```
/*
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.  Oracle designates this
 * particular file as subject to the "Classpath" exception as provided
 * by Oracle in the LICENSE file that accompanied this code.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 */

/*
 * This file is available under and governed by the GNU General Public
 * License version 2 only, as published by the Free Software Foundation.
 * However, the following notice accompanied the original version of this
 * file:
 *
 * Written by Josh Bloch of Google Inc. and released to the public domain,
 * as explained at http://creativecommons.org/publicdomain/zero/1.0/.
 */

package java.util;

import java.io.Serializable;
import java.util.function.Consumer;
import sun.misc.SharedSecrets;

/**
 * Resizable-array implementation of the {@link Deque} interface.  Array
 * deques have no capacity restrictions; they grow as necessary to support
 * usage.  They are not thread-safe; in the absence of external
 * synchronization, they do not support concurrent access by multiple threads.
 * Null elements are prohibited.  This class is likely to be faster than
 * {@link Stack} when used as a stack, and faster than {@link LinkedList}
 * when used as a queue.
 *
 * <p>Most {@code ArrayDeque} operations run in amortized constant time.
 * Exceptions include {@link #remove(Object) remove}, {@link
 * #removeFirstOccurrence removeFirstOccurrence}, {@link #removeLastOccurrence
 * removeLastOccurrence}, {@link #contains contains}, {@link #iterator
 * iterator.remove()}, and the bulk operations, all of which run in linear
 * time.
 *
 * <p>The iterators returned by this class's {@code iterator} method are
 * <i>fail-fast</i>: If the deque is modified at any time after the iterator
 * is created, in any way except through the iterator's own {@code remove}
 * method, the iterator will generally throw a {@link
 * ConcurrentModificationException}.  Thus, in the face of concurrent
 * modification, the iterator fails quickly and cleanly, rather than risking
 * arbitrary, non-deterministic behavior at an undetermined time in the
 * future.
 *
 * <p>Note that the fail-fast behavior of an iterator cannot be guaranteed
 * as it is, generally speaking, impossible to make any hard guarantees in the
```

```
 * presence of unsynchronized concurrent modification.  Fail-fast iterators
 * throw {@code ConcurrentModificationException} on a best-effort basis.
 * Therefore, it would be wrong to write a program that depended on this
 * exception for its correctness: <i>the fail-fast behavior of iterators
 * should be used only to detect bugs.</i>
 *
 * <p>This class and its iterator implement all of the
 * <em>optional</em> methods of the {@link Collection} and {@link
 * Iterator} interfaces.
 *
 * <p>This class is a member of the
 * <a href="{@docRoot}/../technotes/guides/collections/index.html">
 * Java Collections Framework</a>.
 *
 * @author  Josh Bloch and Doug Lea
 * @since   1.6
 * @param <E> the type of elements held in this collection
 */
public class ArrayDeque<E> extends AbstractCollection<E>
                           implements Deque<E>, Cloneable, Serializable
{
    /**
     * The array in which the elements of the deque are stored.
     * The capacity of the deque is the length of this array, which is
     * always a power of two. The array is never allowed to become
     * full, except transiently within an addX method where it is
     * resized (see doubleCapacity) immediately upon becoming full,
     * thus avoiding head and tail wrapping around to equal each
     * other.  We also guarantee that all array cells not holding
     * deque elements are always null.
     */
    transient Object[] elements; // non-private to simplify nested class access

    /**
     * The index of the element at the head of the deque (which is the
     * element that would be removed by remove() or pop()); or an
     * arbitrary number equal to tail if the deque is empty.
     */
    transient int head;

    /**
     * The index at which the next element would be added to the tail
     * of the deque (via addLast(E), add(E), or push(E)).
     */
    transient int tail;

    /**
     * The minimum capacity that we'll use for a newly created deque.
     * Must be a power of 2.
