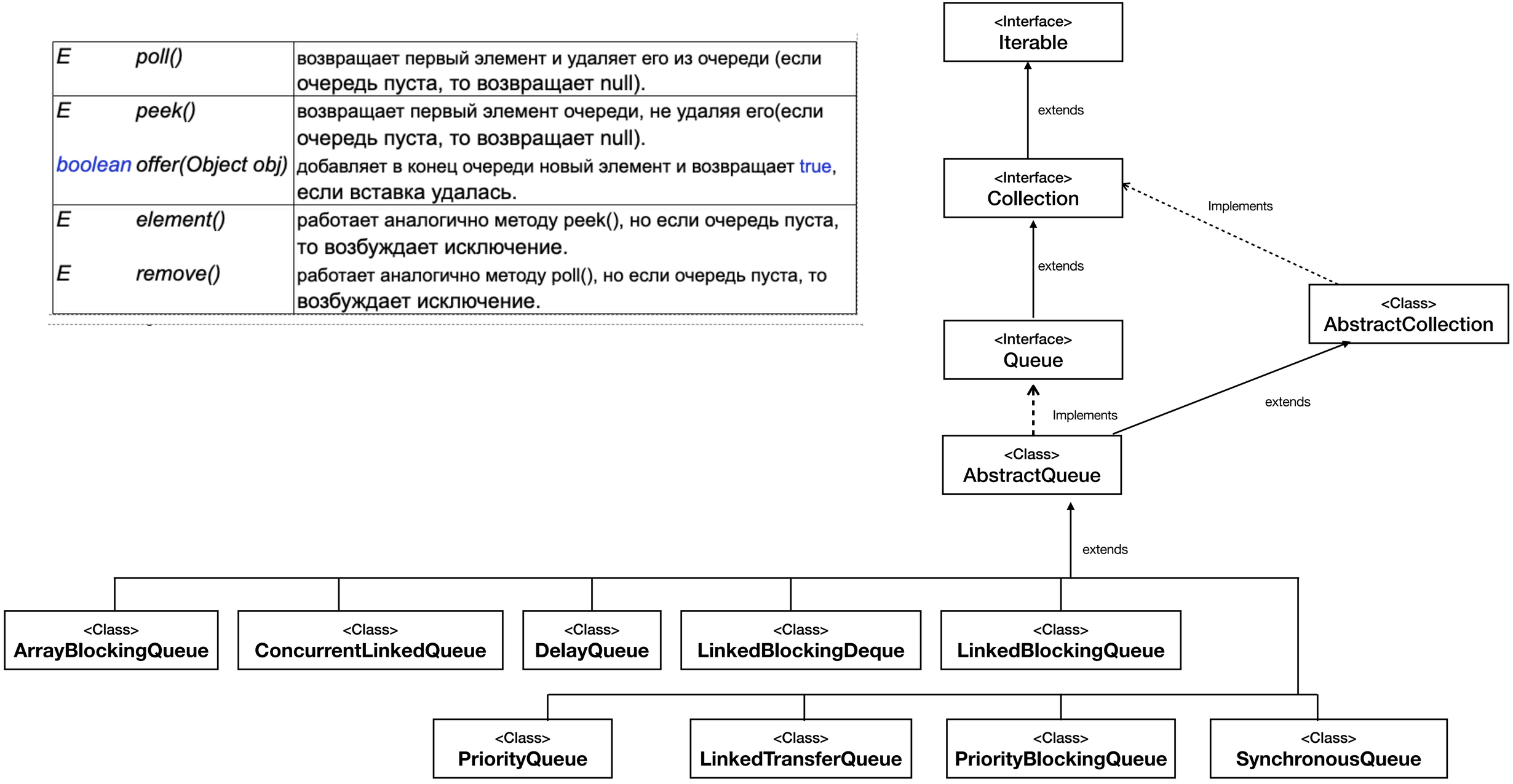
