

BlockingArrayQueue.java

```
1 //BlockingArrayQueue.java
2 //Template blocking queue class, implemented with emulated circular array.
3 //Programmer: Randy Miller
4 //Last modification: April 13, 2015
5
6
7
8 import java.util.ArrayList;
9 import java.util.Collection;
10 import java.util.Iterator;
11 import java.util.NoSuchElementException;
12 import java.util.concurrent.locks.Lock;
13 import java.util.concurrent.locks.ReentrantLock;
14 import java.util.concurrent.locks.Condition;
15
16 public class BlockingArrayQueue<T> /*implements java.util.Queue<T> (not quite, yet: a few more general methods
17                                     from the Collection superinterface remain unimplemented, and a few methods
18                                     implemented here under another name need to be expressly defined as the
19                                     delegates of required methods, or else renamed)*/
20     implements Cloneable
21 {
22     private boolean empty = true,
23                   full = false;
24     private int capacity, //maximum queue size
25               head = 0, //index of the next element to be dequeued
26               tail = 0; //index of the next element to be enqueued
27     private ArrayList<T> queue;
28     private Lock lock;
29     private Condition queueNotEmpty,
30                   queueNotFull;
31
32     //construction from an initial collection of queue members
33     public BlockingArrayQueue(Collection<? extends T> queue, int capacity)
34     {
35         this.queue = new ArrayList<T>(queue);
36         this.capacity = capacity;
37         if (queue.size() > capacity)
38         {
39             for (int i = capacity; i < queue.size(); i++)
40                 this.queue.remove(i);
41             this.queue.trimToSize();
42         }
43         this.tail = (queue.size() < capacity ? queue.size() : /*capacity*/0);
44         if (queue.size() > 0)
45         {
```

```

46         empty = false;
47         full = (tail == 0 ? true : false);
48     }
49     lock = new ReentrantLock();
50     queueNotEmpty = lock.newCondition();
51     queueNotFull = lock.newCondition();
52 }
53
54 //construction from queue capacity only
55 public BlockingArrayQueue(int capacity)
56 {
57     this.queue = new ArrayList<T>(capacity);
58     this.capacity = capacity;
59     lock = new ReentrantLock();
60     queueNotEmpty = lock.newCondition();
61     queueNotFull = lock.newCondition();
62 }
63
64 //default constructor, with default capacity of 10
65 public BlockingArrayQueue()
66 {
67     this(10);
68 }
69
70 public int getCapacity() { return capacity; }
71
72 public synchronized int getSize() { return (tail != head ? ((tail + capacity - head) % capacity) :
73                                             (full ? capacity : 0)); }
74
75 public synchronized boolean isFull() { return full; }
76
77 public synchronized boolean isEmpty() { return empty; }
78
79 //a method of the java.util.Queue interface
80 public synchronized boolean add(T member)
81 {
82     if (isFull())
83         throw new IllegalStateException();
84     else
85     {
86         enqueue(member);
87         return true;
88     }
89 }
90

```

```
91 //a method of the java.util.Queue interface
92 public synchronized boolean offer(T member)
93 {
94     if (isFull())
95         return false;
96     else
97     {
98         enqueue(member);
99         return true;
100     }
101 }
102
103 //base blocking queue put() implementation
104 public void enqueue(T member)
105 {
106     lock.lock();
107     try
108     {
109         while (isFull())
110         {
111             try
112             {
113                 queueNotFull.await();
114             }
115             catch (InterruptedException exception)
116             {
117                 Thread.currentThread().interrupt();
118             }
119         }
120         if (queue.size() < capacity)
121             queue.add(tail, member);
122         else
123             queue.set(tail, member);
124         tail = (tail + 1) % capacity;
125         if (tail == head) full = true;
126         empty = false;
127         queueNotEmpty.signalAll();
128     }
129     finally
130     {
131         lock.unlock();
132     }
133 }
134
135 //a method of the java.util.Queue interface
```

```
136 public synchronized T element()  
137 {  
138     if (isEmpty())  
139         throw new NoSuchElementException();  
140     else  
141     {  
142         return dequeue(false);  
143     }  
144 }  
145  
146 //a method of the java.util.Queue interface  
147 public synchronized T peek()  
148 {  
149     if (isEmpty())  
150         return null;  
151     else  
152     {  
153         return dequeue(false);  
154     }  
155 }  
156  
157 //function of the java.util.Queue interface  
158 public synchronized T remove()  
159 {  
160     if (isEmpty())  
161         throw new NoSuchElementException();  
162     else  
163     {  
164         return dequeue(true);  
165     }  
166 }  
167  
168 //a method of the java.util.Queue interface  
169 public synchronized T poll()  
170 {  
171     if (isEmpty())  
172         return null;  
173     else  
174     {  
175         return dequeue(true);  
176     }  
177 }  
178  
179 //base blocking queue get() implementation  
180 public T dequeue(boolean notJustPeeking)
```

```

181 {
182     T fifo = null;
183     lock.lock();
184     try
185     {
186         while (isEmpty())
187         {
188             try
189             {
190                 queueNotEmpty.await();
191             }
192             catch (InterruptedException exception)
193             {
194                 Thread.currentThread().interrupt();
195             }
196         }
197         fifo = queue.get(head);
198         if (notJustPeeking)
199         {
200             head = (head + 1) % capacity;
201             if (head == tail) empty = true;
202             full = false;
203             queueNotFull.signalAll();
204         }
205     }
206     finally
207     {
208         lock.unlock();
209     }
210     return fifo;
211 }
212
213 @Override
214 @SuppressWarnings("unchecked")
215 public synchronized BlockingArrayQueue<T> clone()
216 {
217     BlockingArrayQueue<T> clone;
218
219     try
220     {
221         clone = (BlockingArrayQueue<T>)super.clone();
222     }
223     catch (CloneNotSupportedException e) { throw new Error(); } //this won't ever happen
224
225     /*would be better to individually clone() the members of the ArrayList, but Object::clone() is