     */
    private static final int MIN_INITIAL_CAPACITY = 8;

    // ******  Array allocation and resizing utilities ******

    private static int calculateSize(int numElements) {
        int initialCapacity = MIN_INITIAL_CAPACITY;
        // Find the best power of two to hold elements.
        // Tests "<=" because arrays aren't kept full.
        if (numElements >= initialCapacity) {
            initialCapacity = numElements;
            initialCapacity |= (initialCapacity >>>  1);
            initialCapacity |= (initialCapacity >>>  2);
            initialCapacity |= (initialCapacity >>>  4);
            initialCapacity |= (initialCapacity >>>  8);
            initialCapacity |= (initialCapacity >>> 16);
            initialCapacity++;

```

```
135             if (initialCapacity < 0)   // Too many elements, must back off
136                 initialCapacity >>>= 1;// Good luck allocating 2 ^ 30 elements
137         }
138         return initialCapacity;
139     }
140 
141     /**
142      * Allocates empty array to hold the given number of elements.
143      *
144      * @param numElements  the number of elements to hold
145      */
146     private void allocateElements(int numElements) {
147         elements = new Object[calculateSize(numElements)];
148     }
149 
150     /**
151      * Doubles the capacity of this deque.  Call only when full, i.e.,
152      * when head and tail have wrapped around to become equal.
153      */
154     private void doubleCapacity() {
155         assert head == tail;
156         int p = head;
157         int n = elements.length;
158         int r = n - p; // number of elements to the right of p
159         int newCapacity = n << 1;
160         if (newCapacity < 0)
161             throw new IllegalStateException("Sorry, deque too big");
162         Object[] a = new Object[newCapacity];
163         System.arraycopy(elements, p, a, 0, r);
164         System.arraycopy(elements, 0, a, r, p);
165         elements = a;
166         head = 0;
167         tail = n;
168     }
169 
170     /**
171      * Copies the elements from our element array into the specified array,
172      * in order (from first to last element in the deque).  It is assumed
173      * that the array is large enough to hold all elements in the deque.
174      *
175      * @return its argument
176      */
177     private <T> T[] copyElements(T[] a) {
178         if (head < tail) {
179             System.arraycopy(elements, head, a, 0, size());
180         } else if (head > tail) {
181             int headPortionLen = elements.length - head;
182             System.arraycopy(elements, head, a, 0, headPortionLen);
183             System.arraycopy(elements, 0, a, headPortionLen, tail);
184         }
185         return a;
186     }
187 
188     /**
189      * Constructs an empty array deque with an initial capacity
190      * sufficient to hold 16 elements.
191      */
192     public ArrayDeque() {
193         elements = new Object[16];
194     }
195 
196     /**
197      * Constructs an empty array deque with an initial capacity
198      * sufficient to hold the specified number of elements.
199      *
200      * @param numElements  lower bound on initial capacity of the deque
201      */
```

```
202     public ArrayDeque(int numElements) {
203         allocateElements(numElements);
204     }
205
206     /**
207      * Constructs a deque containing the elements of the specified
208      * collection, in the order they are returned by the collection's
209      * iterator.  (The first element returned by the collection's
210      * iterator becomes the first element, or <i>front</i> of the
211      * deque.)
212      *
213      * @param c the collection whose elements are to be placed into the deque
214      * @throws NullPointerException if the specified collection is null
215      */
216     public ArrayDeque(Collection<? extends E> c) {
217         allocateElements(c.size());
218         addAll(c);
219     }
220
221     // The main insertion and extraction methods are addFirst,
222     // addLast, pollFirst, pollLast. The other methods are defined in
223     // terms of these.
224
225     /**
226      * Inserts the specified element at the front of this deque.
227      *
228      * @param e the element to add
229      * @throws NullPointerException if the specified element is null
230      */
231     public void addFirst(E e) {
232         if (e == null)
233             throw new NullPointerException();
234         elements[head = (head - 1) & (elements.length - 1)] = e;
235         if (head == tail)
236             doubleCapacity();
237     }
238
239     /**
240      * Inserts the specified element at the end of this deque.
