BlockingArrayQueue.java

//Template blocking queue class, implemented with emulated circular array.
//Programmer: Randy Miller
//Last modification: April 13, 2015

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

public class BlockingArrayQueue<T> /*implements java.util.Queue<T> (not quite, yet: a few more general methods from the Collection superinterface remain unimplemented, and a few methods implemented here under another name need to be expressly defined as the delegates of required methods, or else renamed)*/ implements Cloneable
{
    private boolean empty = true,
    full = false;
    private int capacity, //maximum queue size
    head = 0, //index of the next element to be dequeued
    tail = 0; //index of the next element to be enqueued
    private ArrayList<T> queue;
    private Lock lock;
    private Condition queueNotEmpty,
    queueNotFull;

    //construction from an initial collection of queue members
    public BlockingArrayQueue(Collection<? extends T> queue, int capacity)
    {
        this.queue = new ArrayList<T>(queue);
        this.capacity = capacity;
        if (queue.size() > capacity)
        {
            for (int i = capacity; i < queue.size(); i++)
            this.queue.remove(i);
            this.queue.trimToSize();
        }
        this.tail = (queue.size() < capacity ? queue.size() : /*capacity*/0);
        if (queue.size() > 0)
        {
BlockingArrayQueue.java

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

//base blocking queue put() implementation
public void enqueue(T member)
{
    lock.lock();
    try
    {
        while (isFull())
        {
            try
            {
                queueNotFull.await();
            }
            catch (InterruptedException exception)
            {
                Thread.currentThread().interrupt();
            }
        }
        if (queue.size() < capacity)
            queue.add(tail, member);
        else
            queue.set(tail, member);
        tail = (tail + 1) % capacity;
        if (tail == head) full = true;
        empty = false;
        queueNotEmpty.signalAll();
    }
    finally
    {
        lock.unlock();
    }
}

//a method of the java.util.Queue interface
public synchronized T element() 
{
    if (isEmpty())
        throw new NoSuchElementException();
    else
    {
        return dequeue(false);
    }
}

//a method of the java.util.Queue interface
public synchronized T peek()
{
    if (isEmpty())
        return null;
    else
    {
        return dequeue(false);
    }
}

//function of the java.util.Queue interface
public synchronized T remove()
{
    if (isEmpty())
        throw new NoSuchElementException();
    else
    {
        return dequeue(true);
    }
}

//a method of the java.util.Queue interface
public synchronized T poll()
{
    if (isEmpty())
        return null;
    else
    {
        return dequeue(true);
    }
}

//base blocking queue get() implementation
public T dequeue(boolean notJustPeeking)
BlockingArrayQueue.java

```java
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  }
207  finally
208  {
209      lock.unlock();
210  }
211  return fifo;
212}
213
@Override
@SuppressWarnings("unchecked")
public synchronized BlockingArrayQueue<T> clone()
{
    BlockingArrayQueue<T> clone;
    try
    {
        clone = (BlockingArrayQueue<T>)super.clone();
    }
    catch (CloneNotSupportedException e) { throw new Error(); } //this won't ever happen
    /*would be better to individually clone() the members of the ArrayList, but Object::clone() is
```
a protected method and I can't see how to bring it off. So if T does not wrap a primitive
we will just get a new ArrayList populated with new references to the same old objects.
Better than nothing I suppose. */
clone.queue = new ArrayList<T>(this.queue);

return clone;

/*It's not usually a good idea to lock an object for time-consuming IO operations, but
I was unable to see how to define a publicly accessible override of the Object::clone()
method for the unknown type T that is the class instantiated by members of the queue,
and so was unable to create a threadsafe clone of the entire containing queue for IO.
I did what I could above, and am now limiting this method's lock on the object to its
call to this.clone(). At least the efficiency problem is minimized*/

@Override
public String toString()
{
    BlockingArrayQueue<T> clone = this.clone();
    StringBuilder textBuffer = new StringBuilder();
    textBuffer.append(String.format("Queued elements, ordered from the head of the line to the end:%n"));
    if (!isEmpty())
    {
        for (int i = 0; i < getSize(); i++)
        {
            String memberString = (clone.queue).get((head + i) % capacity).toString();
            if (memberString.length() > 15)
                memberString = memberString.substring(0, 15);
            textBuffer.append(String.format("Element %3d: %-15s%n", i + 1, memberString));
        }
    }
    else
        textBuffer.append("(queue is empty)\n");
    return textBuffer.toString();
}

//output the members of the queue, oldest first
public void print()
{
```java
BlockingArrayQueue.java

    System.out.print(toString());

  /*The following implementation of an iterator for this queue is not thread-safe or even weakly consistent.
  (by the time that a (mutable) element of the cloned queue is accessed another thread may have altered it
  through the use of a different reference.) Where multi-threaded access to the queue is possible the queue
  must be locked throughout the iteration*/

  public Iterator<T> iterator()
  {
    return new BlockingArrayQueueIterator<T>();
  }

  private class BlockingArrayQueueIterator<E> implements Iterator<E>
  {
    int index;
    boolean firstCycleOfAFullQueueInProgress = false;
    BlockingArrayQueue<E> clone;

    @SuppressWarnings("unchecked")
    public BlockingArrayQueueIterator()
    {
      clone = (BlockingArrayQueue<E>)this.clone();
      index = clone.head;
      if (clone.isFull())
        firstCycleOfAFullQueueInProgress = true;
    }

    public boolean hasNext()
    {
      return (!isEmpty() && (index != clone.tail || (index == clone.head && firstCycleOfAFullQueueInProgress)));
    }

    public E next()
    {
      E element;

      if (hasNext())
      {
        element = clone.queue.get(index);
        index = (index + 1) % capacity;
        if (index == tail && firstCycleOfAFullQueueInProgress)
          firstCycleOfAFullQueueInProgress = false;
      }
  ```
else
    throw new NoSuchElementException();

return element;
}