001/\*  
002 \* Licensed to the Apache Software Foundation (ASF) under one or more  
003 \* contributor license agreements. See the NOTICE file distributed with  
004 \* this work for additional information regarding copyright ownership.  
005 \* The ASF licenses this file to You under the Apache License, Version 2.0  
006 \* (the "License"); you may not use this file except in compliance with  
007 \* the License. You may obtain a copy of the License at  
008 \*  
009 \* http://www.apache.org/licenses/LICENSE-2.0  
010 \*  
011 \* Unless required by applicable law or agreed to in writing, software  
012 \* distributed under the License is distributed on an "AS IS" BASIS,  
013 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
014 \* See the License for the specific language governing permissions and  
015 \* limitations under the License.  
016 \*/  
017package org.apache.commons.collections4.map;  
018  
019import java.io.IOException;  
020import java.io.ObjectInputStream;  
021import java.io.ObjectOutputStream;  
022import java.io.Serializable;  
023import java.util.Map;  
024  
025import org.apache.commons.collections4.BoundedMap;  
026  
027/\*\*  
028 \* A <code>Map</code> implementation with a fixed maximum size which removes  
029 \* the least recently used entry if an entry is added when full.  
030 \* <p>  
031 \* The least recently used algorithm works on the get and put operations only.  
032 \* Iteration of any kind, including setting the value by iteration, does not  
033 \* change the order. Queries such as containsKey and containsValue or access  
034 \* via views also do not change the order.  
035 \* </p>  
036 \* <p>  
037 \* A somewhat subtle ramification of the least recently used  
038 \* algorithm is that calls to {@link #get(Object)} stand a very good chance  
039 \* of modifying the map's iteration order and thus invalidating any  
040 \* iterators currently in use. It is therefore suggested that iterations  
041 \* over an {@link LRUMap} instance access entry values only through a  
042 \* {@link org.apache.commons.collections4.MapIterator MapIterator} or {@link #entrySet()} iterator.  
043 \* </p>  
044 \* <p>  
045 \* The map implements <code>OrderedMap</code> and entries may be queried using  
046 \* the bidirectional <code>OrderedMapIterator</code>. The order returned is  
047 \* least recently used to most recently used. Iterators from map views can  
048 \* also be cast to <code>OrderedIterator</code> if required.  
049 \* </p>  
050 \* <p>  
051 \* All the available iterators can be reset back to the start by casting to  
052 \* <code>ResettableIterator</code> and calling <code>reset()</code>.  
053 \* </p>  
054 \* <p>  
055 \* <strong>Note that LRUMap is not synchronized and is not thread-safe.</strong>  
056 \* If you wish to use this map from multiple threads concurrently, you must use  
057 \* appropriate synchronization. The simplest approach is to wrap this map  
058 \* using {@link java.util.Collections#synchronizedMap(Map)}. This class may throw  
059 \* <code>NullPointerException</code>'s when accessed by concurrent threads.  
060 \* </p>  
061 \*  
062 \* @param <K> the type of the keys in this map  
063 \* @param <V> the type of the values in this map  
064 \* @since 3.0 (previously in main package v1.0)  
065 \*/  
066public class LRUMap<K, V>  
067 extends AbstractLinkedMap<K, V> implements BoundedMap<K, V>, Serializable, Cloneable {  
068  
069 /\*\* Serialisation version \*/  
070 private static final long serialVersionUID = -612114643488955218L;  
071 /\*\* Default maximum size \*/  
072 protected static final int DEFAULT\_MAX\_SIZE = 100;  
073  
074 /\*\* Maximum size \*/  
075 private transient int maxSize;  
076 /\*\* Scan behaviour \*/  
077 private boolean scanUntilRemovable;  
078  
079 /\*\*  
080 \* Constructs a new empty map with a maximum size of 100.  
081 \*/  
082 public LRUMap() {  
083 this(DEFAULT\_MAX\_SIZE, DEFAULT\_LOAD\_FACTOR, false);  
084 }  
085  
086 /\*\*  
087 \* Constructs a new, empty map with the specified maximum size.  
088 \*  
089 \* @param maxSize the maximum size of the map  
090 \* @throws IllegalArgumentException if the maximum size is less than one  
091 \*/  
092 public LRUMap(final int maxSize) {  
093 this(maxSize, DEFAULT\_LOAD\_FACTOR);  
094 }  
095  
096 /\*\*  
097 \* Constructs a new, empty map with the specified maximum size.  
