```java
//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)
        {
```

```
            empty = false;
            full  = (tail == 0 ? true : false);
        }
        lock = new ReentrantLock();
        queueNotEmpty = lock.newCondition();
        queueNotFull  = lock.newCondition();
    }

    //construction from queue capacity only
    public BlockingArrayQueue(int capacity)
    {
        this.queue = new ArrayList<T>(capacity);
        this.capacity = capacity;
        lock = new ReentrantLock();
        queueNotEmpty = lock.newCondition();
        queueNotFull  = lock.newCondition();
    }

    //default constructor, with default capacity of 10
    public BlockingArrayQueue()
    {
        this(10);
    }

    public int getCapacity() { return capacity; }

    public synchronized int getSize() { return  (tail != head ? ((tail + capacity - head) % capacity) :
                                                (full ? capacity : 0)); }

    public synchronized boolean isFull()  { return full;  }

    public synchronized boolean isEmpty() { return empty; }

    //a method of the java.util.Queue interface
    public synchronized boolean add(T member)
    {
        if (isFull())
            throw new IllegalStateException();
        else
        {
            enqueue(member);
            return true;
        }
    }

```

```java
    //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
```

```java
    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)
```

```
    {
        T fifo = null;
        lock.lock();
        try
        {
            while (isEmpty())
            {
                try
                {
                    queueNotEmpty.await();
                }
                catch (InterruptedException exception)
                {
                    Thread.currentThread().interrupt();
                }
            }
            fifo = queue.get(head);
            if (notJustPeeking)
            {
                head = (head + 1) % capacity;
                if (head == tail) empty = true;
                full = false;
                queueNotFull.signalAll();
            }
        }
        finally
        {
            lock.unlock();
        }
        return fifo;
    }

    @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()
  {
```

```
        System.out.print(toString());
    }

    /*The following implementation of an iterator for this queue is not threadsafe 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>)BlockingArrayQueue.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;
        }
    }
}

```