241      *
242      * <p>This method is equivalent to {@link #add}.
243      *
244      * @param e the element to add
245      * @throws NullPointerException if the specified element is null
246      */
247     public void addLast(E e) {
248         if (e == null)
249             throw new NullPointerException();
250         elements[tail] = e;
251         if ( (tail = (tail + 1) & (elements.length - 1)) == head)
252             doubleCapacity();
253     }
254
255     /**
256      * Inserts the specified element at the front of this deque.
257      *
258      * @param e the element to add
259      * @return {@code true} (as specified by {@link Deque#offerFirst})
260      * @throws NullPointerException if the specified element is null
261      */
262     public boolean offerFirst(E e) {
263         addFirst(e);
264         return true;
265     }
266
267     /**
268      * Inserts the specified element at the end of this deque.
```

```
269      *
270      * @param e the element to add
271      * @return {@code true} (as specified by {@link Deque#offerLast})
272      * @throws NullPointerException if the specified element is null
273      */
274     public boolean offerLast(E e) {
275         addLast(e);
276         return true;
277     }
278 
279     /**
280      * @throws NoSuchElementException {@inheritDoc}
281      */
282     public E removeFirst() {
283         E x = pollFirst();
284         if (x == null)
285             throw new NoSuchElementException();
286         return x;
287     }
288 
289     /**
290      * @throws NoSuchElementException {@inheritDoc}
291      */
292     public E removeLast() {
293         E x = pollLast();
294         if (x == null)
295             throw new NoSuchElementException();
296         return x;
297     }
298 
299     public E pollFirst() {
300         int h = head;
301         @SuppressWarnings("unchecked")
302         E result = (E) elements[h];
303         // Element is null if deque empty
304         if (result == null)
305             return null;
306         elements[h] = null;     // Must null out slot
307         head = (h + 1) & (elements.length - 1);
308         return result;
309     }
310 
311     public E pollLast() {
312         int t = (tail - 1) & (elements.length - 1);
313         @SuppressWarnings("unchecked")
314         E result = (E) elements[t];
315         if (result == null)
316             return null;
317         elements[t] = null;
318         tail = t;
319         return result;
320     }
321 
322     /**
323      * @throws NoSuchElementException {@inheritDoc}
324      */
325     public E getFirst() {
326         @SuppressWarnings("unchecked")
327         E result = (E) elements[head];
328         if (result == null)
329             throw new NoSuchElementException();
330         return result;
331     }
332 
333     /**
334      * @throws NoSuchElementException {@inheritDoc}
335      */
```

```
336     public E getLast() {
337         @SuppressWarnings("unchecked")
338         E result = (E) elements[(tail - 1) & (elements.length - 1)];
339         if (result == null)
340             throw new NoSuchElementException();
341         return result;
342     }
343 
344     @SuppressWarnings("unchecked")
345     public E peekFirst() {
346         // elements[head] is null if deque empty
347         return (E) elements[head];
348     }
349 
350     @SuppressWarnings("unchecked")
351     public E peekLast() {
352         return (E) elements[(tail - 1) & (elements.length - 1)];
353     }
354 
355     /**
356      * Removes the first occurrence of the specified element in this
357      * deque (when traversing the deque from head to tail).
358      * If the deque does not contain the element, it is unchanged.
359      * More formally, removes the first element {@code e} such that
360      * {@code o.equals(e)} (if such an element exists).
361      * Returns {@code true} if this deque contained the specified element
362      * (or equivalently, if this deque changed as a result of the call).
363      *
364      * @param o element to be removed from this deque, if present
365      * @return {@code true} if the deque contained the specified element
366      */
367     public boolean removeFirstOccurrence(Object o) {
368         if (o == null)
369             return false;
370         int mask = elements.length - 1;
371         int i = head;
372         Object x;
373         while ( (x = elements[i]) != null) {
374             if (o.equals(x)) {
375                 delete(i);
376                 return true;
377             }
378             i = (i + 1) & mask;
379         }
380         return false;
381     }
382 
383     /**
384      * Removes the last occurrence of the specified element in this
385      * deque (when traversing the deque from head to tail).