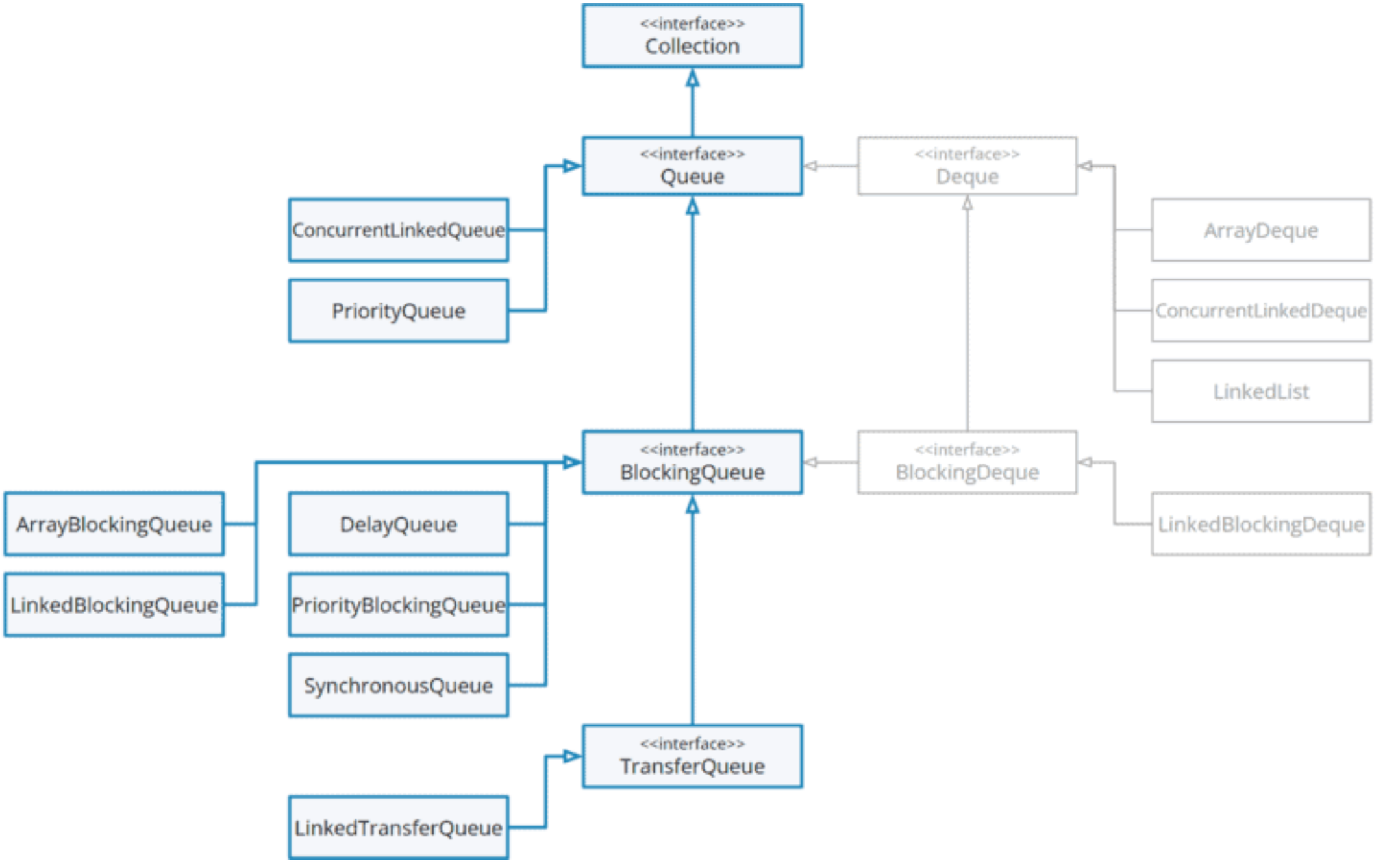
Queues in Java

