mapping to remove  
287 \* @return the value mapped to the removed key, null if key not in map  
288 \*/  
289 @Override  
290 public V remove(final Object key) {  
291 if (key == null) {  
292 return null;  
293 }  
294 purgeBeforeWrite();  
295 return super.remove(key);  
296 }  
297  
298 /\*\*  
299 \* Clears this map.  
300 \*/  
301 @Override  
302 public void clear() {  
303 super.clear();  
304 // drain the queue  
305 while (queue.poll() != null) {  
306 // empty  
307 }  
308 }  
309  
310 //-----------------------------------------------------------------------  
311 /\*\*  
312 \* Gets a MapIterator over the reference map.  
313 \* The iterator only returns valid key/value pairs.  
314 \*  
315 \* @return a map iterator  
316 \*/  
317 @Override  
318 public MapIterator<K, V> mapIterator() {  
319 return new ReferenceMapIterator<>(this);  
320 }  
321  
322 /\*\*  
323 \* Returns a set view of this map's entries.  
324 \* An iterator returned entry is valid until <code>next()</code> is called again.  
325 \* The <code>setValue()</code> method on the <code>toArray</code> entries has no effect.  
326 \*  
327 \* @return a set view of this map's entries  
328 \*/  
329 @Override  
330 public Set<Map.Entry<K, V>> entrySet() {  
331 if (entrySet == null) {  
332 entrySet = new ReferenceEntrySet<>(this);  
333 }  
334 return entrySet;  
335 }  
336  
337 /\*\*  
338 \* Returns a set view of this map's keys.  
339 \*  
340 \* @return a set view of this map's keys  
341 \*/  
342 @Override  
343 public Set<K> keySet() {  
344 if (keySet == null) {  
345 keySet = new ReferenceKeySet<>(this);  
346 }  
347 return keySet;  
348 }  
349  
350 /\*\*  
351 \* Returns a collection view of this map's values.  
352 \*  
353 \* @return a set view of this map's values  
354 \*/  
355 @Override  
356 public Collection<V> values() {  
357 if (values == null) {  
358 values = new ReferenceValues<>(this);  
359 }  
360 return values;  
361 }  
362  
363 //-----------------------------------------------------------------------  
364 /\*\*  
365 \* Purges stale mappings from this map before read operations.  
366 \* <p>  
367 \* This implementation calls {@link #purge()} to maintain a consistent state.  
368 \*/  
369 protected void purgeBeforeRead() {  
370 purge();  
371 }  
372  
373 /\*\*  
374 \* Purges stale mappings from this map before write operations.  
375 \* <p>  
376 \* This implementation calls {@link #purge()} to maintain a consistent state.  
377 \*/  
378 protected void purgeBeforeWrite() {  
379 purge();  
380 }  
381  
382 /\*\*  
383 \* Purges stale mappings from this map.  
384 \* <p>  
385 \* Note that this method is not synchronized! Special  
386 \* care must be taken if, for instance, you want stale  
387 \* mappings to be removed on a periodic basis by some  
388 \* background thread.  
389 \*/  
390 protected void purge() {  
391 Reference<?> ref = queue.poll();  
392 while (ref != null) {  
393 purge(ref);  
394 ref = queue.poll();  
395 }  
396 }  
397  
398 /\*\*  
399 \* Purges the specified reference.  
400 \*  
401 \* @param ref the reference to purge  
402 \*/  
403 protected void purge(final Reference<?> ref) {  
404 // The hashCode of the reference is the hashCode of the  
405 // mapping key, even if the reference refers to the  
406 // mapping value...  
407 final int hash = ref.hashCode();  
408 final int index = hashIndex(hash, data.length);  
409 HashEntry<K, V> previous = null;  
410 HashEntry<K, V> entry = data[index];  
411 while (entry != null) {  
412 ReferenceEntry<K, V> refEntry = (ReferenceEntry<K, V>) entry;  
413 if (refEntry.purge(ref)) {  
414 if (previous == null) {  
415 data[index] = entry.next;  
416 } else {  
417 previous.next = entry.next;  
418 }  
419 this.size--;  
420 refEntry.onPurge();  
421 return;  
422 }  
423 previous = entry;  
424 entry = entry.next;  
425 }  
426  
427 }  
428  
429 //-----------------------------------------------------------------------  
430 /\*\*  
431 \* Gets the entry mapped to the key specified.  