386      * If the deque does not contain the element, it is unchanged.
387      * More formally, removes the last element {@code e} such that
388      * {@code o.equals(e)} (if such an element exists).
389      * Returns {@code true} if this deque contained the specified element
390      * (or equivalently, if this deque changed as a result of the call).
391      *
392      * @param o element to be removed from this deque, if present
393      * @return {@code true} if the deque contained the specified element
394      */
395     public boolean removeLastOccurrence(Object o) {
396         if (o == null)
397             return false;
398         int mask = elements.length - 1;
399         int i = (tail - 1) & mask;
400         Object x;
401         while ( (x = elements[i]) != null) {
402             if (o.equals(x)) {
```

```
403                 delete(i);
404                 return true;
405             }
406             i = (i - 1) & mask;
407         }
408         return false;
409     }
410 
411     // *** Queue methods ***
412 
413     /**
414      * Inserts the specified element at the end of this deque.
415      *
416      * <p>This method is equivalent to {@link #addLast}.
417      *
418      * @param e the element to add
419      * @return {@code true} (as specified by {@link Collection#add})
420      * @throws NullPointerException if the specified element is null
421      */
422     public boolean add(E e) {
423         addLast(e);
424         return true;
425     }
426 
427     /**
428      * Inserts the specified element at the end of this deque.
429      *
430      * <p>This method is equivalent to {@link #offerLast}.
431      *
432      * @param e the element to add
433      * @return {@code true} (as specified by {@link Queue#offer})
434      * @throws NullPointerException if the specified element is null
435      */
436     public boolean offer(E e) {
437         return offerLast(e);
438     }
439 
440     /**
441      * Retrieves and removes the head of the queue represented by this deque.
442      *
443      * This method differs from {@link #poll poll} only in that it throws an
444      * exception if this deque is empty.
445      *
446      * <p>This method is equivalent to {@link #removeFirst}.
447      *
448      * @return the head of the queue represented by this deque
449      * @throws NoSuchElementException {@inheritDoc}
450      */
451     public E remove() {
452         return removeFirst();
453     }
454 
455     /**
456      * Retrieves and removes the head of the queue represented by this deque
457      * (in other words, the first element of this deque), or returns
458      * {@code null} if this deque is empty.
459      *
460      * <p>This method is equivalent to {@link #pollFirst}.
461      *
462      * @return the head of the queue represented by this deque, or
463      *         {@code null} if this deque is empty
464      */
465     public E poll() {
466         return pollFirst();
467     }
468 
469     /**
```

```
470      * Retrieves, but does not remove, the head of the queue represented by
471      * this deque.  This method differs from {@link #peek peek} only in
472      * that it throws an exception if this deque is empty.
473      *
474      * <p>This method is equivalent to {@link #getFirst}.
475      *
476      * @return the head of the queue represented by this deque
477      * @throws NoSuchElementException {@inheritDoc}
478      */
479     public E element() {
480         return getFirst();
481     }
482 
483     /**
484      * Retrieves, but does not remove, the head of the queue represented by
485      * this deque, or returns {@code null} if this deque is empty.
486      *
487      * <p>This method is equivalent to {@link #peekFirst}.
488      *
489      * @return the head of the queue represented by this deque, or
490      *         {@code null} if this deque is empty
491      */
492     public E peek() {
493         return peekFirst();
494     }
495 
496     // *** Stack methods ***
497 
498     /**
499      * Pushes an element onto the stack represented by this deque.  In other
500      * words, inserts the element at the front of this deque.
501      *
502      * <p>This method is equivalent to {@link #addFirst}.
