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.queue;  
018  
019import java.io.IOException;  
020import java.io.ObjectInputStream;  
021import java.io.ObjectOutputStream;  
022import java.io.Serializable;  
023import java.util.AbstractCollection;  
024import java.util.Arrays;  
025import java.util.Collection;  
026import java.util.Iterator;  
027import java.util.NoSuchElementException;  
028import java.util.Queue;  
029  
030import org.apache.commons.collections4.BoundedCollection;  
031  
032/\*\*  
033 \* CircularFifoQueue is a first-in first-out queue with a fixed size that  
034 \* replaces its oldest element if full.  
035 \* <p>  
036 \* The removal order of a {@link CircularFifoQueue} is based on the  
037 \* insertion order; elements are removed in the same order in which they  
038 \* were added. The iteration order is the same as the removal order.  
039 \* </p>  
040 \* <p>  
041 \* The {@link #add(Object)}, {@link #remove()}, {@link #peek()}, {@link #poll},  
042 \* {@link #offer(Object)} operations all perform in constant time.  
043 \* All other operations perform in linear time or worse.  
044 \* </p>  
045 \* <p>  
046 \* This queue prevents null objects from being added.  
047 \* </p>  
048 \*  
049 \* @param <E> the type of elements in this collection  
050 \* @since 4.0  
051 \*/  
052public class CircularFifoQueue<E> extends AbstractCollection<E>  
053 implements Queue<E>, BoundedCollection<E>, Serializable {  
054  
055 /\*\* Serialization version. \*/  
056 private static final long serialVersionUID = -8423413834657610406L;  
057  
058 /\*\* Underlying storage array. \*/  
059 private transient E[] elements;  
060  
061 /\*\* Array index of first (oldest) queue element. \*/  
062 private transient int start = 0;  
063  
064 /\*\*  
065 \* Index mod maxElements of the array position following the last queue  
066 \* element. Queue elements start at elements[start] and "wrap around"  
067 \* elements[maxElements-1], ending at elements[decrement(end)].  
068 \* For example, elements = {c,a,b}, start=1, end=1 corresponds to  
069 \* the queue [a,b,c].  
070 \*/  
071 private transient int end = 0;  
072  
073 /\*\* Flag to indicate if the queue is currently full. \*/  
074 private transient boolean full = false;  
075  
076 /\*\* Capacity of the queue. \*/  
077 private final int maxElements;  
078  
079 /\*\*  
080 \* Constructor that creates a queue with the default size of 32.  
081 \*/  
082 public CircularFifoQueue() {  
083 this(32);  
084 }  
085  
086 /\*\*  
087 \* Constructor that creates a queue with the specified size.  
088 \*  
089 \* @param size the size of the queue (cannot be changed)  
090 \* @throws IllegalArgumentException if the size is < 1  
091 \*/  
092 @SuppressWarnings("unchecked")  
093 public CircularFifoQueue(final int size) {  
094 if (size <= 0) {  
095 throw new IllegalArgumentException("The size must be greater than 0");  
096 }  
097 elements = (E[]) new Object[size];  
098 maxElements = elements.length;  
099 }  
100  
101 /\*\*  
102 \* Constructor that creates a queue from the specified collection.  
103 \* The collection size also sets the queue size.  
104 \*  
105 \* @param coll the collection to copy into the queue, may not be null  
106 \* @throws NullPointerException if the collection is null  
107 \*/  
108 public CircularFifoQueue(final Collection<? extends E> coll) {  
109 this(coll.size());  
110 addAll(coll);  
111 }  
112  
113 //-----------------------------------------------------------------------  
114 /\*\*  
115 \* Write the queue out using a custom routine.  
116 \*  
117 \* @param out the output stream  
118 \* @throws IOException if an I/O error occurs while writing to the output stream  
119 \*/  
120 private void writeObject(final ObjectOutputStream out) throws IOException {  
121 out.defaultWriteObject();  
122 out.writeInt(size());  
123 for (final E e : this) {  
124 out.writeObject(e);  
125 }  
126 }  
127  
128 /\*\*  
129 \* Read the queue in using a custom routine.  