432 \*  
433 \* @param key the key  
434 \* @return the entry, null if no match  
435 \*/  
436 @Override  
437 protected HashEntry<K, V> getEntry(final Object key) {  
438 if (key == null) {  
439 return null;  
440 }  
441 return super.getEntry(key);  
442 }  
443  
444 /\*\*  
445 \* Gets the hash code for a MapEntry.  
446 \* Subclasses can override this, for example to use the identityHashCode.  
447 \*  
448 \* @param key the key to get a hash code for, may be null  
449 \* @param value the value to get a hash code for, may be null  
450 \* @return the hash code, as per the MapEntry specification  
451 \*/  
452 protected int hashEntry(final Object key, final Object value) {  
453 return (key == null ? 0 : key.hashCode()) ^  
454 (value == null ? 0 : value.hashCode());  
455 }  
456  
457 /\*\*  
458 \* Compares two keys, in internal converted form, to see if they are equal.  
459 \* <p>  
460 \* This implementation converts the key from the entry to a real reference  
461 \* before comparison.  
462 \*  
463 \* @param key1 the first key to compare passed in from outside  
464 \* @param key2 the second key extracted from the entry via <code>entry.key</code>  
465 \* @return true if equal  
466 \*/  
467 @Override  
468 @SuppressWarnings("unchecked")  
469 protected boolean isEqualKey(final Object key1, Object key2) {  
470 key2 = keyType == ReferenceStrength.HARD ? key2 : ((Reference<K>) key2).get();  
471 return key1 == key2 || key1.equals(key2);  
472 }  
473  
474 /\*\*  
475 \* Creates a ReferenceEntry instead of a HashEntry.  
476 \*  
477 \* @param next the next entry in sequence  
478 \* @param hashCode the hash code to use  
479 \* @param key the key to store  
480 \* @param value the value to store  
481 \* @return the newly created entry  
482 \*/  
483 @Override  
484 protected ReferenceEntry<K, V> createEntry(final HashEntry<K, V> next, final int hashCode,  
485 final K key, final V value) {  
486 return new ReferenceEntry<>(this, next, hashCode, key, value);  
487 }  
488  
489 /\*\*  
490 \* Creates an entry set iterator.  
491 \*  
492 \* @return the entrySet iterator  
493 \*/  
494 @Override  
495 protected Iterator<Map.Entry<K, V>> createEntrySetIterator() {  
496 return new ReferenceEntrySetIterator<>(this);  
497 }  
498  
499 /\*\*  
500 \* Creates an key set iterator.  
501 \*  
502 \* @return the keySet iterator  
503 \*/  
504 @Override  
505 protected Iterator<K> createKeySetIterator() {  
506 return new ReferenceKeySetIterator<>(this);  
507 }  
508  
509 /\*\*  
510 \* Creates an values iterator.  
511 \*  
512 \* @return the values iterator  
513 \*/  
514 @Override  
515 protected Iterator<V> createValuesIterator() {  
516 return new ReferenceValuesIterator<>(this);  
517 }  
518  
519 //-----------------------------------------------------------------------  
520 /\*\*  
521 \* EntrySet implementation.  
522 \*/  
523 static class ReferenceEntrySet<K, V> extends EntrySet<K, V> {  
524  
525 protected ReferenceEntrySet(final AbstractHashedMap<K, V> parent) {  
526 super(parent);  
527 }  
528  
529 @Override  
530 public Object[] toArray() {  
531 return toArray(new Object[size()]);  
532 }  
533  
534 @Override  
535 public <T> T[] toArray(final T[] arr) {  
536 // special implementation to handle disappearing entries  
537 final ArrayList<Map.Entry<K, V>> list = new ArrayList<>(size());  
538 for (final Map.Entry<K, V> entry : this) {  
539 list.add(new DefaultMapEntry<>(entry));  
540 }  
541 return list.toArray(arr);  
542 }  
543 }  
544  
545 //-----------------------------------------------------------------------  
546 /\*\*  
547 \* KeySet implementation.  