503      *
504      * @param e the element to push
505      * @throws NullPointerException if the specified element is null
506      */
507     public void push(E e) {
508         addFirst(e);
509     }
510 
511     /**
512      * Pops an element from the stack represented by this deque.  In other
513      * words, removes and returns the first element of this deque.
514      *
515      * <p>This method is equivalent to {@link #removeFirst()}.
516      *
517      * @return the element at the front of this deque (which is the top
518      *         of the stack represented by this deque)
519      * @throws NoSuchElementException {@inheritDoc}
520      */
521     public E pop() {
522         return removeFirst();
523     }
524 
525     private void checkInvariants() {
526         assert elements[tail] == null;
527         assert head == tail ? elements[head] == null :
528             (elements[head] != null &&
529              elements[(tail - 1) & (elements.length - 1)] != null);
530         assert elements[(head - 1) & (elements.length - 1)] == null;
531     }
532 
533     /**
534      * Removes the element at the specified position in the elements array,
535      * adjusting head and tail as necessary.  This can result in motion of
536      * elements backwards or forwards in the array.
```

```
537      *
538      * <p>This method is called delete rather than remove to emphasize
539      * that its semantics differ from those of {@link List#remove(int)}.
540      *
541      * @return true if elements moved backwards
542      */
543     private boolean delete(int i) {
544         checkInvariants();
545         final Object[] elements = this.elements;
546         final int mask = elements.length - 1;
547         final int h = head;
548         final int t = tail;
549         final int front = (i - h) & mask;
550         final int back  = (t - i) & mask;
551 
552         // Invariant: head <= i < tail mod circularity
553         if (front >= ((t - h) & mask))
554             throw new ConcurrentModificationException();
555 
556         // Optimize for least element motion
557         if (front < back) {
558             if (h <= i) {
559                 System.arraycopy(elements, h, elements, h + 1, front);
560             } else { // Wrap around
561                 System.arraycopy(elements, 0, elements, 1, i);
562                 elements[0] = elements[mask];
563                 System.arraycopy(elements, h, elements, h + 1, mask - h);
564             }
565             elements[h] = null;
566             head = (h + 1) & mask;
567             return false;
568         } else {
569             if (i < t) { // Copy the null tail as well
570                 System.arraycopy(elements, i + 1, elements, i, back);
571                 tail = t - 1;
572             } else { // Wrap around
573                 System.arraycopy(elements, i + 1, elements, i, mask - i);
574                 elements[mask] = elements[0];
575                 System.arraycopy(elements, 1, elements, 0, t);
576                 tail = (t - 1) & mask;
577             }
578             return true;
579         }
580     }
581 
582     // *** Collection Methods ***
583 
584     /**
585      * Returns the number of elements in this deque.
586      *
587      * @return the number of elements in this deque
588      */
589     public int size() {
590         return (tail - head) & (elements.length - 1);
591     }
592 
593     /**
594      * Returns {@code true} if this deque contains no elements.
595      *
596      * @return {@code true} if this deque contains no elements
597      */
598     public boolean isEmpty() {
599         return head == tail;
600     }
601 
602     /**
603      * Returns an iterator over the elements in this deque.  The elements
```

```
604      * will be ordered from first (head) to last (tail).  This is the same
605      * order that elements would be dequeued (via successive calls to
606      * {@link #remove} or popped (via successive calls to {@link #pop}).
607      *
608      * @return an iterator over the elements in this deque
609      */
610     public Iterator<E> iterator() {
611         return new DeqIterator();
612     }
613 
614     public Iterator<E> descendingIterator() {
615         return new DescendingIterator();
616     }
617 
618     private class DeqIterator implements Iterator<E> {
619         /**
620          * Index of element to be returned by subsequent call to next.
621          */
622         private int cursor = head;
623 
624         /**
625          * Tail recorded at construction (also in remove), to stop
626          * iterator and also to check for comodification.
627          */
628         private int fence = tail;
629 
630         /**
631          * Index of element returned by most recent call to next.
632          * Reset to -1 if element is deleted by a call to remove.