Игорь Выпов 28.05.2024

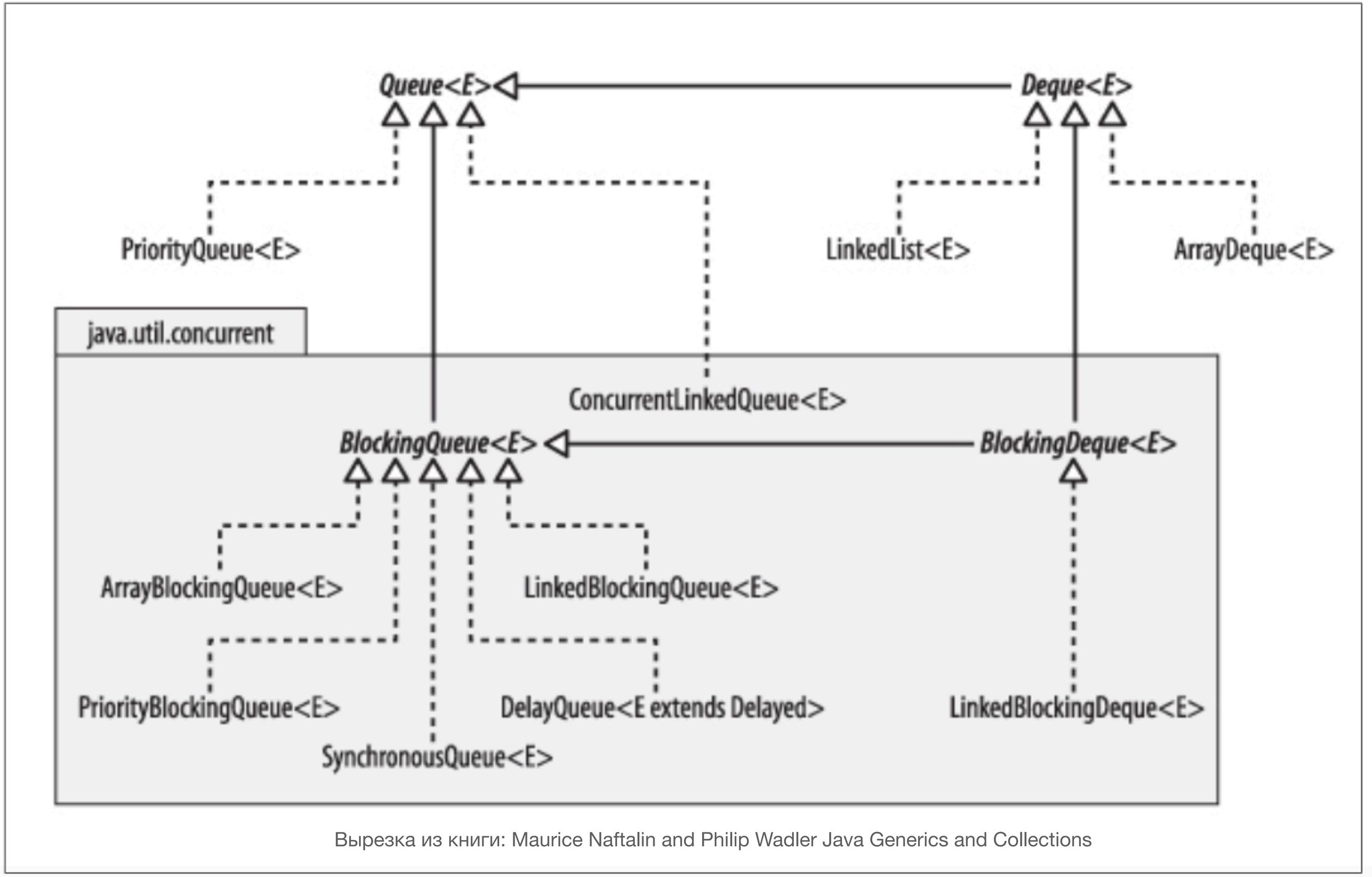
AbstractQueue

E poll()	возвращает первый элемент и удаляет его из очереди (если очередь пуста, то возвращает null).
E peek()	возвращает первый элемент очереди, не удаляя его(если очередь пуста, то возвращает null).
boolean offer(Object obj)	добавляет в конец очереди новый элемент и возвращает true, если вставка удалась.
E element()	работает аналогично методу peek(), но если очередь пуста, то возбуждает исключение.
E remove()	работает аналогично методу poll(), но если очередь пуста, то возбуждает исключение.





https://www.happycoders.eu/algorithms/java-queue/



Вырезка из книги: Maurice Naftalin and Philip Wadler Java Generics and Collections

Особенности многопоточных очередей:

Потокобезопасность

Управления ресурсами

Производительность

Блокирующие очереди VS Неблокирующие очереди

Блокирующие или не блокирующие очереди? Какие выбрать для многопоточности?

Преимущества Blocking Queue

- **контроль доступа к общим ресурсам в многопоточной среде**
- **гарантия потокобезопасности** (объект является потокобезопасным thread-safe)

Преимущества NON-Blocking Queue

- отсутствие взаимоблокировок
- отсутствие инверсии приоритетов
- повышенная безопасность
- **выигрыш в производительности**

Не блокирующие concurrent очереди

Collections Framework

Всегда ли нужен synchronize?