098 \*  
099 \* @param maxSize the maximum size of the map  
100 \* @param initialSize the initial size of the map  
101 \* @throws IllegalArgumentException if the maximum size is less than one  
102 \* @throws IllegalArgumentException if the initial size is negative or larger than the maximum size  
103 \* @since 4.1  
104 \*/  
105 public LRUMap(final int maxSize, final int initialSize) {  
106 this(maxSize, initialSize, DEFAULT\_LOAD\_FACTOR);  
107 }  
108  
109 /\*\*  
110 \* Constructs a new, empty map with the specified maximum size.  
111 \*  
112 \* @param maxSize the maximum size of the map  
113 \* @param scanUntilRemovable scan until a removeable entry is found, default false  
114 \* @throws IllegalArgumentException if the maximum size is less than one  
115 \* @since 3.1  
116 \*/  
117 public LRUMap(final int maxSize, final boolean scanUntilRemovable) {  
118 this(maxSize, DEFAULT\_LOAD\_FACTOR, scanUntilRemovable);  
119 }  
120  
121 /\*\*  
122 \* Constructs a new, empty map with the specified max capacity and  
123 \* load factor.  
124 \*  
125 \* @param maxSize the maximum size of the map  
126 \* @param loadFactor the load factor  
127 \* @throws IllegalArgumentException if the maximum size is less than one  
128 \* @throws IllegalArgumentException if the load factor is less than zero  
129 \*/  
130 public LRUMap(final int maxSize, final float loadFactor) {  
131 this(maxSize, loadFactor, false);  
132 }  
133  
134 /\*\*  
135 \* Constructs a new, empty map with the specified max / initial capacity and  
136 \* load factor.  
137 \*  
138 \* @param maxSize the maximum size of the map  
139 \* @param initialSize the initial size of the map  
140 \* @param loadFactor the load factor  
141 \* @throws IllegalArgumentException if the maximum size is less than one  
142 \* @throws IllegalArgumentException if the initial size is negative or larger than the maximum size  
143 \* @throws IllegalArgumentException if the load factor is less than zero  
144 \* @since 4.1  
145 \*/  
146 public LRUMap(final int maxSize, final int initialSize, final float loadFactor) {  
147 this(maxSize, initialSize, loadFactor, false);  
148 }  
149  
150 /\*\*  
151 \* Constructs a new, empty map with the specified max capacity and load factor.  
152 \*  
153 \* @param maxSize the maximum size of the map  
154 \* @param loadFactor the load factor  
155 \* @param scanUntilRemovable scan until a removeable entry is found, default false  
156 \* @throws IllegalArgumentException if the maximum size is less than one  
157 \* @throws IllegalArgumentException if the load factor is less than zero  
158 \* @since 3.1  
159 \*/  
160 public LRUMap(final int maxSize, final float loadFactor, final boolean scanUntilRemovable) {  
161 this(maxSize, maxSize, loadFactor, scanUntilRemovable);  
162 }  
163  
164 /\*\*  
165 \* Constructs a new, empty map with the specified max / initial capacity and load factor.  
166 \*  
167 \* @param maxSize the maximum size of the map  
168 \* @param initialSize the initial size of the map  
169 \* @param loadFactor the load factor  
170 \* @param scanUntilRemovable scan until a removeable entry is found, default false  
171 \* @throws IllegalArgumentException if the maximum size is less than one  
172 \* @throws IllegalArgumentException if the initial size is negative or larger than the maximum size  
173 \* @throws IllegalArgumentException if the load factor is less than zero  
174 \* @since 4.1  
175 \*/  
176 public LRUMap(final int maxSize,  
177 final int initialSize,  
178 final float loadFactor,  
179 final boolean scanUntilRemovable) {  
180  
181 super(initialSize, loadFactor);  
182 if (maxSize < 1) {  
183 throw new IllegalArgumentException("LRUMap max size must be greater than 0");  
184 }  
185 if (initialSize > maxSize) {  
186 throw new IllegalArgumentException("LRUMap initial size must not be greather than max size");  
187 }  
188 this.maxSize = maxSize;  
189 this.scanUntilRemovable = scanUntilRemovable;  
190 }  
191  
192 /\*\*  
193 \* Constructor copying elements from another map.  