633          */
634         private int lastRet = -1;
635 
636         public boolean hasNext() {
637             return cursor != fence;
638         }
639 
640         public E next() {
641             if (cursor == fence)
642                 throw new NoSuchElementException();
643             @SuppressWarnings("unchecked")
644             E result = (E) elements[cursor];
645             // This check doesn't catch all possible comodifications,
646             // but does catch the ones that corrupt traversal
647             if (tail != fence || result == null)
648                 throw new ConcurrentModificationException();
649             lastRet = cursor;
650             cursor = (cursor + 1) & (elements.length - 1);
651             return result;
652         }
653 
654         public void remove() {
655             if (lastRet < 0)
656                 throw new IllegalStateException();
657             if (delete(lastRet)) { // if left-shifted, undo increment in next()
658                 cursor = (cursor - 1) & (elements.length - 1);
659                 fence = tail;
660             }
661             lastRet = -1;
662         }
663 
664         public void forEachRemaining(Consumer<? super E> action) {
665             Objects.requireNonNull(action);
666             Object[] a = elements;
667             int m = a.length - 1, f = fence, i = cursor;
668             cursor = f;
669             while (i != f) {
670                 @SuppressWarnings("unchecked") E e = (E)a[i];
```

```
671                 i = (i + 1) & m;
672                 if (e == null)
673                     throw new ConcurrentModificationException();
674                 action.accept(e);
675             }
676         }
677     }
678
679     private class DescendingIterator implements Iterator<E> {
680         /*
681          * This class is nearly a mirror-image of DeqIterator, using
682          * tail instead of head for initial cursor, and head instead of
683          * tail for fence.
684          */
685         private int cursor = tail;
686         private int fence = head;
687         private int lastRet = -1;
688
689         public boolean hasNext() {
690             return cursor != fence;
691         }
692
693         public E next() {
694             if (cursor == fence)
695                 throw new NoSuchElementException();
696             cursor = (cursor - 1) & (elements.length - 1);
697             @SuppressWarnings("unchecked")
698             E result = (E) elements[cursor];
699             if (head != fence || result == null)
700                 throw new ConcurrentModificationException();
701             lastRet = cursor;
702             return result;
703         }
704
705         public void remove() {
706             if (lastRet < 0)
707                 throw new IllegalStateException();
708             if (!delete(lastRet)) {
709                 cursor = (cursor + 1) & (elements.length - 1);
710                 fence = head;
711             }
712             lastRet = -1;
713         }
714     }
715
716     /**
717      * Returns {@code true} if this deque contains the specified element.
718      * More formally, returns {@code true} if and only if this deque contains
719      * at least one element {@code e} such that {@code o.equals(e)}.
720      *
721      * @param o object to be checked for containment in this deque
722      * @return {@code true} if this deque contains the specified element
723      */
724     public boolean contains(Object o) {
725         if (o == null)
726             return false;
727         int mask = elements.length - 1;
728         int i = head;
729         Object x;
730         while ( (x = elements[i]) != null) {
731             if (o.equals(x))
732                 return true;
733             i = (i + 1) & mask;
734         }
735         return false;
736     }
737
```

```
738     /**
739      * Removes a single instance of the specified element from this deque.
740      * If the deque does not contain the element, it is unchanged.
741      * More formally, removes the first element {@code e} such that
742      * {@code o.equals(e)} (if such an element exists).
743      * Returns {@code true} if this deque contained the specified element
744      * (or equivalently, if this deque changed as a result of the call).
745      *
746      * <p>This method is equivalent to {@link #removeFirstOccurrence(Object)}.
747      *
748      * @param o element to be removed from this deque, if present
749      * @return {@code true} if this deque contained the specified element
750      */
751     public boolean remove(Object o) {
752         return removeFirstOccurrence(o);
753     }
754 
755     /**
756      * Removes all of the elements from this deque.
757      * The deque will be empty after this call returns.