130 \*  
131 \* @param in the input stream  
132 \* @throws IOException if an I/O error occurs while writing to the output stream  
133 \* @throws ClassNotFoundException if the class of a serialized object can not be found  
134 \*/  
135 @SuppressWarnings("unchecked")  
136 private void readObject(final ObjectInputStream in) throws IOException, ClassNotFoundException {  
137 in.defaultReadObject();  
138 elements = (E[]) new Object[maxElements];  
139 final int size = in.readInt();  
140 for (int i = 0; i < size; i++) {  
141 elements[i] = (E) in.readObject();  
142 }  
143 start = 0;  
144 full = size == maxElements;  
145 if (full) {  
146 end = 0;  
147 } else {  
148 end = size;  
149 }  
150 }  
151  
152 //-----------------------------------------------------------------------  
153 /\*\*  
154 \* Returns the number of elements stored in the queue.  
155 \*  
156 \* @return this queue's size  
157 \*/  
158 @Override  
159 public int size() {  
160 int size = 0;  
161  
162 if (end < start) {  
163 size = maxElements - start + end;  
164 } else if (end == start) {  
165 size = full ? maxElements : 0;  
166 } else {  
167 size = end - start;  
168 }  
169  
170 return size;  
171 }  
172  
173 /\*\*  
174 \* Returns true if this queue is empty; false otherwise.  
175 \*  
176 \* @return true if this queue is empty  
177 \*/  
178 @Override  
179 public boolean isEmpty() {  
180 return size() == 0;  
181 }  
182  
183 /\*\*  
184 \* {@inheritDoc}  
185 \* <p>  
186 \* A {@code CircularFifoQueue} can never be full, thus this returns always  
187 \* {@code false}.  
188 \*  
189 \* @return always returns {@code false}  
190 \*/  
191 @Override  
192 public boolean isFull() {  
193 return false;  
194 }  
195  
196 /\*\*  
197 \* Returns {@code true} if the capacity limit of this queue has been reached,  
198 \* i.e. the number of elements stored in the queue equals its maximum size.  
199 \*  
200 \* @return {@code true} if the capacity limit has been reached, {@code false} otherwise  
201 \* @since 4.1  
202 \*/  
203 public boolean isAtFullCapacity() {  
204 return size() == maxElements;  
205 }  
206  
207 /\*\*  
208 \* Gets the maximum size of the collection (the bound).  
209 \*  
210 \* @return the maximum number of elements the collection can hold  
211 \*/  
212 @Override  
213 public int maxSize() {  
214 return maxElements;  
215 }  
216  
217 /\*\*  
218 \* Clears this queue.  
219 \*/  
220 @Override  
221 public void clear() {  
222 full = false;  
223 start = 0;  
224 end = 0;  
225 Arrays.fill(elements, null);  
226 }  
227  
228 /\*\*  
229 \* Adds the given element to this queue. If the queue is full, the least recently added  
230 \* element is discarded so that a new element can be inserted.  
231 \*  
232 \* @param element the element to add  
233 \* @return true, always  
234 \* @throws NullPointerException if the given element is null  
235 \*/  
236 @Override  
237 public boolean add(final E element) {  
238 if (null == element) {  
239 throw new NullPointerException("Attempted to add null object to queue");  
240 }  
241  
242 if (isAtFullCapacity()) {  
243 remove();  
244 }  
245  
246 elements[end++] = element;  
247  
248 if (end >= maxElements) {  
249 end = 0;  
250 }  
251  
252 if (end == start) {  
253 full = true;  
254 }  
255  
256 return true;  
257 }  
258  
259 /\*\*  
260 \* Returns the element at the specified position in this queue.  
261 \*  
262 \* @param index the position of the element in the queue  
263 \* @return the element at position {@code index}  
264 \* @throws NoSuchElementException if the requested position is outside the range [0, size)  
265 \*/  
266 public E get(final int index) {  
267 final int sz = size();  
268 if (index < 0 || index >= sz) {  
269 throw new NoSuchElementException(  
270 String.format("The specified index (%1$d) is outside the available range [0, %2$d)",  
271 Integer.valueOf(index), Integer.valueOf(sz)));  
272 }  
273  
274 final int idx = (start + index) % maxElements;  
275 return elements[idx];  
276 }  
277  
278 //-----------------------------------------------------------------------  
279  
280 /\*\*  
281 \* Adds the given element to this queue. If the queue is full, the least recently added  
282 \* element is discarded so that a new element can be inserted.  