548 \*/  
549 static class ReferenceKeySet<K> extends KeySet<K> {  
550  
551 protected ReferenceKeySet(final AbstractHashedMap<K, ?> parent) {  
552 super(parent);  
553 }  
554  
555 @Override  
556 public Object[] toArray() {  
557 return toArray(new Object[size()]);  
558 }  
559  
560 @Override  
561 public <T> T[] toArray(final T[] arr) {  
562 // special implementation to handle disappearing keys  
563 final List<K> list = new ArrayList<>(size());  
564 for (final K key : this) {  
565 list.add(key);  
566 }  
567 return list.toArray(arr);  
568 }  
569 }  
570  
571 //-----------------------------------------------------------------------  
572 /\*\*  
573 \* Values implementation.  
574 \*/  
575 static class ReferenceValues<V> extends Values<V> {  
576  
577 protected ReferenceValues(final AbstractHashedMap<?, V> parent) {  
578 super(parent);  
579 }  
580  
581 @Override  
582 public Object[] toArray() {  
583 return toArray(new Object[size()]);  
584 }  
585  
586 @Override  
587 public <T> T[] toArray(final T[] arr) {  
588 // special implementation to handle disappearing values  
589 final List<V> list = new ArrayList<>(size());  
590 for (final V value : this) {  
591 list.add(value);  
592 }  
593 return list.toArray(arr);  
594 }  
595 }  
596  
597 //-----------------------------------------------------------------------  
598 /\*\*  
599 \* A MapEntry implementation for the map.  
600 \* <p>  
601 \* If getKey() or getValue() returns null, it means  
602 \* the mapping is stale and should be removed.  
603 \*  
604 \* @since 3.1  
605 \*/  
606 protected static class ReferenceEntry<K, V> extends HashEntry<K, V> {  
607 /\*\* The parent map \*/  
608 private final AbstractReferenceMap<K, V> parent;  
609  
610 /\*\*  
611 \* Creates a new entry object for the ReferenceMap.  
612 \*  
613 \* @param parent the parent map  
614 \* @param next the next entry in the hash bucket  
615 \* @param hashCode the hash code of the key  
616 \* @param key the key  
617 \* @param value the value  
618 \*/  
619 public ReferenceEntry(final AbstractReferenceMap<K, V> parent, final HashEntry<K, V> next,  
620 final int hashCode, final K key, final V value) {  
621 super(next, hashCode, null, null);  
622 this.parent = parent;  
623 this.key = toReference(parent.keyType, key, hashCode);  
624 this.value = toReference(parent.valueType, value, hashCode); // the key hashCode is passed in deliberately  
625 }  
626  
627 /\*\*  
628 \* Gets the key from the entry.  
629 \* This method dereferences weak and soft keys and thus may return null.  
630 \*  
631 \* @return the key, which may be null if it was garbage collected  
632 \*/  
633 @Override  
634 @SuppressWarnings("unchecked")  
635 public K getKey() {  
636 return (K) (parent.keyType == ReferenceStrength.HARD ? key : ((Reference<K>) key).get());  
637 }  
638  
639 /\*\*  
640 \* Gets the value from the entry.  
641 \* This method dereferences weak and soft value and thus may return null.  
642 \*  
643 \* @return the value, which may be null if it was garbage collected  
644 \*/  
645 @Override  
646 @SuppressWarnings("unchecked")  
647 public V getValue() {  
648 return (V) (parent.valueType == ReferenceStrength.HARD ? value : ((Reference<V>) value).get());  
649 }  
650  
651 /\*\*  
652 \* Sets the value of the entry.  