- Как сделать такой код без синхронайзов, без локов, без семафоров, и при этом чтобы он был многопоточным и безопасным?

Существуют задачи, где локи не нужны и можно организовать безопасный доступ за счет использования не блокирующих алгоритмов

Преимущество неблокирующих алгоритмов — в лучшей масштабируемости по количеству процессоров, и скорость работы

Неблокирующие алгоритмы строятся на атомарных операциях, например, чтение-модификация-запись (RCU read-create-update) и самая значимая из них — сравнение с обменом (CAS).

Подход при котором не используют блокировки типа синхронизации, семафоров, локов…

Разделение доступа между потоками идёт за счёт атомарных операций и специальных, разработанных под конкретную задачу, механизмов блокировки.

Алгоритмы неблокирующей синхронизации

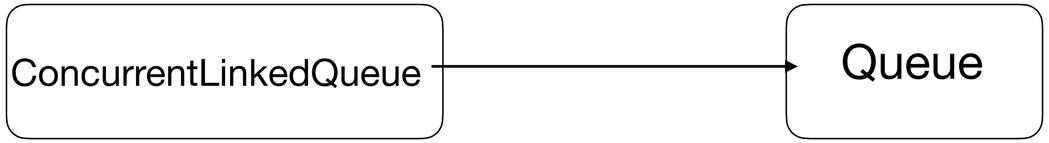
lock-free ***wait-free*** ***obstruction-free***

AtomicReferrence, AtomicInteger

Эти классы реализуют не блокирующие алгоритмы и позволяют выполнять операции получения и изменения каких то значений делать как одной атомарной операцией

При этом не блокировать разные потоки

На основании этих неблокорующих алгоритмов реализованы coccurrent коллекции



Класс *ConcurrentLinkedQueue* формирует неблокирующую, основанную на связанных узлах и ориентированную и на многопоточное исполнение очередь.

Очередь *ConcurrentLinkedQueue* использует эффективный неблокирующий алгоритм "без ожидания”(wait-free) предложенный Мэджедом М. Майклом и Майклом Л. Скоттом (Maged M. Michael, Michael L. Scott).

Wait-free

Любой поток всегда может завершить свою текущую операцию, независимо от состояния других потоков, обращающихся к очереди. Тривиальным примером структуры данных без ожидания является атомарный счетчик (в x86 он будет использовать инструкцию LOCK XADD), например AtomicInteger.

```java
public class WaitFree {
    2 usages
    private final AtomicInteger cnt = new AtomicInteger();
    new *
    public int add(int delta) {
        return cnt.addAndGet(delta);
    }
    new *
    public int get() {
        return cnt.get();
    }
}
```

loc-free

означает, что приложение в целом может развиваться независимо ни от чего. Таким образом, хотя отдельные потоки могут быть заблокированы, по крайней мере один из них будет добиваться прогресса. Тривиальным примером может быть тот же атомарный счетчик на основе цикла с операцией CAS.

```java
public class LockFree {
    1 usage
    private static final AtomicIntegerFieldUpdater<LockFree> updater =
            AtomicIntegerFieldUpdater.newUpdater(LockFree.class, fieldName: "cnt");
    3 usages
    private volatile int cnt;

    new *
    public int add(int delta) {
        int cur;
        do {
            cur = cnt;
        } while (!updater.compareAndSet( obj: this, cur,  update: cur + delta));
        return cur + delta;
    }
    new *
    public int get() {
        return cnt;
    }
}
```

