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.util.AbstractCollection;  
023import java.util.AbstractMap;  
024import java.util.AbstractSet;  
025import java.util.Collection;  
026import java.util.ConcurrentModificationException;  
027import java.util.Iterator;  
028import java.util.Map;  
029import java.util.NoSuchElementException;  
030import java.util.Set;  
031  
032import org.apache.commons.collections4.IterableMap;  
033import org.apache.commons.collections4.KeyValue;  
034import org.apache.commons.collections4.MapIterator;  
035import org.apache.commons.collections4.iterators.EmptyIterator;  
036import org.apache.commons.collections4.iterators.EmptyMapIterator;  
037  
038/\*\*  
039 \* An abstract implementation of a hash-based map which provides numerous points for  
040 \* subclasses to override.  
041 \* <p>  
042 \* This class implements all the features necessary for a subclass hash-based map.  
043 \* Key-value entries are stored in instances of the <code>HashEntry</code> class,  
044 \* which can be overridden and replaced. The iterators can similarly be replaced,  
045 \* without the need to replace the KeySet, EntrySet and Values view classes.  
046 \* <p>  
047 \* Overridable methods are provided to change the default hashing behaviour, and  
048 \* to change how entries are added to and removed from the map. Hopefully, all you  
049 \* need for unusual subclasses is here.  
050 \* <p>  
051 \* NOTE: From Commons Collections 3.1 this class extends AbstractMap.  
052 \* This is to provide backwards compatibility for ReferenceMap between v3.0 and v3.1.  
053 \* This extends clause will be removed in v5.0.  
054 \*  
055 \* @param <K> the type of the keys in this map  
056 \* @param <V> the type of the values in this map  
057 \* @since 3.0  
058 \*/  
059public class AbstractHashedMap<K, V> extends AbstractMap<K, V> implements IterableMap<K, V> {  
060  
061 protected static final String NO\_NEXT\_ENTRY = "No next() entry in the iteration";  
062 protected static final String NO\_PREVIOUS\_ENTRY = "No previous() entry in the iteration";  
063 protected static final String REMOVE\_INVALID = "remove() can only be called once after next()";  
064 protected static final String GETKEY\_INVALID = "getKey() can only be called after next() and before remove()";  
065 protected static final String GETVALUE\_INVALID = "getValue() can only be called after next() and before remove()";  
066 protected static final String SETVALUE\_INVALID = "setValue() can only be called after next() and before remove()";  
067  
068 /\*\* The default capacity to use \*/  
069 protected static final int DEFAULT\_CAPACITY = 16;  
070 /\*\* The default threshold to use \*/  
071 protected static final int DEFAULT\_THRESHOLD = 12;  
072 /\*\* The default load factor to use \*/  
073 protected static final float DEFAULT\_LOAD\_FACTOR = 0.75f;  
074 /\*\* The maximum capacity allowed \*/  
075 protected static final int MAXIMUM\_CAPACITY = 1 << 30;  
076 /\*\* An object for masking null \*/  
077 protected static final Object NULL = new Object();  
078  
079 /\*\* Load factor, normally 0.75 \*/  
080 transient float loadFactor;  
081 /\*\* The size of the map \*/  
082 transient int size;  
083 /\*\* Map entries \*/  
084 transient HashEntry<K, V>[] data;  
085 /\*\* Size at which to rehash \*/  
086 transient int threshold;  
087 /\*\* Modification count for iterators \*/  
088 transient int modCount;  
089 /\*\* Entry set \*/  
090 transient EntrySet<K, V> entrySet;  
091 /\*\* Key set \*/  
092 transient KeySet<K> keySet;  
093 /\*\* Values \*/  
094 transient Values<V> values;  
095  
096 /\*\*  
097 \* Constructor only used in deserialization, do not use otherwise.  
098 \*/  
099 protected AbstractHashedMap() {  
100 super();  
101 }  
102  
103 /\*\*  
104 \* Constructor which performs no validation on the passed in parameters.  
105 \*  
106 \* @param initialCapacity the initial capacity, must be a power of two  
107 \* @param loadFactor the load factor, must be > 0.0f and generally < 1.0f  
108 \* @param threshold the threshold, must be sensible  
109 \*/  
110 @SuppressWarnings("unchecked")  
111 protected AbstractHashedMap(final int initialCapacity, final float loadFactor, final int threshold) {  
112 super();  
113 this.loadFactor = loadFactor;  
114 this.data = new HashEntry[initialCapacity];  
115 this.threshold = threshold;  
116 init();  
117 }  
118  
119 /\*\*  
120 \* Constructs a new, empty map with the specified initial capacity and  
121 \* default load factor.  
122 \*  
123 \* @param initialCapacity the initial capacity  
124 \* @throws IllegalArgumentException if the initial capacity is negative  
125 \*/  
126 protected AbstractHashedMap(final int initialCapacity) {  
127 this(initialCapacity, DEFAULT\_LOAD\_FACTOR);  
128 }  
129  
130 /\*\*  
131 \* Constructs a new, empty map with the specified initial capacity and  
132 \* load factor.  
133 \*  
134 \* @param initialCapacity the initial capacity  
135 \* @param loadFactor the load factor  
136 \* @throws IllegalArgumentException if the initial capacity is negative  
137 \* @throws IllegalArgumentException if the load factor is less than or equal to zero  
138 \*/  
139 @SuppressWarnings("unchecked")  
140 protected AbstractHashedMap(int initialCapacity, final float loadFactor) {  
141 super();  
142 if (initialCapacity < 0) {  
143 throw new IllegalArgumentException("Initial capacity must be a non negative number");  
144 }  
145 if (loadFactor <= 0.0f || Float.isNaN(loadFactor)) {  
146 throw new IllegalArgumentException("Load factor must be greater than 0");  
147 }  
148 this.loadFactor = loadFactor;  
149 initialCapacity = calculateNewCapacity(initialCapacity);  
150 this.threshold = calculateThreshold(initialCapacity, loadFactor);  
151 this.data = new HashEntry[initialCapacity];  
152 init();  
153 }  
154  
155 /\*\*  
156 \* Constructor copying elements from another map.  