653 \*  
654 \* @param obj the object to store  
655 \* @return the previous value  
656 \*/  
657 @Override  
658 @SuppressWarnings("unchecked")  
659 public V setValue(final V obj) {  
660 final V old = getValue();  
661 if (parent.valueType != ReferenceStrength.HARD) {  
662 ((Reference<V>) value).clear();  
663 }  
664 value = toReference(parent.valueType, obj, hashCode);  
665 return old;  
666 }  
667  
668 /\*\*  
669 \* Compares this map entry to another.  
670 \* <p>  
671 \* This implementation uses <code>isEqualKey</code> and  
672 \* <code>isEqualValue</code> on the main map for comparison.  
673 \*  
674 \* @param obj the other map entry to compare to  
675 \* @return true if equal, false if not  
676 \*/  
677 @Override  
678 public boolean equals(final Object obj) {  
679 if (obj == this) {  
680 return true;  
681 }  
682 if (obj instanceof Map.Entry == false) {  
683 return false;  
684 }  
685  
686 final Map.Entry<?, ?> entry = (Map.Entry<?, ?>)obj;  
687 final Object entryKey = entry.getKey(); // convert to hard reference  
688 final Object entryValue = entry.getValue(); // convert to hard reference  
689 if (entryKey == null || entryValue == null) {  
690 return false;  
691 }  
692 // compare using map methods, aiding identity subclass  
693 // note that key is direct access and value is via method  
694 return parent.isEqualKey(entryKey, key) &&  
695 parent.isEqualValue(entryValue, getValue());  
696 }  
697  
698 /\*\*  
699 \* Gets the hashcode of the entry using temporary hard references.  
700 \* <p>  
701 \* This implementation uses <code>hashEntry</code> on the main map.  
702 \*  
703 \* @return the hashcode of the entry  
704 \*/  
705 @Override  
706 public int hashCode() {  
707 return parent.hashEntry(getKey(), getValue());  
708 }  
709  
710 /\*\*  
711 \* Constructs a reference of the given type to the given referent.  
712 \* The reference is registered with the queue for later purging.  
713 \*  
714 \* @param <T> the type of the referenced object  
715 \* @param type HARD, SOFT or WEAK  
716 \* @param referent the object to refer to  
717 \* @param hash the hash code of the <i>key</i> of the mapping;  
718 \* this number might be different from referent.hashCode() if  
719 \* the referent represents a value and not a key  
720 \* @return the reference to the object  
721 \*/  
722 protected <T> Object toReference(final ReferenceStrength type, final T referent, final int hash) {  
723 if (type == ReferenceStrength.HARD) {  
724 return referent;  
725 }  
726 if (type == ReferenceStrength.SOFT) {  
727 return new SoftRef<>(hash, referent, parent.queue);  
728 }  
729 if (type == ReferenceStrength.WEAK) {  
730 return new WeakRef<>(hash, referent, parent.queue);  
731 }  
732 throw new Error();  
733 }  
734  
735 /\*\*  
736 \* This is the callback for custom "after purge" logic  
737 \*/  
738 protected void onPurge() {  
739 // empty  
740 }  
741  
742 /\*\*  
743 \* Purges the specified reference  
744 \* @param ref the reference to purge  
745 \* @return true or false  
746 \*/  
747 protected boolean purge(final Reference<?> ref) {  
748 boolean r = parent.keyType != ReferenceStrength.HARD && key == ref;  
749 r = r || parent.valueType != ReferenceStrength.HARD && value == ref;  
750 if (r) {  
751 if (parent.keyType != ReferenceStrength.HARD) {  
752 ((Reference<?>) key).clear();  
753 }  
754 if (parent.valueType != ReferenceStrength.HARD) {  
755 ((Reference<?>) value).clear();  
756 } else if (parent.purgeValues) {  
757 nullValue();  
758 }  
759 }  
760 return r;  
761 }  
762  
763 /\*\*  
764 \* Gets the next entry in the bucket.  
