```
/*
 * 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 Doug Lea with assistance from members of JCP JSR-166
 * Expert Group and released to the public domain, as explained at
 * http://creativecommons.org/publicdomain/zero/1.0/
 */

package java.util;

/**
 * A collection designed for holding elements prior to processing.
 * Besides basic {@link java.util.Collection Collection} operations,
 * queues provide additional insertion, extraction, and inspection
 * operations.  Each of these methods exists in two forms: one throws
 * an exception if the operation fails, the other returns a special
 * value (either {@code null} or {@code false}, depending on the
 * operation).  The latter form of the insert operation is designed
 * specifically for use with capacity-restricted {@code Queue}
 * implementations; in most implementations, insert operations cannot
 * fail.
 *
 * <table BORDER CELLPADDING=3 CELLSPACING=1>
 * <caption>Summary of Queue methods</caption>
 *  <tr>
 *    <td></td>
 *    <td ALIGN=CENTER><em>Throws exception</em></td>
 *    <td ALIGN=CENTER><em>Returns special value</em></td>
 *  </tr>
 *  <tr>
 *    <td><b>Insert</b></td>
 *    <td>{@link Queue#add add(e)}</td>
 *    <td>{@link Queue#offer offer(e)}</td>
 *  </tr>
 *  <tr>
 *    <td><b>Remove</b></td>
 *    <td>{@link Queue#remove remove()}</td>
 *    <td>{@link Queue#poll poll()}</td>
 *  </tr>
 *  <tr>
```

```
68   *    <td><b>Examine</b></td>
69   *    <td>{@link Queue#element element()}</td>
70   *    <td>{@link Queue#peek peek()}</td>
71   *  </tr>
72   * </table>
73   *
74   * <p>Queues typically, but do not necessarily, order elements in a
75   * FIFO (first-in-first-out) manner.  Among the exceptions are
76   * priority queues, which order elements according to a supplied
77   * comparator, or the elements' natural ordering, and LIFO queues (or
78   * stacks) which order the elements LIFO (last-in-first-out).
79   * Whatever the ordering used, the <em>head</em> of the queue is that
80   * element which would be removed by a call to {@link #remove() } or
81   * {@link #poll()}.  In a FIFO queue, all new elements are inserted at
82   * the <em>tail</em> of the queue. Other kinds of queues may use
83   * different placement rules.  Every {@code Queue} implementation
84   * must specify its ordering properties.
85   *
86   * <p>The {@link #offer offer} method inserts an element if possible,
87   * otherwise returning {@code false}.  This differs from the {@link
88   * java.util.Collection#add Collection.add} method, which can fail to
89   * add an element only by throwing an unchecked exception.  The
90   * {@code offer} method is designed for use when failure is a normal,
91   * rather than exceptional occurrence, for example, in fixed-capacity
92   * (or &amp;quot;bounded&amp;quot;) queues.
93   *
94   * <p>The {@link #remove()} and {@link #poll()} methods remove and
95   * return the head of the queue.
96   * Exactly which element is removed from the queue is a
97   * function of the queue's ordering policy, which differs from
98   * implementation to implementation. The {@code remove()} and
99   * {@code poll()} methods differ only in their behavior when the
100  * queue is empty: the {@code remove()} method throws an exception,
101  * while the {@code poll()} method returns {@code null}.
102  *
103  * <p>The {@link #element()} and {@link #peek()} methods return, but do
104  * not remove, the head of the queue.
105  *
106  * <p>The {@code Queue} interface does not define the <i>blocking queue
107  * methods</i>, which are common in concurrent programming.  These methods,
108  * which wait for elements to appear or for space to become available, are
109  * defined in the {@link java.util.concurrent.BlockingQueue} interface, which
110  * extends this interface.
111  *
112  * <p>{@code Queue} implementations generally do not allow insertion
113  * of {@code null} elements, although some implementations, such as
114  * {@link LinkedList}, do not prohibit insertion of {@code null}.