157 \*  
158 \* @param map the map to copy  
159 \* @throws NullPointerException if the map is null  
160 \*/  
161 protected AbstractHashedMap(final Map<? extends K, ? extends V> map) {  
162 this(Math.max(2 \* map.size(), DEFAULT\_CAPACITY), DEFAULT\_LOAD\_FACTOR);  
163 \_putAll(map);  
164 }  
165  
166 /\*\*  
167 \* Initialise subclasses during construction, cloning or deserialization.  
168 \*/  
169 protected void init() {  
170 }  
171  
172 //-----------------------------------------------------------------------  
173 /\*\*  
174 \* Gets the value mapped to the key specified.  
175 \*  
176 \* @param key the key  
177 \* @return the mapped value, null if no match  
178 \*/  
179 @Override  
180 public V get(Object key) {  
181 key = convertKey(key);  
182 final int hashCode = hash(key);  
183 HashEntry<K, V> entry = data[hashIndex(hashCode, data.length)]; // no local for hash index  
184 while (entry != null) {  
185 if (entry.hashCode == hashCode && isEqualKey(key, entry.key)) {  
186 return entry.getValue();  
187 }  
188 entry = entry.next;  
189 }  
190 return null;  
191 }  
192  
193 /\*\*  
194 \* Gets the size of the map.  
195 \*  
196 \* @return the size  
197 \*/  
198 @Override  
199 public int size() {  
200 return size;  
201 }  
202  
203 /\*\*  
204 \* Checks whether the map is currently empty.  
205 \*  
206 \* @return true if the map is currently size zero  
207 \*/  
208 @Override  
209 public boolean isEmpty() {  
210 return size == 0;  
211 }  
212  
213 //-----------------------------------------------------------------------  
214 /\*\*  
215 \* Checks whether the map contains the specified key.  
216 \*  
217 \* @param key the key to search for  
218 \* @return true if the map contains the key  
219 \*/  
220 @Override  
221 public boolean containsKey(Object key) {  
222 key = convertKey(key);  
223 final int hashCode = hash(key);  
224 HashEntry<K, V> entry = data[hashIndex(hashCode, data.length)]; // no local for hash index  
225 while (entry != null) {  
226 if (entry.hashCode == hashCode && isEqualKey(key, entry.key)) {  
227 return true;  
228 }  
229 entry = entry.next;  
230 }  
231 return false;  
232 }  
233  
234 /\*\*  
235 \* Checks whether the map contains the specified value.  
236 \*  
237 \* @param value the value to search for  
238 \* @return true if the map contains the value  
239 \*/  
240 @Override  
241 public boolean containsValue(final Object value) {  
242 if (value == null) {  
243 for (final HashEntry<K, V> element : data) {  
244 HashEntry<K, V> entry = element;  
245 while (entry != null) {  
246 if (entry.getValue() == null) {  
247 return true;  
248 }  
249 entry = entry.next;  
250 }  
251 }  
252 } else {  
253 for (final HashEntry<K, V> element : data) {  
254 HashEntry<K, V> entry = element;  
255 while (entry != null) {  
256 if (isEqualValue(value, entry.getValue())) {  
257 return true;  
258 }  
259 entry = entry.next;  
260 }  
261 }  
262 }  
263 return false;  
264 }  
265  
266 //-----------------------------------------------------------------------  
267 /\*\*  
268 \* Puts a key-value mapping into this map.  
269 \*  
270 \* @param key the key to add  
271 \* @param value the value to add  
272 \* @return the value previously mapped to this key, null if none  
273 \*/  
274 @Override  
275 public V put(final K key, final V value) {  
276 final Object convertedKey = convertKey(key);  
277 final int hashCode = hash(convertedKey);  
278 final int index = hashIndex(hashCode, data.length);  
279 HashEntry<K, V> entry = data[index];  
280 while (entry != null) {  
281 if (entry.hashCode == hashCode && isEqualKey(convertedKey, entry.key)) {  
282 final V oldValue = entry.getValue();  
283 updateEntry(entry, value);  
284 return oldValue;  
285 }  
286 entry = entry.next;  
287 }  
288  
289 addMapping(index, hashCode, key, value);  
290 return null;  
291 }  
292  
293 /\*\*  
294 \* Puts all the values from the specified map into this map.  
295 \* <p>  
296 \* This implementation iterates around the specified map and  
297 \* uses {@link #put(Object, Object)}.  
298 \*  
299 \* @param map the map to add  
300 \* @throws NullPointerException if the map is null  
301 \*/  
302 @Override  
303 public void putAll(final Map<? extends K, ? extends V> map) {  
304 \_putAll(map);  
305 }  
306  
307 /\*\*  
308 \* Puts all the values from the specified map into this map.  
309 \* <p>  
310 \* This implementation iterates around the specified map and  
311 \* uses {@link #put(Object, Object)}.  
312 \* <p>  
313 \* It is private to allow the constructor to still call it  
314 \* even when putAll is overriden.  
315 \*  
316 \* @param map the map to add  
317 \* @throws NullPointerException if the map is null  
318 \*/  
319 private void \_putAll(final Map<? extends K, ? extends V> map) {  
320 final int mapSize = map.size();  
321 if (mapSize == 0) {  
322 return;  
323 }  
324 final int newSize = (int) ((size + mapSize) / loadFactor + 1);  
325 ensureCapacity(calculateNewCapacity(newSize));  
326 for (final Map.Entry<? extends K, ? extends V> entry: map.entrySet()) {  
327 put(entry.getKey(), entry.getValue());  
328 }  
329 }  
330  
331 /\*\*  
332 \* Removes the specified mapping from this map.  
333 \*  
334 \* @param key the mapping to remove  
335 \* @return the value mapped to the removed key, null if key not in map  
336 \*/  
337 @Override  
338 public V remove(Object key) {  
339 key = convertKey(key);  
340 final int hashCode = hash(key);  
341 final int index = hashIndex(hashCode, data.length);  
342 HashEntry<K, V> entry = data[index];  
343 HashEntry<K, V> previous = null;  
344 while (entry != null) {  
345 if (entry.hashCode == hashCode && isEqualKey(key, entry.key)) {  
346 final V oldValue = entry.getValue();  
347 removeMapping(entry, index, previous);  
348 return oldValue;  
349 }  
350 previous = entry;  
351 entry = entry.next;  
352 }  
353 return null;  
354 }  
355  
356 /\*\*  
357 \* Clears the map, resetting the size to zero and nullifying references  
358 \* to avoid garbage collection issues.  