```

BlockingArrayQueue.java

```

226     a protected method and I can't see how to bring it off.  So if T does not wrap a primitive
227     we will just get a new ArrayList populated with new references to the same old objects.
228     Better than nothing I suppose.*/
229     clone.queue = new ArrayList<T>(this.queue);
230
231     return clone;
232 }
233
234 /*It's not usually a good idea to lock an object for time-consuming IO operations, but
235 I was unable to see how to define a publicly accessible override of the Object::clone()
236 method for the unknown type T that is the class instantiated by members of the queue,
237 and so was unable to create a threadsafe clone of the entire containing queue for IO.
238
239 I did what I could above, and am now limiting this method's lock on the object to its
240 call to this.clone(). At least the efficiency problem is minimized*/
241
242 @Override
243 public String toString()
244 {
245     BlockingArrayQueue<T> clone= this.clone();
246
247     StringBuilder textBuffer = new StringBuilder();
248
249     textBuffer.append(String.format("Queued elements, ordered from the head of the line to the end:%n"));
250
251     if (!isEmpty())
252     {
253         for (int i = 0; i < getSize(); i++)
254         {
255
256             String memberString = (clone.queue).get((head + i) % capacity).toString();
257             if (memberString.length() > 15)
258                 memberString = memberString.substring(0, 15);
259             textBuffer.append(String.format("Element %3d: %-15s%n", i + 1, memberString));
260         }
261     }
262     else
263         textBuffer.append("(queue is empty)\n");
264
265     return textBuffer.toString();
266 }
267
268 //output the members of the queue, oldest first
269 public void print()
270 {

```

```

271     System.out.print(toString());
272 }
273
274 /*The following implementation of an iterator for this queue is not threadsafe or even weakly consistent.
275 (by the time that a (mutable) element of the cloned queue is accessed another thread may have altered it
276 through the use of a different reference.) Where multi-threaded access to the queue is possible the queue
277 must be locked throughout the iteration*/
278
279 public Iterator<T> iterator()
280 {
281     return new BlockingArrayQueueIterator<T>();
282 }
283
284 private class BlockingArrayQueueIterator<E> implements Iterator<E>
285 {
286     int index;
287     boolean firstCycleOfAFullQueueInProgress = false;
288     BlockingArrayQueue<E> clone;
289
290     @SuppressWarnings("unchecked")
291     public BlockingArrayQueueIterator()
292     {
293         clone = (BlockingArrayQueue<E>)BlockingArrayQueue.this.clone();
294         index = clone.head;
295         if (clone.isEmpty())
296             firstCycleOfAFullQueueInProgress = true;
297     }
298
299     public boolean hasNext()
300     {
301         return (!isEmpty() && (index != clone.tail || (index == clone.head && firstCycleOfAFullQueueInProgress)));
302     }
303
304     public E next()
305     {
306         E element;
307
308         if (hasNext())
309         {
310             element = clone.queue.get(index);
311             index = (index + 1) % capacity;
312             if (index == tail && firstCycleOfAFullQueueInProgress)
313                 firstCycleOfAFullQueueInProgress = false;
314         }
315     }

```

```
316         else
317             throw new NoSuchElementException();
318
319         return element;
320     }
321 }
322 }
323
```