115  * Even in the implementations that permit it, {@code null} should
116  * not be inserted into a {@code Queue}, as {@code null} is also
117  * used as a special return value by the {@code poll} method to
118  * indicate that the queue contains no elements.
119  *
120  * <p>{@code Queue} implementations generally do not define
121  * element-based versions of methods {@code equals} and
122  * {@code hashCode} but instead inherit the identity based versions
123  * from class {@code Object}, because element-based equality is not
124  * always well-defined for queues with the same elements but different
125  * ordering properties.
126  *
127  *
128  * <p>This interface is a member of the
129  * <a href="{@docRoot}/../technotes/guides/collections/index.html">
130  * Java Collections Framework</a>.
131  *
132  * @see java.util.Collection
133  * @see LinkedList
134  * @see PriorityQueue
```

```
135  * @see java.util.concurrent.LinkedBlockingQueue
136  * @see java.util.concurrent.BlockingQueue
137  * @see java.util.concurrent.ArrayBlockingQueue
138  * @see java.util.concurrent.LinkedBlockingQueue
139  * @see java.util.concurrent.PriorityBlockingQueue
140  * @since 1.5
141  * @author Doug Lea
142  * @param <E> the type of elements held in this collection
143  */
144 public interface Queue<E> extends Collection<E> {
145     /**
146      * Inserts the specified element into this queue if it is possible to do so
147      * immediately without violating capacity restrictions, returning
148      * {@code true} upon success and throwing an {@code IllegalStateException}
149      * if no space is currently available.
150      *
151      * @param e the element to add
152      * @return {@code true} (as specified by {@link Collection#add})
153      * @throws IllegalStateException if the element cannot be added at this
154      *         time due to capacity restrictions
155      * @throws ClassCastException if the class of the specified element
156      *         prevents it from being added to this queue
157      * @throws NullPointerException if the specified element is null and
158      *         this queue does not permit null elements
159      * @throws IllegalArgumentException if some property of this element
160      *         prevents it from being added to this queue
161      */
162     boolean add(E e);
163 
164     /**
165      * Inserts the specified element into this queue if it is possible to do
166      * so immediately without violating capacity restrictions.
167      * When using a capacity-restricted queue, this method is generally
168      * preferable to {@link #add}, which can fail to insert an element only
169      * by throwing an exception.
170      *
171      * @param e the element to add
172      * @return {@code true} if the element was added to this queue, else
173      *         {@code false}
174      * @throws ClassCastException if the class of the specified element
175      *         prevents it from being added to this queue
176      * @throws NullPointerException if the specified element is null and
177      *         this queue does not permit null elements
178      * @throws IllegalArgumentException if some property of this element
179      *         prevents it from being added to this queue
180      */
181     boolean offer(E e);
182 
183     /**
184      * Retrieves and removes the head of this queue.  This method differs
185      * from {@link #poll poll} only in that it throws an exception if this
186      * queue is empty.
187      *
188      * @return the head of this queue
189      * @throws NoSuchElementException if this queue is empty
190      */
191     E remove();
192 
193     /**
194      * Retrieves and removes the head of this queue,
195      * or returns {@code null} if this queue is empty.
196      *
197      * @return the head of this queue, or {@code null} if this queue is empty
198      */
199     E poll();
200 
201     /**
```

```
202      * Retrieves, but does not remove, the head of this queue.  This method
203      * differs from {@link #peek peek} only in that it throws an exception
204      * if this queue is empty.
205      *
206      * @return the head of this queue
207      * @throws NoSuchElementException if this queue is empty
208      */
209     E element();
210
211     /**
212      * Retrieves, but does not remove, the head of this queue,
213      * or returns {@code null} if this queue is empty.
214      *
215      * @return the head of this queue, or {@code null} if this queue is empty
216      */
217     E peek();
218 }
219
```