359 \*/  
360 @Override  
361 public void clear() {  
362 modCount++;  
363 final HashEntry<K, V>[] data = this.data;  
364 for (int i = data.length - 1; i >= 0; i--) {  
365 data[i] = null;  
366 }  
367 size = 0;  
368 }  
369  
370 //-----------------------------------------------------------------------  
371 /\*\*  
372 \* Converts input keys to another object for storage in the map.  
373 \* This implementation masks nulls.  
374 \* Subclasses can override this to perform alternate key conversions.  
375 \* <p>  
376 \* The reverse conversion can be changed, if required, by overriding the  
377 \* getKey() method in the hash entry.  
378 \*  
379 \* @param key the key convert  
380 \* @return the converted key  
381 \*/  
382 protected Object convertKey(final Object key) {  
383 return key == null ? NULL : key;  
384 }  
385  
386 /\*\*  
387 \* Gets the hash code for the key specified.  
388 \* This implementation uses the additional hashing routine from JDK1.4.  
389 \* Subclasses can override this to return alternate hash codes.  
390 \*  
391 \* @param key the key to get a hash code for  
392 \* @return the hash code  
393 \*/  
394 protected int hash(final Object key) {  
395 // same as JDK 1.4  
396 int h = key.hashCode();  
397 h += ~(h << 9);  
398 h ^= h >>> 14;  
399 h += h << 4;  
400 h ^= h >>> 10;  
401 return h;  
402 }  
403  
404 /\*\*  
405 \* Compares two keys, in internal converted form, to see if they are equal.  
406 \* This implementation uses the equals method and assumes neither key is null.  
407 \* Subclasses can override this to match differently.  
408 \*  
409 \* @param key1 the first key to compare passed in from outside  
410 \* @param key2 the second key extracted from the entry via <code>entry.key</code>  
411 \* @return true if equal  
412 \*/  
413 protected boolean isEqualKey(final Object key1, final Object key2) {  
414 return key1 == key2 || key1.equals(key2);  
415 }  
416  
417 /\*\*  
418 \* Compares two values, in external form, to see if they are equal.  
419 \* This implementation uses the equals method and assumes neither value is null.  
420 \* Subclasses can override this to match differently.  
421 \*  
422 \* @param value1 the first value to compare passed in from outside  
423 \* @param value2 the second value extracted from the entry via <code>getValue()</code>  
424 \* @return true if equal  
425 \*/  
426 protected boolean isEqualValue(final Object value1, final Object value2) {  
427 return value1 == value2 || value1.equals(value2);  
428 }  
429  
430 /\*\*  
431 \* Gets the index into the data storage for the hashCode specified.  
432 \* This implementation uses the least significant bits of the hashCode.  
433 \* Subclasses can override this to return alternate bucketing.  
434 \*  
435 \* @param hashCode the hash code to use  
436 \* @param dataSize the size of the data to pick a bucket from  
437 \* @return the bucket index  
438 \*/  
439 protected int hashIndex(final int hashCode, final int dataSize) {  
440 return hashCode & dataSize - 1;  
441 }  
442  
443 //-----------------------------------------------------------------------  
444 /\*\*  
445 \* Gets the entry mapped to the key specified.  
446 \* <p>  
447 \* This method exists for subclasses that may need to perform a multi-step  
448 \* process accessing the entry. The public methods in this class don't use this  
449 \* method to gain a small performance boost.  
450 \*  
451 \* @param key the key  
452 \* @return the entry, null if no match  
453 \*/  
454 protected HashEntry<K, V> getEntry(Object key) {  
455 key = convertKey(key);  
456 final int hashCode = hash(key);  
457 HashEntry<K, V> entry = data[hashIndex(hashCode, data.length)]; // no local for hash index  
458 while (entry != null) {  
459 if (entry.hashCode == hashCode && isEqualKey(key, entry.key)) {  
460 return entry;  
461 }  
462 entry = entry.next;  
463 }  
464 return null;  
465 }  
466  
467 //-----------------------------------------------------------------------  
468 /\*\*  
469 \* Updates an existing key-value mapping to change the value.  
470 \* <p>  
471 \* This implementation calls <code>setValue()</code> on the entry.  
472 \* Subclasses could override to handle changes to the map.  
473 \*  
474 \* @param entry the entry to update  
475 \* @param newValue the new value to store  
476 \*/  
477 protected void updateEntry(final HashEntry<K, V> entry, final V newValue) {  
478 entry.setValue(newValue);  
479 }  
480  
481 /\*\*  
482 \* Reuses an existing key-value mapping, storing completely new data.  
483 \* <p>  
484 \* This implementation sets all the data fields on the entry.  
485 \* Subclasses could populate additional entry fields.  
486 \*  
487 \* @param entry the entry to update, not null  
488 \* @param hashIndex the index in the data array  
489 \* @param hashCode the hash code of the key to add  
490 \* @param key the key to add  
491 \* @param value the value to add  
492 \*/  
493 protected void reuseEntry(final HashEntry<K, V> entry, final int hashIndex, final int hashCode,  
494 final K key, final V value) {  
495 entry.next = data[hashIndex];  
496 entry.hashCode = hashCode;  
497 entry.key = key;  
498 entry.value = value;  
499 }  
500  
501 //-----------------------------------------------------------------------  
502 /\*\*  
503 \* Adds a new key-value mapping into this map.  
504 \* <p>  
505 \* This implementation calls <code>createEntry()</code>, <code>addEntry()</code>  
506 \* and <code>checkCapacity()</code>.  
507 \* It also handles changes to <code>modCount</code> and <code>size</code>.  
508 \* Subclasses could override to fully control adds to the map.  