765 \*  
766 \* @return the next entry in the bucket  
767 \*/  
768 protected ReferenceEntry<K, V> next() {  
769 return (ReferenceEntry<K, V>) next;  
770 }  
771  
772 /\*\*  
773 \* This method can be overriden to provide custom logic to purge value  
774 \*/  
775 protected void nullValue() {  
776 value = null;  
777 }  
778 }  
779  
780 //-----------------------------------------------------------------------  
781 /\*\*  
782 \* Base iterator class.  
783 \*/  
784 static class ReferenceBaseIterator<K, V> {  
785 /\*\* The parent map \*/  
786 final AbstractReferenceMap<K, V> parent;  
787  
788 // These fields keep track of where we are in the table.  
789 int index;  
790 ReferenceEntry<K, V> entry;  
791 ReferenceEntry<K, V> previous;  
792  
793 // These Object fields provide hard references to the  
794 // current and next entry; this assures that if hasNext()  
795 // returns true, next() will actually return a valid element.  
796 K currentKey, nextKey;  
797 V currentValue, nextValue;  
798  
799 int expectedModCount;  
800  
801 public ReferenceBaseIterator(final AbstractReferenceMap<K, V> parent) {  
802 super();  
803 this.parent = parent;  
804 index = parent.size() != 0 ? parent.data.length : 0;  
805 // have to do this here! size() invocation above  
806 // may have altered the modCount.  
807 expectedModCount = parent.modCount;  
808 }  
809  
810 public boolean hasNext() {  
811 checkMod();  
812 while (nextNull()) {  
813 ReferenceEntry<K, V> e = entry;  
814 int i = index;  
815 while (e == null && i > 0) {  
816 i--;  
817 e = (ReferenceEntry<K, V>) parent.data[i];  
818 }  
819 entry = e;  
820 index = i;  
821 if (e == null) {  
822 currentKey = null;  
823 currentValue = null;  
824 return false;  
825 }  
826 nextKey = e.getKey();  
827 nextValue = e.getValue();  
828 if (nextNull()) {  
829 entry = entry.next();  
830 }  
831 }  
832 return true;  
833 }  
834  
835 private void checkMod() {  
836 if (parent.modCount != expectedModCount) {  
837 throw new ConcurrentModificationException();  
838 }  
839 }  
840  
841 private boolean nextNull() {  
842 return nextKey == null || nextValue == null;  
843 }  
844  
845 protected ReferenceEntry<K, V> nextEntry() {  
846 checkMod();  
847 if (nextNull() && !hasNext()) {  
848 throw new NoSuchElementException();  
849 }  
850 previous = entry;  
851 entry = entry.next();  
852 currentKey = nextKey;  
853 currentValue = nextValue;  
854 nextKey = null;  
855 nextValue = null;  
856 return previous;  
857 }  
858  
859 protected ReferenceEntry<K, V> currentEntry() {  
860 checkMod();  
861 return previous;  
862 }  
863  
864 public void remove() {  
865 checkMod();  
866 if (previous == null) {  
867 throw new IllegalStateException();  
868 }  
869 parent.remove(currentKey);  
870 previous = null;  
871 currentKey = null;  
872 currentValue = null;  
873 expectedModCount = parent.modCount;  
874 }  
875 }  
876  
877 /\*\*  
878 \* The EntrySet iterator.  
879 \*/  
880 static class ReferenceEntrySetIterator<K, V>  
881 extends ReferenceBaseIterator<K, V> implements Iterator<Map.Entry<K, V>> {  
882  
883 public ReferenceEntrySetIterator(final AbstractReferenceMap<K, V> parent) {  
884 super(parent);  
885 }  
886  
887 @Override  
888 public Map.Entry<K, V> next() {  
889 return nextEntry();  
890 }  
891  
892 }  
893  
894 /\*\*  
895 \* The keySet iterator.  