194 \* <p>  
195 \* The maximum size is set from the map's size.  
196 \*  
197 \* @param map the map to copy  
198 \* @throws NullPointerException if the map is null  
199 \* @throws IllegalArgumentException if the map is empty  
200 \*/  
201 public LRUMap(final Map<? extends K, ? extends V> map) {  
202 this(map, false);  
203 }  
204  
205 /\*\*  
206 \* Constructor copying elements from another map.  
207 \*  
208 \* <p>The maximum size is set from the map's size.</p>  
209 \*  
210 \* @param map the map to copy  
211 \* @param scanUntilRemovable scan until a removeable entry is found, default false  
212 \* @throws NullPointerException if the map is null  
213 \* @throws IllegalArgumentException if the map is empty  
214 \* @since 3.1  
215 \*/  
216 public LRUMap(final Map<? extends K, ? extends V> map, final boolean scanUntilRemovable) {  
217 this(map.size(), DEFAULT\_LOAD\_FACTOR, scanUntilRemovable);  
218 putAll(map);  
219 }  
220  
221 //-----------------------------------------------------------------------  
222 /\*\*  
223 \* Gets the value mapped to the key specified.  
224 \* <p>  
225 \* This operation changes the position of the key in the map to the  
226 \* most recently used position (last).  
227 \*  
228 \* @param key the key  
229 \* @return the mapped value, null if no match  
230 \*/  
231 @Override  
232 public V get(final Object key) {  
233 return get(key, true);  
234 }  
235  
236 /\*\*  
237 \* Gets the value mapped to the key specified.  
238 \* <p>  
239 \* If {@code updateToMRU} is {@code true}, the position of the key in the map  
240 \* is changed to the most recently used position (last), otherwise the iteration  
241 \* order is not changed by this operation.  
242 \*  
243 \* @param key the key  
244 \* @param updateToMRU whether the key shall be updated to the  
245 \* most recently used position  
246 \* @return the mapped value, null if no match  
247 \* @since 4.1  
248 \*/  
249 public V get(final Object key, final boolean updateToMRU) {  
250 final LinkEntry<K, V> entry = getEntry(key);  
251 if (entry == null) {  
252 return null;  
253 }  
254 if (updateToMRU) {  
255 moveToMRU(entry);  
256 }  
257 return entry.getValue();  
258 }  
259  
260 //-----------------------------------------------------------------------  
261 /\*\*  
262 \* Moves an entry to the MRU position at the end of the list.  
263 \* <p>  
264 \* This implementation moves the updated entry to the end of the list.  
265 \*  
266 \* @param entry the entry to update  
267 \*/  
268 protected void moveToMRU(final LinkEntry<K, V> entry) {  
269 if (entry.after != header) {  
270 modCount++;  
271 // remove  
272 if(entry.before == null) {  
273 throw new IllegalStateException("Entry.before is null." +  
274 " This should not occur if your keys are immutable, and you have used synchronization properly.");  
275 }  
276 entry.before.after = entry.after;  
277 entry.after.before = entry.before;  
278 // add first  
279 entry.after = header;  
280 entry.before = header.before;  
281 header.before.after = entry;  
282 header.before = entry;  
283 } else if (entry == header) {  
284 throw new IllegalStateException("Can't move header to MRU" +  
285 " This should not occur if your keys are immutable, and you have used synchronization properly.");  
286 }  
287 }  
288  
289 /\*\*  
290 \* Updates an existing key-value mapping.  
291 \* <p>  
292 \* This implementation moves the updated entry to the end of the list  
293 \* using {@link #moveToMRU(AbstractLinkedMap.LinkEntry)}.  
294 \*  
295 \* @param entry the entry to update  
296 \* @param newValue the new value to store  
297 \*/  
298 @Override  
299 protected void updateEntry(final HashEntry<K, V> entry, final V newValue) {  
300 moveToMRU((LinkEntry<K, V>) entry); // handles modCount  
301 entry.setValue(newValue);  
302 }  
303  
304 /\*\*  
305 \* Adds a new key-value mapping into this map.  
306 \* <p>  
307 \* This implementation checks the LRU size and determines whether to  
308 \* discard an entry or not using {@link #removeLRU(AbstractLinkedMap.LinkEntry)}.  