509 \*  
510 \* @param hashIndex the index into the data array to store at  
511 \* @param hashCode the hash code of the key to add  
512 \* @param key the key to add  
513 \* @param value the value to add  
514 \*/  
515 protected void addMapping(final int hashIndex, final int hashCode, final K key, final V value) {  
516 modCount++;  
517 final HashEntry<K, V> entry = createEntry(data[hashIndex], hashCode, key, value);  
518 addEntry(entry, hashIndex);  
519 size++;  
520 checkCapacity();  
521 }  
522  
523 /\*\*  
524 \* Creates an entry to store the key-value data.  
525 \* <p>  
526 \* This implementation creates a new HashEntry instance.  
527 \* Subclasses can override this to return a different storage class,  
528 \* or implement caching.  
529 \*  
530 \* @param next the next entry in sequence  
531 \* @param hashCode the hash code to use  
532 \* @param key the key to store  
533 \* @param value the value to store  
534 \* @return the newly created entry  
535 \*/  
536 protected HashEntry<K, V> createEntry(final HashEntry<K, V> next, final int hashCode, final K key, final V value) {  
537 return new HashEntry<>(next, hashCode, convertKey(key), value);  
538 }  
539  
540 /\*\*  
541 \* Adds an entry into this map.  
542 \* <p>  
543 \* This implementation adds the entry to the data storage table.  
544 \* Subclasses could override to handle changes to the map.  
545 \*  
546 \* @param entry the entry to add  
547 \* @param hashIndex the index into the data array to store at  
548 \*/  
549 protected void addEntry(final HashEntry<K, V> entry, final int hashIndex) {  
550 data[hashIndex] = entry;  
551 }  
552  
553 //-----------------------------------------------------------------------  
554 /\*\*  
555 \* Removes a mapping from the map.  
556 \* <p>  
557 \* This implementation calls <code>removeEntry()</code> and <code>destroyEntry()</code>.  
558 \* It also handles changes to <code>modCount</code> and <code>size</code>.  
559 \* Subclasses could override to fully control removals from the map.  
560 \*  
561 \* @param entry the entry to remove  
562 \* @param hashIndex the index into the data structure  
563 \* @param previous the previous entry in the chain  
564 \*/  
565 protected void removeMapping(final HashEntry<K, V> entry, final int hashIndex, final HashEntry<K, V> previous) {  
566 modCount++;  
567 removeEntry(entry, hashIndex, previous);  
568 size--;  
569 destroyEntry(entry);  
570 }  
571  
572 /\*\*  
573 \* Removes an entry from the chain stored in a particular index.  
574 \* <p>  
575 \* This implementation removes the entry from the data storage table.  
576 \* The size is not updated.  
577 \* Subclasses could override to handle changes to the map.  
578 \*  
579 \* @param entry the entry to remove  
580 \* @param hashIndex the index into the data structure  
581 \* @param previous the previous entry in the chain  
582 \*/  
583 protected void removeEntry(final HashEntry<K, V> entry, final int hashIndex, final HashEntry<K, V> previous) {  
584 if (previous == null) {  
585 data[hashIndex] = entry.next;  
586 } else {  
587 previous.next = entry.next;  
588 }  
589 }  
590  
591 /\*\*  
592 \* Kills an entry ready for the garbage collector.  
593 \* <p>  
594 \* This implementation prepares the HashEntry for garbage collection.  
595 \* Subclasses can override this to implement caching (override clear as well).  
596 \*  
597 \* @param entry the entry to destroy  
598 \*/  
599 protected void destroyEntry(final HashEntry<K, V> entry) {  
600 entry.next = null;  
601 entry.key = null;  
602 entry.value = null;  
603 }  
604  
605 //-----------------------------------------------------------------------  
606 /\*\*  
607 \* Checks the capacity of the map and enlarges it if necessary.  
608 \* <p>  
609 \* This implementation uses the threshold to check if the map needs enlarging  
610 \*/  
611 protected void checkCapacity() {  
612 if (size >= threshold) {  
613 final int newCapacity = data.length \* 2;  
614 if (newCapacity <= MAXIMUM\_CAPACITY) {  
615 ensureCapacity(newCapacity);  
616 }  
617 }  
618 }  
619  
620 /\*\*  
621 \* Changes the size of the data structure to the capacity proposed.  
622 \*  
623 \* @param newCapacity the new capacity of the array (a power of two, less or equal to max)  
624 \*/  
625 @SuppressWarnings("unchecked")  
626 protected void ensureCapacity(final int newCapacity) {  
627 final int oldCapacity = data.length;  
628 if (newCapacity <= oldCapacity) {  
629 return;  
630 }  
631 if (size == 0) {  
632 threshold = calculateThreshold(newCapacity, loadFactor);  
633 data = new HashEntry[newCapacity];  
634 } else {  
635 final HashEntry<K, V> oldEntries[] = data;  
636 final HashEntry<K, V> newEntries[] = new HashEntry[newCapacity];  
637  
638 modCount++;  
639 for (int i = oldCapacity - 1; i >= 0; i--) {  
640 HashEntry<K, V> entry = oldEntries[i];  
641 if (entry != null) {  
642 oldEntries[i] = null; // gc  
643 do {  
644 final HashEntry<K, V> next = entry.next;  
645 final int index = hashIndex(entry.hashCode, newCapacity);  
646 entry.next = newEntries[index];  
647 newEntries[index] = entry;  
648 entry = next;  
649 } while (entry != null);  
650 }  
651 }  
652 threshold = calculateThreshold(newCapacity, loadFactor);  
653 data = newEntries;  
654 }  
655 }  
656  
657 /\*\*  
658 \* Calculates the new capacity of the map.  
659 \* This implementation normalizes the capacity to a power of two.  
660 \*  
661 \* @param proposedCapacity the proposed capacity  
662 \* @return the normalized new capacity  
663 \*/  
664 protected int calculateNewCapacity(final int proposedCapacity) {  
665 int newCapacity = 1;  
666 if (proposedCapacity > MAXIMUM\_CAPACITY) {  
667 newCapacity = MAXIMUM\_CAPACITY;  
668 } else {  
669 while (newCapacity < proposedCapacity) {  
670 newCapacity <<= 1; // multiply by two  
671 }  
672 if (newCapacity > MAXIMUM\_CAPACITY) {  
673 newCapacity = MAXIMUM\_CAPACITY;  
674 }  
675 }  
676 return newCapacity;  
677 }  
678  
679 /\*\*  
680 \* Calculates the new threshold of the map, where it will be resized.  