896 \*/  
897 static class ReferenceKeySetIterator<K> extends ReferenceBaseIterator<K, Object> implements Iterator<K> {  
898  
899 @SuppressWarnings("unchecked")  
900 ReferenceKeySetIterator(final AbstractReferenceMap<K, ?> parent) {  
901 super((AbstractReferenceMap<K, Object>) parent);  
902 }  
903  
904 @Override  
905 public K next() {  
906 return nextEntry().getKey();  
907 }  
908 }  
909  
910 /\*\*  
911 \* The values iterator.  
912 \*/  
913 static class ReferenceValuesIterator<V> extends ReferenceBaseIterator<Object, V> implements Iterator<V> {  
914  
915 @SuppressWarnings("unchecked")  
916 ReferenceValuesIterator(final AbstractReferenceMap<?, V> parent) {  
917 super((AbstractReferenceMap<Object, V>) parent);  
918 }  
919  
920 @Override  
921 public V next() {  
922 return nextEntry().getValue();  
923 }  
924 }  
925  
926 /\*\*  
927 \* The MapIterator implementation.  
928 \*/  
929 static class ReferenceMapIterator<K, V> extends ReferenceBaseIterator<K, V> implements MapIterator<K, V> {  
930  
931 protected ReferenceMapIterator(final AbstractReferenceMap<K, V> parent) {  
932 super(parent);  
933 }  
934  
935 @Override  
936 public K next() {  
937 return nextEntry().getKey();  
938 }  
939  
940 @Override  
941 public K getKey() {  
942 final HashEntry<K, V> current = currentEntry();  
943 if (current == null) {  
944 throw new IllegalStateException(AbstractHashedMap.GETKEY\_INVALID);  
945 }  
946 return current.getKey();  
947 }  
948  
949 @Override  
950 public V getValue() {  
951 final HashEntry<K, V> current = currentEntry();  
952 if (current == null) {  
953 throw new IllegalStateException(AbstractHashedMap.GETVALUE\_INVALID);  
954 }  
955 return current.getValue();  
956 }  
957  
958 @Override  
959 public V setValue(final V value) {  
960 final HashEntry<K, V> current = currentEntry();  
961 if (current == null) {  
962 throw new IllegalStateException(AbstractHashedMap.SETVALUE\_INVALID);  
963 }  
964 return current.setValue(value);  
965 }  
966 }  
967  
968 //-----------------------------------------------------------------------  
969 // These two classes store the hashCode of the key of  
970 // of the mapping, so that after they're dequeued a quick  
971 // lookup of the bucket in the table can occur.  
972  
973 /\*\*  
974 \* A soft reference holder.  
975 \*/  
976 static class SoftRef<T> extends SoftReference<T> {  
977 /\*\* the hashCode of the key (even if the reference points to a value) \*/  
978 private final int hash;  
979  
980 public SoftRef(final int hash, final T r, final ReferenceQueue<? super T> q) {  
981 super(r, q);  
982 this.hash = hash;  
983 }  
984  
985 @Override  
986 public int hashCode() {  
987 return hash;  
988 }  
989 }  
990  
991 /\*\*  
992 \* A weak reference holder.  
993 \*/  
994 static class WeakRef<T> extends WeakReference<T> {  
995 /\*\* the hashCode of the key (even if the reference points to a value) \*/  
996 private final int hash;  
997  
998 public WeakRef(final int hash, final T r, final ReferenceQueue<? super T> q) {  
999 super(r, q);  
1000 this.hash = hash;  
1001 }  
1002  
1003 @Override  
1004 public int hashCode() {  
1005 return hash;  
1006 }  
1007 }  
1008  
1009 //-----------------------------------------------------------------------  
1010 /\*\*  
1011 \* Replaces the superclass method to store the state of this class.  
1012 \* <p>  
1013 \* Serialization is not one of the JDK's nicest topics. Normal serialization will  
1014 \* initialise the superclass before the subclass. Sometimes however, this isn't  
1015 \* what you want, as in this case the <code>put()</code> method on read can be  
1016 \* affected by subclass state.  