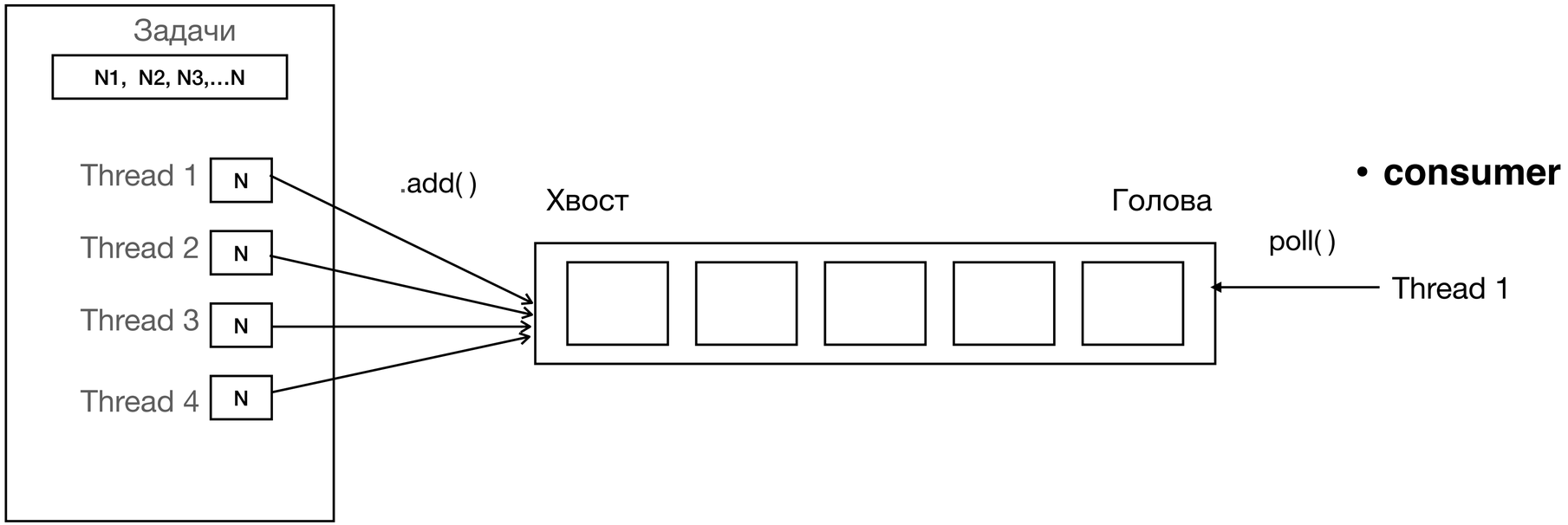
obstruction-free

означает, что поток может продвигаться вперед только в том случае, если нет конкуренции со стороны других потоков.

(https://puzpuzpuz.dev/buzzwordbusters-what-does-lock-free-wait-free-really-mean)

Producer-Consumer Queues

- **Producers**



Очереди реализуются с помощью таких структур данных как

Array **Stack** **LinckedList** **Heap**

Оптимизированные MPMC, MPSC, SPMC, and SPSC Queues

Все реализации потокобезопасных очередей, предоставляемые JDK, могут использоваться в средах с несколькими производителями и несколькими потребителями. Это означает, что один или несколько потоков записи и один или несколько потоков чтения могут одновременно обращаться к очередям JDK.

С помощью специальных механизмов можно оптимизировать очереди так, чтобы минимизировать накладные расходы на поддержание безопасности потоков при наличии ограничения на один поток чтения и/или один поток записи.

Соответственно, выделяют следующие четыре случая:

- Multi-producer-multi-consumer (MPMC)
- Multi-producer-single-consumer (MPSC)
- Single-producer-multi-consumer (SPMC)
- Single-producer-single-consumer (SPSC)

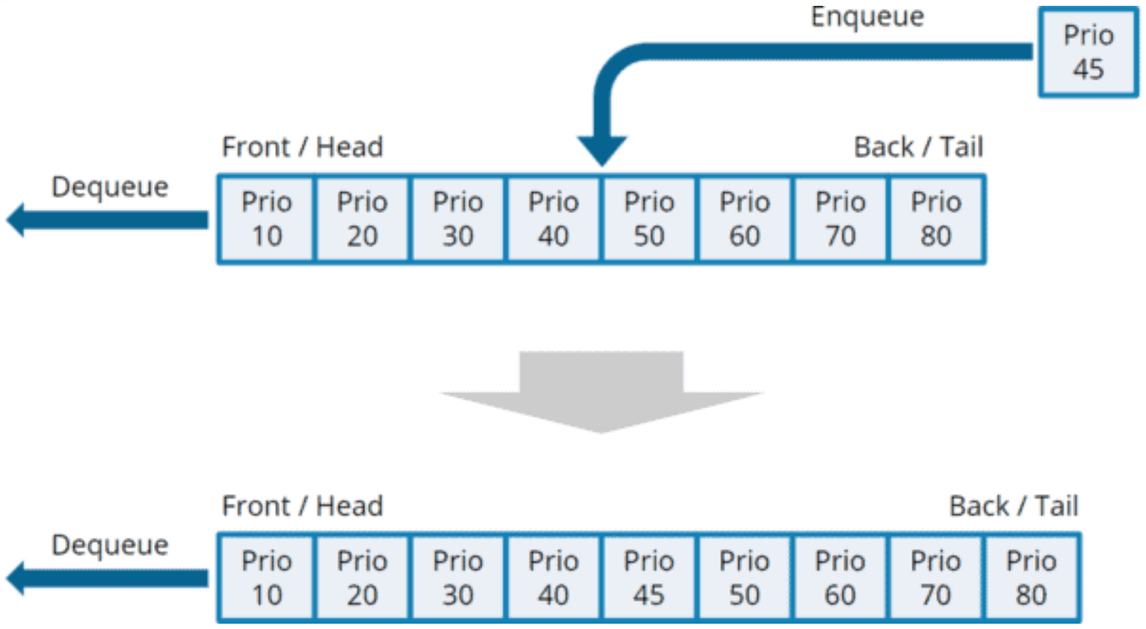
Библиотека JCTools обеспечивает высокооптимизированную реализацию очередей для всех четырех случаев.