681 \* This implementation uses the load factor.  
682 \*  
683 \* @param newCapacity the new capacity  
684 \* @param factor the load factor  
685 \* @return the new resize threshold  
686 \*/  
687 protected int calculateThreshold(final int newCapacity, final float factor) {  
688 return (int) (newCapacity \* factor);  
689 }  
690  
691 //-----------------------------------------------------------------------  
692 /\*\*  
693 \* Gets the <code>next</code> field from a <code>HashEntry</code>.  
694 \* Used in subclasses that have no visibility of the field.  
695 \*  
696 \* @param entry the entry to query, must not be null  
697 \* @return the <code>next</code> field of the entry  
698 \* @throws NullPointerException if the entry is null  
699 \* @since 3.1  
700 \*/  
701 protected HashEntry<K, V> entryNext(final HashEntry<K, V> entry) {  
702 return entry.next;  
703 }  
704  
705 /\*\*  
706 \* Gets the <code>hashCode</code> field from a <code>HashEntry</code>.  
707 \* Used in subclasses that have no visibility of the field.  
708 \*  
709 \* @param entry the entry to query, must not be null  
710 \* @return the <code>hashCode</code> field of the entry  
711 \* @throws NullPointerException if the entry is null  
712 \* @since 3.1  
713 \*/  
714 protected int entryHashCode(final HashEntry<K, V> entry) {  
715 return entry.hashCode;  
716 }  
717  
718 /\*\*  
719 \* Gets the <code>key</code> field from a <code>HashEntry</code>.  
720 \* Used in subclasses that have no visibility of the field.  
721 \*  
722 \* @param entry the entry to query, must not be null  
723 \* @return the <code>key</code> field of the entry  
724 \* @throws NullPointerException if the entry is null  
725 \* @since 3.1  
726 \*/  
727 protected K entryKey(final HashEntry<K, V> entry) {  
728 return entry.getKey();  
729 }  
730  
731 /\*\*  
732 \* Gets the <code>value</code> field from a <code>HashEntry</code>.  
733 \* Used in subclasses that have no visibility of the field.  
734 \*  
735 \* @param entry the entry to query, must not be null  
736 \* @return the <code>value</code> field of the entry  
737 \* @throws NullPointerException if the entry is null  
738 \* @since 3.1  
739 \*/  
740 protected V entryValue(final HashEntry<K, V> entry) {  
741 return entry.getValue();  
742 }  
743  
744 //-----------------------------------------------------------------------  
745 /\*\*  
746 \* Gets an iterator over the map.  
747 \* Changes made to the iterator affect this map.  
748 \* <p>  
749 \* A MapIterator returns the keys in the map. It also provides convenient  
750 \* methods to get the key and value, and set the value.  
751 \* It avoids the need to create an entrySet/keySet/values object.  
752 \* It also avoids creating the Map.Entry object.  
753 \*  
754 \* @return the map iterator  
755 \*/  
756 @Override  
757 public MapIterator<K, V> mapIterator() {  
758 if (size == 0) {  
759 return EmptyMapIterator.<K, V>emptyMapIterator();  
760 }  
761 return new HashMapIterator<>(this);  
762 }  
763  
764 /\*\*  
765 \* MapIterator implementation.  
766 \*/  
767 protected static class HashMapIterator<K, V> extends HashIterator<K, V> implements MapIterator<K, V> {  
768  
769 protected HashMapIterator(final AbstractHashedMap<K, V> parent) {  
770 super(parent);  
771 }  
772  
773 @Override  
774 public K next() {  
775 return super.nextEntry().getKey();  
776 }  
777  
778 @Override  
779 public K getKey() {  
780 final HashEntry<K, V> current = currentEntry();  
781 if (current == null) {  
782 throw new IllegalStateException(AbstractHashedMap.GETKEY\_INVALID);  
783 }  
784 return current.getKey();  
785 }  
786  
787 @Override  
788 public V getValue() {  
789 final HashEntry<K, V> current = currentEntry();  
790 if (current == null) {  
791 throw new IllegalStateException(AbstractHashedMap.GETVALUE\_INVALID);  
792 }  
793 return current.getValue();  
794 }  
795  
796 @Override  
797 public V setValue(final V value) {  
798 final HashEntry<K, V> current = currentEntry();  
799 if (current == null) {  
800 throw new IllegalStateException(AbstractHashedMap.SETVALUE\_INVALID);  
801 }  
802 return current.setValue(value);  
803 }  
804 }  
805  
806 //-----------------------------------------------------------------------  
807 /\*\*  
808 \* Gets the entrySet view of the map.  
809 \* Changes made to the view affect this map.  
810 \* To simply iterate through the entries, use {@link #mapIterator()}.  
811 \*  
812 \* @return the entrySet view  
813 \*/  
814 @Override  
815 public Set<Map.Entry<K, V>> entrySet() {  
816 if (entrySet == null) {  
817 entrySet = new EntrySet<>(this);  
818 }  
819 return entrySet;  
820 }  
821  
822 /\*\*  
823 \* Creates an entry set iterator.  
824 \* Subclasses can override this to return iterators with different properties.  
825 \*  
826 \* @return the entrySet iterator  
827 \*/  
828 protected Iterator<Map.Entry<K, V>> createEntrySetIterator() {  
829 if (size() == 0) {  
830 return EmptyIterator.<Map.Entry<K, V>>emptyIterator();  
831 }  
832 return new EntrySetIterator<>(this);  
833 }  
834  
835 /\*\*  
836 \* EntrySet implementation.  