1017 \* <p>  
1018 \* The solution adopted here is to serialize the state data of this class in  
1019 \* this protected method. This method must be called by the  
1020 \* <code>writeObject()</code> of the first serializable subclass.  
1021 \* <p>  
1022 \* Subclasses may override if they have a specific field that must be present  
1023 \* on read before this implementation will work. Generally, the read determines  
1024 \* what must be serialized here, if anything.  
1025 \*  
1026 \* @param out the output stream  
1027 \* @throws IOException if an error occurs while writing to the stream  
1028 \*/  
1029 @Override  
1030 protected void doWriteObject(final ObjectOutputStream out) throws IOException {  
1031 out.writeInt(keyType.value);  
1032 out.writeInt(valueType.value);  
1033 out.writeBoolean(purgeValues);  
1034 out.writeFloat(loadFactor);  
1035 out.writeInt(data.length);  
1036 for (final MapIterator<K, V> it = mapIterator(); it.hasNext();) {  
1037 out.writeObject(it.next());  
1038 out.writeObject(it.getValue());  
1039 }  
1040 out.writeObject(null); // null terminate map  
1041 // do not call super.doWriteObject() as code there doesn't work for reference map  
1042 }  
1043  
1044 /\*\*  
1045 \* Replaces the superclass method to read the state of this class.  
1046 \* <p>  
1047 \* Serialization is not one of the JDK's nicest topics. Normal serialization will  
1048 \* initialise the superclass before the subclass. Sometimes however, this isn't  
1049 \* what you want, as in this case the <code>put()</code> method on read can be  
1050 \* affected by subclass state.  
1051 \* <p>  
1052 \* The solution adopted here is to deserialize the state data of this class in  
1053 \* this protected method. This method must be called by the  
1054 \* <code>readObject()</code> of the first serializable subclass.  
1055 \* <p>  
1056 \* Subclasses may override if the subclass has a specific field that must be present  
1057 \* before <code>put()</code> or <code>calculateThreshold()</code> will work correctly.  
1058 \*  
1059 \* @param in the input stream  
1060 \* @throws IOException if an error occurs while reading from the stream  
1061 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
1062 \*/  
1063 @Override  
1064 @SuppressWarnings("unchecked")  
1065 protected void doReadObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
1066 this.keyType = ReferenceStrength.resolve(in.readInt());  
1067 this.valueType = ReferenceStrength.resolve(in.readInt());  
1068 this.purgeValues = in.readBoolean();  
1069 this.loadFactor = in.readFloat();  
1070 final int capacity = in.readInt();  
1071 init();  
1072 data = new HashEntry[capacity];  
1073  
1074 // COLLECTIONS-599: Calculate threshold before populating, otherwise it will be 0  
1075 // when it hits AbstractHashedMap.checkCapacity() and so will unnecessarily  
1076 // double up the size of the "data" array during population.  
1077 //  
1078 // NB: AbstractHashedMap.doReadObject() DOES calculate the threshold before populating.  
1079 //  
1080 threshold = calculateThreshold(data.length, loadFactor);  
1081  
1082 while (true) {  
1083 final K key = (K) in.readObject();  
1084 if (key == null) {  
1085 break;  
1086 }  
1087 final V value = (V) in.readObject();  
1088 put(key, value);  
1089 }  
1090 // do not call super.doReadObject() as code there doesn't work for reference map  
1091 }  
1092  
1093 /\*\*  
1094 \* Provided protected read-only access to the key type.  
1095 \* @param type the type to check against.  
1096 \* @return true if keyType has the specified type  
1097 \*/  
1098 protected boolean isKeyType(final ReferenceStrength type) {  
1099 return this.keyType == type;  
1100 }  
1101  
1102 /\*\*  
1103 \* Provided protected read-only access to the value type.  
1104 \* @param type the type to check against.  
1105 \* @return true if valueType has the specified type  
1106 \*/  
1107 protected boolean isValueType(final ReferenceStrength type) {  
1108 return this.valueType == type;  
1109 }  
1110}