758      */
759     public void clear() {
760         int h = head;
761         int t = tail;
762         if (h != t) { // clear all cells
763             head = tail = 0;
764             int i = h;
765             int mask = elements.length - 1;
766             do {
767                 elements[i] = null;
768                 i = (i + 1) & mask;
769             } while (i != t);
770         }
771     }
772 
773     /**
774      * Returns an array containing all of the elements in this deque
775      * in proper sequence (from first to last element).
776      *
777      * <p>The returned array will be "safe" in that no references to it are
778      * maintained by this deque.  (In other words, this method must allocate
779      * a new array).  The caller is thus free to modify the returned array.
780      *
781      * <p>This method acts as bridge between array-based and collection-based
782      * APIs.
783      *
784      * @return an array containing all of the elements in this deque
785      */
786     public Object[] toArray() {
787         return copyElements(new Object[size()]);
788     }
789 
790     /**
791      * Returns an array containing all of the elements in this deque in
792      * proper sequence (from first to last element); the runtime type of the
793      * returned array is that of the specified array.  If the deque fits in
794      * the specified array, it is returned therein.  Otherwise, a new array
795      * is allocated with the runtime type of the specified array and the
796      * size of this deque.
797      *
798      * <p>If this deque fits in the specified array with room to spare
799      * (i.e., the array has more elements than this deque), the element in
800      * the array immediately following the end of the deque is set to
801      * {@code null}.
802      *
803      * <p>Like the {@link #toArray()} method, this method acts as bridge between
804      * array-based and collection-based APIs.  Further, this method allows
```

```
805      * precise control over the runtime type of the output array, and may,
806      * under certain circumstances, be used to save allocation costs.
807      *
808      * <p>Suppose {@code x} is a deque known to contain only strings.
809      * The following code can be used to dump the deque into a newly
810      * allocated array of {@code String}:
811      *
812      *  <pre> {@code String[] y = x.toArray(new String[0]);}</pre>
813      *
814      * Note that {@code toArray(new Object[0])} is identical in function to
815      * {@code toArray()}.
816      *
817      * @param a the array into which the elements of the deque are to
818      *          be stored, if it is big enough; otherwise, a new array of the
819      *          same runtime type is allocated for this purpose
820      * @return an array containing all of the elements in this deque
821      * @throws ArrayStoreException if the runtime type of the specified array
822      *         is not a supertype of the runtime type of every element in
823      *         this deque
824      * @throws NullPointerException if the specified array is null
825      */
826     @SuppressWarnings("unchecked")
827     public <T> T[] toArray(T[] a) {
828         int size = size();
829         if (a.length < size)
830             a = (T[])java.lang.reflect.Array.newInstance(
831                     a.getClass().getComponentType(), size);
832         copyElements(a);
833         if (a.length > size)
834             a[size] = null;
835         return a;
836     }
837
838     // *** Object methods ***
839
840     /**
841      * Returns a copy of this deque.
842      *
843      * @return a copy of this deque
844      */
845     public ArrayDeque<E> clone() {
846         try {
847             @SuppressWarnings("unchecked")
848             ArrayDeque<E> result = (ArrayDeque<E>) super.clone();
849             result.elements = Arrays.copyOf(elements, elements.length);
850             return result;
851         } catch (CloneNotSupportedException e) {
852             throw new AssertionError();
853         }
854     }
855
856     private static final long serialVersionUID = 2340985798034038923L;
857
858     /**
859      * Saves this deque to a stream (that is, serializes it).
860      *
861      * @serialData The current size ({@code int}) of the deque,
862      * followed by all of its elements (each an object reference) in
863      * first-to-last order.
864      */
865     private void writeObject(java.io.ObjectOutputStream s)
866             throws java.io.IOException {
867         s.defaultWriteObject();
868
869         // Write out size
870         s.writeInt(size());
871
```

```
872         // Write out elements in order.