837 \*/  
838 protected static class EntrySet<K, V> extends AbstractSet<Map.Entry<K, V>> {  
839 /\*\* The parent map \*/  
840 private final AbstractHashedMap<K, V> parent;  
841  
842 protected EntrySet(final AbstractHashedMap<K, V> parent) {  
843 super();  
844 this.parent = parent;  
845 }  
846  
847 @Override  
848 public int size() {  
849 return parent.size();  
850 }  
851  
852 @Override  
853 public void clear() {  
854 parent.clear();  
855 }  
856  
857 @Override  
858 public boolean contains(final Object entry) {  
859 if (entry instanceof Map.Entry) {  
860 final Map.Entry<?, ?> e = (Map.Entry<?, ?>) entry;  
861 final Entry<K, V> match = parent.getEntry(e.getKey());  
862 return match != null && match.equals(e);  
863 }  
864 return false;  
865 }  
866  
867 @Override  
868 public boolean remove(final Object obj) {  
869 if (obj instanceof Map.Entry == false) {  
870 return false;  
871 }  
872 if (contains(obj) == false) {  
873 return false;  
874 }  
875 final Map.Entry<?, ?> entry = (Map.Entry<?, ?>) obj;  
876 parent.remove(entry.getKey());  
877 return true;  
878 }  
879  
880 @Override  
881 public Iterator<Map.Entry<K, V>> iterator() {  
882 return parent.createEntrySetIterator();  
883 }  
884 }  
885  
886 /\*\*  
887 \* EntrySet iterator.  
888 \*/  
889 protected static class EntrySetIterator<K, V> extends HashIterator<K, V> implements Iterator<Map.Entry<K, V>> {  
890  
891 protected EntrySetIterator(final AbstractHashedMap<K, V> parent) {  
892 super(parent);  
893 }  
894  
895 @Override  
896 public Map.Entry<K, V> next() {  
897 return super.nextEntry();  
898 }  
899 }  
900  
901 //-----------------------------------------------------------------------  
902 /\*\*  
903 \* Gets the keySet view of the map.  
904 \* Changes made to the view affect this map.  
905 \* To simply iterate through the keys, use {@link #mapIterator()}.  
906 \*  
907 \* @return the keySet view  
908 \*/  
909 @Override  
910 public Set<K> keySet() {  
911 if (keySet == null) {  
912 keySet = new KeySet<>(this);  
913 }  
914 return keySet;  
915 }  
916  
917 /\*\*  
918 \* Creates a key set iterator.  
919 \* Subclasses can override this to return iterators with different properties.  
920 \*  
921 \* @return the keySet iterator  
922 \*/  
923 protected Iterator<K> createKeySetIterator() {  
924 if (size() == 0) {  
925 return EmptyIterator.<K>emptyIterator();  
926 }  
927 return new KeySetIterator<>(this);  
928 }  
929  
930 /\*\*  
931 \* KeySet implementation.  
932 \*/  
933 protected static class KeySet<K> extends AbstractSet<K> {  
934 /\*\* The parent map \*/  
935 private final AbstractHashedMap<K, ?> parent;  
936  
937 protected KeySet(final AbstractHashedMap<K, ?> parent) {  
938 super();  
939 this.parent = parent;  
940 }  
941  
942 @Override  
943 public int size() {  
944 return parent.size();  
945 }  
946  
947 @Override  
948 public void clear() {  
949 parent.clear();  
950 }  
951  
952 @Override  
953 public boolean contains(final Object key) {  
954 return parent.containsKey(key);  
955 }  
956  
957 @Override  
958 public boolean remove(final Object key) {  
959 final boolean result = parent.containsKey(key);  
960 parent.remove(key);  
961 return result;  
962 }  
963  
964 @Override  
965 public Iterator<K> iterator() {  
966 return parent.createKeySetIterator();  
967 }  
968 }  
969  
970 /\*\*  
971 \* KeySet iterator.  
972 \*/  
973 protected static class KeySetIterator<K> extends HashIterator<K, Object> implements Iterator<K> {  
974  
975 @SuppressWarnings("unchecked")  
976 protected KeySetIterator(final AbstractHashedMap<K, ?> parent) {  
977 super((AbstractHashedMap<K, Object>) parent);  
978 }  
979  
980 @Override  
981 public K next() {  
982 return super.nextEntry().getKey();  
983 }  
984 }  
985  
986 //-----------------------------------------------------------------------  
987 /\*\*  
988 \* Gets the values view of the map.  
989 \* Changes made to the view affect this map.  
990 \* To simply iterate through the values, use {@link #mapIterator()}.  
991 \*  
992 \* @return the values view  
993 \*/  
994 @Override  
995 public Collection<V> values() {  
996 if (values == null) {  
997 values = new Values<>(this);  
998 }  
999 return values;  
1000 }  
1001  
1002 /\*\*  
1003 \* Creates a values iterator.  
1004 \* Subclasses can override this to return iterators with different properties.  
1005 \*  
1006 \* @return the values iterator  
1007 \*/  
1008 protected Iterator<V> createValuesIterator() {  
1009 if (size() == 0) {  
1010 return EmptyIterator.<V>emptyIterator();  
1011 }  
1012 return new ValuesIterator<>(this);  
1013 }  
1014  
1015 /\*\*  
1016 \* Values implementation.  
1017 \*/  
1018 protected static class Values<V> extends AbstractCollection<V> {  
1019 /\*\* The parent map \*/  
1020 private final AbstractHashedMap<?, V> parent;  
1021  
1022 protected Values(final AbstractHashedMap<?, V> parent) {  
1023 super();  
1024 this.parent = parent;  
1025 }  
1026  
1027 @Override  
1028 public int size() {  
1029 return parent.size();  
1030 }  
1031  
1032 @Override  
1033 public void clear() {  
1034 parent.clear();  
1035 }  
1036  
1037 @Override  
1038 public boolean contains(final Object value) {  
1039 return parent.containsValue(value);  
1040 }  
1041  
1042 @Override  
1043 public Iterator<V> iterator() {  
1044 return parent.createValuesIterator();  
1045 }  
1046 }  
1047  
1048 /\*\*  
1049 \* Values iterator.  
1050 \*/  
1051 protected static class ValuesIterator<V> extends HashIterator<Object, V> implements Iterator<V> {  
1052  
1053 @SuppressWarnings("unchecked")  
1054 protected ValuesIterator(final AbstractHashedMap<?, V> parent) {  
1055 super((AbstractHashedMap<Object, V>) parent);  
1056 }  
1057  
1058 @Override  
1059 public V next() {  
1060 return super.nextEntry().getValue();  
1061 }  
1062 }  
1063  
1064 //-----------------------------------------------------------------------  
1065 /\*\*  
1066 \* HashEntry used to store the data.  