283 \*  
284 \* @param element the element to add  
285 \* @return true, always  
286 \* @throws NullPointerException if the given element is null  
287 \*/  
288 @Override  
289 public boolean offer(final E element) {  
290 return add(element);  
291 }  
292  
293 @Override  
294 public E poll() {  
295 if (isEmpty()) {  
296 return null;  
297 }  
298 return remove();  
299 }  
300  
301 @Override  
302 public E element() {  
303 if (isEmpty()) {  
304 throw new NoSuchElementException("queue is empty");  
305 }  
306 return peek();  
307 }  
308  
309 @Override  
310 public E peek() {  
311 if (isEmpty()) {  
312 return null;  
313 }  
314 return elements[start];  
315 }  
316  
317 @Override  
318 public E remove() {  
319 if (isEmpty()) {  
320 throw new NoSuchElementException("queue is empty");  
321 }  
322  
323 final E element = elements[start];  
324 if (null != element) {  
325 elements[start++] = null;  
326  
327 if (start >= maxElements) {  
328 start = 0;  
329 }  
330 full = false;  
331 }  
332 return element;  
333 }  
334  
335 //-----------------------------------------------------------------------  
336 /\*\*  
337 \* Increments the internal index.  
338 \*  
339 \* @param index the index to increment  
340 \* @return the updated index  
341 \*/  
342 private int increment(int index) {  
343 index++;  
344 if (index >= maxElements) {  
345 index = 0;  
346 }  
347 return index;  
348 }  
349  
350 /\*\*  
351 \* Decrements the internal index.  
352 \*  
353 \* @param index the index to decrement  
354 \* @return the updated index  
355 \*/  
356 private int decrement(int index) {  
357 index--;  
358 if (index < 0) {  
359 index = maxElements - 1;  
360 }  
361 return index;  
362 }  
363  
364 /\*\*  
365 \* Returns an iterator over this queue's elements.  
366 \*  
367 \* @return an iterator over this queue's elements  
368 \*/  
369 @Override  
370 public Iterator<E> iterator() {  
371 return new Iterator<E>() {  
372  
373 private int index = start;  
374 private int lastReturnedIndex = -1;  
375 private boolean isFirst = full;  
376  
377 @Override  
378 public boolean hasNext() {  
379 return isFirst || index != end;  
380 }  
381  
382 @Override  
383 public E next() {  
384 if (!hasNext()) {  
385 throw new NoSuchElementException();  
386 }  
387 isFirst = false;  
388 lastReturnedIndex = index;  
389 index = increment(index);  
390 return elements[lastReturnedIndex];  
391 }  
392  
393 @Override  
394 public void remove() {  
395 if (lastReturnedIndex == -1) {  
396 throw new IllegalStateException();  
397 }  
398  
399 // First element can be removed quickly  
400 if (lastReturnedIndex == start) {  
401 CircularFifoQueue.this.remove();  
402 lastReturnedIndex = -1;  
403 return;  
404 }  
405  
406 int pos = lastReturnedIndex + 1;  
407 if (start < lastReturnedIndex && pos < end) {  
408 // shift in one part  
409 System.arraycopy(elements, pos, elements, lastReturnedIndex, end - pos);  
410 } else {  
411 // Other elements require us to shift the subsequent elements  
412 while (pos != end) {  
413 if (pos >= maxElements) {  
414 elements[pos - 1] = elements[0];  
415 pos = 0;  
416 } else {  
417 elements[decrement(pos)] = elements[pos];  
418 pos = increment(pos);  
419 }  
420 }  
421 }  
422  
423 lastReturnedIndex = -1;  
424 end = decrement(end);  
425 elements[end] = null;  
426 full = false;  
427 index = decrement(index);  
428 }  
429  
430 };  
431 }  
432  
433}