873         int mask = elements.length - 1;
874         for (int i = head; i != tail; i = (i + 1) & mask)
875             s.writeObject(elements[i]);
876     }
877
878     /**
879      * Reconstitutes this deque from a stream (that is, deserializes it).
880      */
881     private void readObject(java.io.ObjectInputStream s)
882             throws java.io.IOException, ClassNotFoundException {
883         s.defaultReadObject();
884
885         // Read in size and allocate array
886         int size = s.readInt();
887         int capacity = calculateSize(size);
888         SharedSecrets.getJavaOISAccess().checkArray(s, Object[].class, capacity);
889         allocateElements(size);
890         head = 0;
891         tail = size;
892
893         // Read in all elements in the proper order.
894         for (int i = 0; i < size; i++)
895             elements[i] = s.readObject();
896     }
897
898     /**
899      * Creates a <em><a href="Spliterator.html#binding">late-binding</a></em>
900      * and <em>fail-fast</em> {@link Spliterator} over the elements in this
901      * deque.
902      *
903      * <p>The {@code Spliterator} reports {@link Spliterator#SIZED},
904      * {@link Spliterator#SUBSIZED}, {@link Spliterator#ORDERED}, and
905      * {@link Spliterator#NONNULL}.  Overriding implementations should document
906      * the reporting of additional characteristic values.
907      *
908      * @return a {@code Spliterator} over the elements in this deque
909      * @since 1.8
910      */
911     public Spliterator<E> spliterator() {
912         return new DeqSpliterator<E>(this, -1, -1);
913     }
914
915     static final class DeqSpliterator<E> implements Spliterator<E> {
916         private final ArrayDeque<E> deq;
917         private int fence;  // -1 until first use
918         private int index;  // current index, modified on traverse/split
919
920         /** Creates new spliterator covering the given array and range */
921         DeqSpliterator(ArrayDeque<E> deq, int origin, int fence) {
922             this.deq = deq;
923             this.index = origin;
924             this.fence = fence;
925         }
926
927         private int getFence() { // force initialization
928             int t;
929             if ((t = fence) < 0) {
930                 t = fence = deq.tail;
931                 index = deq.head;
932             }
933             return t;
934         }
935
936         public DeqSpliterator<E> trySplit() {
937             int t = getFence(), h = index, n = deq.elements.length;
938             if (h != t && ((h + 1) & (n - 1)) != t) {
```

```
939                 if (h > t)
940                     t += n;
941                 int m = ((h + t) >>> 1) & (n - 1);
942                 return new DeqSpliterator<>(deq, h, index = m);
943             }
944             return null;
945         }
946
947         public void forEachRemaining(Consumer<? super E> consumer) {
948             if (consumer == null)
949                 throw new NullPointerException();
950             Object[] a = deq.elements;
951             int m = a.length - 1, f = getFence(), i = index;
952             index = f;
953             while (i != f) {
954                 @SuppressWarnings("unchecked") E e = (E)a[i];
955                 i = (i + 1) & m;
956                 if (e == null)
957                     throw new ConcurrentModificationException();
958                 consumer.accept(e);
959             }
960         }
961
962         public boolean tryAdvance(Consumer<? super E> consumer) {
963             if (consumer == null)
964                 throw new NullPointerException();
965             Object[] a = deq.elements;
966             int m = a.length - 1, f = getFence(), i = index;
967             if (i != fence) {
968                 @SuppressWarnings("unchecked") E e = (E)a[i];
969                 index = (i + 1) & m;
970                 if (e == null)
971                     throw new ConcurrentModificationException();
972                 consumer.accept(e);
973                 return true;
974             }
975             return false;
976         }
977
978         public long estimateSize() {
979             int n = getFence() - index;
980             if (n < 0)
981                 n += deq.elements.length;
982             return (long) n;
983         }
984
985         @Override
986         public int characteristics() {
987             return Spliterator.ORDERED | Spliterator.SIZED |
988                 Spliterator.NONNULL | Spliterator.SUBSIZED;
989         }
990     }
991
992 }
993
```