1067 \* <p>  
1068 \* If you subclass <code>AbstractHashedMap</code> but not <code>HashEntry</code>  
1069 \* then you will not be able to access the protected fields.  
1070 \* The <code>entryXxx()</code> methods on <code>AbstractHashedMap</code> exist  
1071 \* to provide the necessary access.  
1072 \*/  
1073 protected static class HashEntry<K, V> implements Map.Entry<K, V>, KeyValue<K, V> {  
1074 /\*\* The next entry in the hash chain \*/  
1075 protected HashEntry<K, V> next;  
1076 /\*\* The hash code of the key \*/  
1077 protected int hashCode;  
1078 /\*\* The key \*/  
1079 protected Object key;  
1080 /\*\* The value \*/  
1081 protected Object value;  
1082  
1083 protected HashEntry(final HashEntry<K, V> next, final int hashCode, final Object key, final V value) {  
1084 super();  
1085 this.next = next;  
1086 this.hashCode = hashCode;  
1087 this.key = key;  
1088 this.value = value;  
1089 }  
1090  
1091 @Override  
1092 @SuppressWarnings("unchecked")  
1093 public K getKey() {  
1094 if (key == NULL) {  
1095 return null;  
1096 }  
1097 return (K) key;  
1098 }  
1099  
1100 @Override  
1101 @SuppressWarnings("unchecked")  
1102 public V getValue() {  
1103 return (V) value;  
1104 }  
1105  
1106 @Override  
1107 @SuppressWarnings("unchecked")  
1108 public V setValue(final V value) {  
1109 final Object old = this.value;  
1110 this.value = value;  
1111 return (V) old;  
1112 }  
1113  
1114 @Override  
1115 public boolean equals(final Object obj) {  
1116 if (obj == this) {  
1117 return true;  
1118 }  
1119 if (obj instanceof Map.Entry == false) {  
1120 return false;  
1121 }  
1122 final Map.Entry<?, ?> other = (Map.Entry<?, ?>) obj;  
1123 return  
1124 (getKey() == null ? other.getKey() == null : getKey().equals(other.getKey())) &&  
1125 (getValue() == null ? other.getValue() == null : getValue().equals(other.getValue()));  
1126 }  
1127  
1128 @Override  
1129 public int hashCode() {  
1130 return (getKey() == null ? 0 : getKey().hashCode()) ^  
1131 (getValue() == null ? 0 : getValue().hashCode());  
1132 }  
1133  
1134 @Override  
1135 public String toString() {  
1136 return new StringBuilder().append(getKey()).append('=').append(getValue()).toString();  
1137 }  
1138 }  
1139  
1140 /\*\*  
1141 \* Base Iterator  
1142 \*/  
1143 protected static abstract class HashIterator<K, V> {  
1144  
1145 /\*\* The parent map \*/  
1146 private final AbstractHashedMap<K, V> parent;  
1147 /\*\* The current index into the array of buckets \*/  
1148 private int hashIndex;  
1149 /\*\* The last returned entry \*/  
1150 private HashEntry<K, V> last;  
1151 /\*\* The next entry \*/  
1152 private HashEntry<K, V> next;  
1153 /\*\* The modification count expected \*/  
1154 private int expectedModCount;  
1155  
1156 protected HashIterator(final AbstractHashedMap<K, V> parent) {  
1157 super();  
1158 this.parent = parent;  
1159 final HashEntry<K, V>[] data = parent.data;  
1160 int i = data.length;  
1161 HashEntry<K, V> next = null;  
1162 while (i > 0 && next == null) {  
1163 next = data[--i];  
1164 }  
1165 this.next = next;  
1166 this.hashIndex = i;  
1167 this.expectedModCount = parent.modCount;  
1168 }  
1169  
1170 public boolean hasNext() {  
1171 return next != null;  
1172 }  
1173  
1174 protected HashEntry<K, V> nextEntry() {  
1175 if (parent.modCount != expectedModCount) {  
1176 throw new ConcurrentModificationException();  
1177 }  
1178 final HashEntry<K, V> newCurrent = next;  
1179 if (newCurrent == null) {  
1180 throw new NoSuchElementException(AbstractHashedMap.NO\_NEXT\_ENTRY);  
1181 }  
1182 final HashEntry<K, V>[] data = parent.data;  
1183 int i = hashIndex;  
1184 HashEntry<K, V> n = newCurrent.next;  
1185 while (n == null && i > 0) {  
1186 n = data[--i];  
1187 }  
1188 next = n;  
1189 hashIndex = i;  
1190 last = newCurrent;  
1191 return newCurrent;  
1192 }  
1193  
1194 protected HashEntry<K, V> currentEntry() {  
1195 return last;  
1196 }  
1197  
1198 public void remove() {  
1199 if (last == null) {  
1200 throw new IllegalStateException(AbstractHashedMap.REMOVE\_INVALID);  
1201 }  
1202 if (parent.modCount != expectedModCount) {  
1203 throw new ConcurrentModificationException();  
1204 }  
1205 parent.remove(last.getKey());  
1206 last = null;  
1207 expectedModCount = parent.modCount;  
1208 }  
1209  
1210 @Override  
1211 public String toString() {  
1212 if (last != null) {  
1213 return "Iterator[" + last.getKey() + "=" + last.getValue() + "]";  
1214 }  
1215 return "Iterator[]";  
1216 }  
1217 }  
1218  
1219 //-----------------------------------------------------------------------  
1220 /\*\*  
1221 \* Writes the map data to the stream. This method must be overridden if a  
1222 \* subclass must be setup before <code>put()</code> is used.  
1223 \* <p>  
1224 \* Serialization is not one of the JDK's nicest topics. Normal serialization will  
1225 \* initialise the superclass before the subclass. Sometimes however, this isn't  
1226 \* what you want, as in this case the <code>put()</code> method on read can be  
1227 \* affected by subclass state.  
1228 \* <p>  
1229 \* The solution adopted here is to serialize the state data of this class in  
1230 \* this protected method. This method must be called by the  
1231 \* <code>writeObject()</code> of the first serializable subclass.  