309 \* <p>  
310 \* From Commons Collections 3.1 this method uses {@link #isFull()} rather  
311 \* than accessing <code>size</code> and <code>maxSize</code> directly.  
312 \* It also handles the scanUntilRemovable functionality.  
313 \*  
314 \* @param hashIndex the index into the data array to store at  
315 \* @param hashCode the hash code of the key to add  
316 \* @param key the key to add  
317 \* @param value the value to add  
318 \*/  
319 @Override  
320 protected void addMapping(final int hashIndex, final int hashCode, final K key, final V value) {  
321 if (isFull()) {  
322 LinkEntry<K, V> reuse = header.after;  
323 boolean removeLRUEntry = false;  
324 if (scanUntilRemovable) {  
325 while (reuse != header && reuse != null) {  
326 if (removeLRU(reuse)) {  
327 removeLRUEntry = true;  
328 break;  
329 }  
330 reuse = reuse.after;  
331 }  
332 if (reuse == null) {  
333 throw new IllegalStateException(  
334 "Entry.after=null, header.after=" + header.after + " header.before=" + header.before +  
335 " key=" + key + " value=" + value + " size=" + size + " maxSize=" + maxSize +  
336 " This should not occur if your keys are immutable, and you have used synchronization properly.");  
337 }  
338 } else {  
339 removeLRUEntry = removeLRU(reuse);  
340 }  
341  
342 if (removeLRUEntry) {  
343 if (reuse == null) {  
344 throw new IllegalStateException(  
345 "reuse=null, header.after=" + header.after + " header.before=" + header.before +  
346 " key=" + key + " value=" + value + " size=" + size + " maxSize=" + maxSize +  
347 " This should not occur if your keys are immutable, and you have used synchronization properly.");  
348 }  
349 reuseMapping(reuse, hashIndex, hashCode, key, value);  
350 } else {  
351 super.addMapping(hashIndex, hashCode, key, value);  
352 }  
353 } else {  
354 super.addMapping(hashIndex, hashCode, key, value);  
355 }  
356 }  
357  
358 /\*\*  
359 \* Reuses an entry by removing it and moving it to a new place in the map.  
360 \* <p>  
361 \* This method uses {@link #removeEntry}, {@link #reuseEntry} and {@link #addEntry}.  
362 \*  
363 \* @param entry the entry to reuse  
364 \* @param hashIndex the index into the data array to store at  
365 \* @param hashCode the hash code of the key to add  
366 \* @param key the key to add  
367 \* @param value the value to add  
368 \*/  
369 protected void reuseMapping(final LinkEntry<K, V> entry, final int hashIndex, final int hashCode,  
370 final K key, final V value) {  
371 // find the entry before the entry specified in the hash table  
372 // remember that the parameters (except the first) refer to the new entry,  
373 // not the old one  
374 try {  
375 final int removeIndex = hashIndex(entry.hashCode, data.length);  
376 final HashEntry<K, V>[] tmp = data; // may protect against some sync issues  
377 HashEntry<K, V> loop = tmp[removeIndex];  
378 HashEntry<K, V> previous = null;  
379 while (loop != entry && loop != null) {  
380 previous = loop;  
381 loop = loop.next;  
382 }  
383 if (loop == null) {  
384 throw new IllegalStateException(  
385 "Entry.next=null, data[removeIndex]=" + data[removeIndex] + " previous=" + previous +  
386 " key=" + key + " value=" + value + " size=" + size + " maxSize=" + maxSize +  
387 " This should not occur if your keys are immutable, and you have used synchronization properly.");  
388 }  
389  
390 // reuse the entry  
391 modCount++;  
392 removeEntry(entry, removeIndex, previous);  
393 reuseEntry(entry, hashIndex, hashCode, key, value);  
394 addEntry(entry, hashIndex);  
395 } catch (final NullPointerException ex) {  
396 throw new IllegalStateException(  
397 "NPE, entry=" + entry + " entryIsHeader=" + (entry==header) +  
398 " key=" + key + " value=" + value + " size=" + size + " maxSize=" + maxSize +  
399 " This should not occur if your keys are immutable, and you have used synchronization properly.");  
400 }  
401 }  
402  
403 /\*\*  
404 \* Subclass method to control removal of the least recently used entry from the map.  