1232 \* <p>  
1233 \* Subclasses may override if they have a specific field that must be present  
1234 \* on read before this implementation will work. Generally, the read determines  
1235 \* what must be serialized here, if anything.  
1236 \*  
1237 \* @param out the output stream  
1238 \* @throws IOException if an error occurs while writing tothe stream  
1239 \*/  
1240 protected void doWriteObject(final ObjectOutputStream out) throws IOException {  
1241 out.writeFloat(loadFactor);  
1242 out.writeInt(data.length);  
1243 out.writeInt(size);  
1244 for (final MapIterator<K, V> it = mapIterator(); it.hasNext();) {  
1245 out.writeObject(it.next());  
1246 out.writeObject(it.getValue());  
1247 }  
1248 }  
1249  
1250 /\*\*  
1251 \* Reads the map data from the stream. This method must be overridden if a  
1252 \* subclass must be setup before <code>put()</code> is used.  
1253 \* <p>  
1254 \* Serialization is not one of the JDK's nicest topics. Normal serialization will  
1255 \* initialise the superclass before the subclass. Sometimes however, this isn't  
1256 \* what you want, as in this case the <code>put()</code> method on read can be  
1257 \* affected by subclass state.  
1258 \* <p>  
1259 \* The solution adopted here is to deserialize the state data of this class in  
1260 \* this protected method. This method must be called by the  
1261 \* <code>readObject()</code> of the first serializable subclass.  
1262 \* <p>  
1263 \* Subclasses may override if the subclass has a specific field that must be present  
1264 \* before <code>put()</code> or <code>calculateThreshold()</code> will work correctly.  
1265 \*  
1266 \* @param in the input stream  
1267 \* @throws IOException if an error occurs while reading from the stream  
1268 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
1269 \*/  
1270 @SuppressWarnings("unchecked")  
1271 protected void doReadObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
1272 loadFactor = in.readFloat();  
1273 final int capacity = in.readInt();  
1274 final int size = in.readInt();  
1275 init();  
1276 threshold = calculateThreshold(capacity, loadFactor);  
1277 data = new HashEntry[capacity];  
1278 for (int i = 0; i < size; i++) {  
1279 final K key = (K) in.readObject();  
1280 final V value = (V) in.readObject();  
1281 put(key, value);  
1282 }  
1283 }  
1284  
1285 //-----------------------------------------------------------------------  
1286 /\*\*  
1287 \* Clones the map without cloning the keys or values.  
1288 \* <p>  
1289 \* To implement <code>clone()</code>, a subclass must implement the  
1290 \* <code>Cloneable</code> interface and make this method public.  
1291 \*  
1292 \* @return a shallow clone  
1293 \* @throws InternalError if {@link AbstractMap#clone()} failed  
1294 \*/  
1295 @Override  
1296 @SuppressWarnings("unchecked")  
1297 protected AbstractHashedMap<K, V> clone() {  
1298 try {  
1299 final AbstractHashedMap<K, V> cloned = (AbstractHashedMap<K, V>) super.clone();  
1300 cloned.data = new HashEntry[data.length];  
1301 cloned.entrySet = null;  
1302 cloned.keySet = null;  
1303 cloned.values = null;  
1304 cloned.modCount = 0;  
1305 cloned.size = 0;  
1306 cloned.init();  
1307 cloned.putAll(this);  
1308 return cloned;  
1309 } catch (final CloneNotSupportedException ex) {  
1310 throw new InternalError();  
1311 }  
1312 }  
1313  
1314 /\*\*  
1315 \* Compares this map with another.  
1316 \*  
1317 \* @param obj the object to compare to  
1318 \* @return true if equal  
1319 \*/  
1320 @Override  
1321 public boolean equals(final Object obj) {  
1322 if (obj == this) {  
1323 return true;  
1324 }  
1325 if (obj instanceof Map == false) {  
1326 return false;  
1327 }  
1328 final Map<?,?> map = (Map<?,?>) obj;  
1329 if (map.size() != size()) {  
1330 return false;  
1331 }  
1332 final MapIterator<?,?> it = mapIterator();  
1333 try {  
1334 while (it.hasNext()) {  
1335 final Object key = it.next();  
1336 final Object value = it.getValue();  
1337 if (value == null) {  
1338 if (map.get(key) != null || map.containsKey(key) == false) {  
1339 return false;  
1340 }  
1341 } else {  
1342 if (value.equals(map.get(key)) == false) {  
1343 return false;  
1344 }  
1345 }  
1346 }  
1347 } catch (final ClassCastException ignored) {  
1348 return false;  
1349 } catch (final NullPointerException ignored) {  
1350 return false;  
1351 }  
1352 return true;  
1353 }  
1354  
1355 /\*\*  
1356 \* Gets the standard Map hashCode.  
1357 \*  
1358 \* @return the hash code defined in the Map interface  
1359 \*/  
1360 @Override  
1361 public int hashCode() {  
1362 int total = 0;  
1363 final Iterator<Map.Entry<K, V>> it = createEntrySetIterator();  
1364 while (it.hasNext()) {  
1365 total += it.next().hashCode();  
1366 }  
1367 return total;  
1368 }  
1369  
1370 /\*\*  
1371 \* Gets the map as a String.  
1372 \*  
1373 \* @return a string version of the map  
1374 \*/  
1375 @Override  
1376 public String toString() {  
1377 if (size() == 0) {  
1378 return "{}";  
1379 }  
1380 final StringBuilder buf = new StringBuilder(32 \* size());  
1381 buf.append('{');  
1382  
1383 final MapIterator<K, V> it = mapIterator();  
1384 boolean hasNext = it.hasNext();  
1385 while (hasNext) {  
1386 final K key = it.next();  
1387 final V value = it.getValue();  
1388 buf.append(key == this ? "(this Map)" : key)  
1389 .append('=')  
1390 .append(value == this ? "(this Map)" : value);  
1391  
1392 hasNext = it.hasNext();  
1393 if (hasNext) {  
1394 buf.append(',').append(' ');  
1395 }  
1396 }  
1397  
1398 buf.append('}');  
1399 return buf.toString();  
1400 }  
1401}