405 \* <p>  
406 \* This method exists for subclasses to override. A subclass may wish to  
407 \* provide cleanup of resources when an entry is removed. For example:  
408 \* <pre>  
409 \* protected boolean removeLRU(LinkEntry entry) {  
410 \* releaseResources(entry.getValue()); // release resources held by entry  
411 \* return true; // actually delete entry  
412 \* }  
413 \* </pre>  
414 \* <p>  
415 \* Alternatively, a subclass may choose to not remove the entry or selectively  
416 \* keep certain LRU entries. For example:  
417 \* <pre>  
418 \* protected boolean removeLRU(LinkEntry entry) {  
419 \* if (entry.getKey().toString().startsWith("System.")) {  
420 \* return false; // entry not removed from LRUMap  
421 \* } else {  
422 \* return true; // actually delete entry  
423 \* }  
424 \* }  
425 \* </pre>  
426 \* The effect of returning false is dependent on the scanUntilRemovable flag.  
427 \* If the flag is true, the next LRU entry will be passed to this method and so on  
428 \* until one returns false and is removed, or every entry in the map has been passed.  
429 \* If the scanUntilRemovable flag is false, the map will exceed the maximum size.  
430 \* <p>  
431 \* NOTE: Commons Collections 3.0 passed the wrong entry to this method.  
432 \* This is fixed in version 3.1 onwards.  
433 \*  
434 \* @param entry the entry to be removed  
435 \* @return {@code true}  
436 \*/  
437 protected boolean removeLRU(final LinkEntry<K, V> entry) {  
438 return true;  
439 }  
440  
441 //-----------------------------------------------------------------------  
442 /\*\*  
443 \* Returns true if this map is full and no new mappings can be added.  
444 \*  
445 \* @return <code>true</code> if the map is full  
446 \*/  
447 @Override  
448 public boolean isFull() {  
449 return size >= maxSize;  
450 }  
451  
452 /\*\*  
453 \* Gets the maximum size of the map (the bound).  
454 \*  
455 \* @return the maximum number of elements the map can hold  
456 \*/  
457 @Override  
458 public int maxSize() {  
459 return maxSize;  
460 }  
461  
462 /\*\*  
463 \* Whether this LRUMap will scan until a removable entry is found when the  
464 \* map is full.  
465 \*  
466 \* @return true if this map scans  
467 \* @since 3.1  
468 \*/  
469 public boolean isScanUntilRemovable() {  
470 return scanUntilRemovable;  
471 }  
472  
473 //-----------------------------------------------------------------------  
474 /\*\*  
475 \* Clones the map without cloning the keys or values.  
476 \*  
477 \* @return a shallow clone  
478 \*/  
479 @Override  
480 public LRUMap<K, V> clone() {  
481 return (LRUMap<K, V>) super.clone();  
482 }  
483  
484 /\*\*  
485 \* Write the map out using a custom routine.  
486 \*  
487 \* @param out the output stream  
488 \* @throws IOException if an error occurs while writing to the stream  
489 \*/  
490 private void writeObject(final ObjectOutputStream out) throws IOException {  
491 out.defaultWriteObject();  
492 doWriteObject(out);  
493 }  
494  
495 /\*\*  
496 \* Read the map in using a custom routine.  
497 \*  
498 \* @param in the input stream  
499 \* @throws IOException if an error occurs while reading from the stream  
500 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
501 \*/  
502 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
503 in.defaultReadObject();  
504 doReadObject(in);  
505 }  
506  
507 /\*\*  
508 \* Writes the data necessary for <code>put()</code> to work in deserialization.  
509 \*  
510 \* @param out the output stream  
511 \* @throws IOException if an error occurs while writing to the stream  
512 \*/  
513 @Override  
514 protected void doWriteObject(final ObjectOutputStream out) throws IOException {  
515 out.writeInt(maxSize);  
516 super.doWriteObject(out);  
517 }  
518  
519 /\*\*  
520 \* Reads the data necessary for <code>put()</code> to work in the superclass.  
521 \*  
522 \* @param in the input stream  
523 \* @throws IOException if an error occurs while reading from the stream  
524 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
525 \*/  
526 @Override  
527 protected void doReadObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
528 maxSize = in.readInt();  
529 super.doReadObject(in);  
530 }  
531  
532}