PriorityQueue что это такое?

(Не потокобезопасная)

Приоритетная очередь не является очередью в классическом понимании. Причина в том, что элементы извлекаются не в порядке FIFO, а в соответствии с их приоритетом.

Элемент с наивысшим приоритетом всегда берется первым – независимо от того, когда он был вставлен в очередь.



Underlying data structure	Thread-safe?	Blocking/ non-blocking	Bounded/ unbounded	Iterator type
Min-heap (stored in an array)	No	Non-blocking	Unbounded	Fail-fast[2]

https://www.happycoders.eu/algorithms/priorityqueue-java/

В PriorityQueue порядок удаления из очереди определяется либо естественным порядком элементов (дженерик должен имплементировать интерфейс Comparable), либо в соответствии с компаратором, переданным конструктору.

Базовая структура данных представляет собой минимальную кучу, т. е. наименьший элемент всегда находится в начале очереди

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(25);
pq.add(30);
pq.add(20);
System.out.println(pq); // Вывод: [20, 30, 25]
```

```
PriorityQueue<Integer> pq2 = new PriorityQueue<>(Comparator.reverseOrder());
pq.add(25);pq.add(30);pq.add(20);
System.out.println(pq); // Вывод: [30, 25, 20]
```

ConcurrentLinkedQueue

(неблокирующая очередь, даже если очередь пуста, поток не будет заблокирован)

Класс java.util.concurrent.ConcurrentLinkedQueue основан на Linked list и, как и большинство реализаций очередей, является потокобезопасным.

ConcurrentLinkedQueue использует эффективный неблокирующий алгоритм “wait-free” предложенный Мэджедом М. Майклом и Майклом Л. Скоттом (Maged M. Michael, Michael L. Scott).

ConcurrentLinkedQueue — хороший выбор, когда требуется потокобезопасная, неблокирующая и неограниченная очередь.

- Использует метод CAS (compare and swap) который работает на основе атомарных инструкций, используемых в многопоточности для достижения синхронизации.

- идеальна, если требуется высокая производительность при работе с массой потоков-производителей и потоков-потребителей и строгое соблюдение порядка не является приоритетом

Пример

```
[Thread-0] queue.offer(617)         --> queue = [617]
[Thread-5]     queue.poll() =  617 --> queue = []
[Thread-1] queue.offer(751)         --> queue = [751]
[Thread-6]     queue.poll() =  751 --> queue = []
[Thread-4]     queue.poll() = null --> queue = []
[Thread-3] queue.offer(662)         --> queue = [662]
[Thread-2] queue.offer(720)         --> queue = [662, 720]
[Thread-0] queue.offer(537)         --> queue = [662, 720, 537]
[Thread-6]     queue.poll() =  662 --> queue = [720, 537]
[Thread-1] queue.offer(368)         --> queue = [720, 537, 368]
[Thread-4]     queue.poll() =  720 --> queue = [537, 368]
[Thread-5]     queue.poll() =  537 --> queue = [368]
[Thread-3] queue.offer(840)         --> queue = [368, 840]
[Thread-2] queue.offer( 48)         --> queue = [368, 840, 48]
[Thread-0] queue.offer(212)         --> queue = [368, 840, 48, 212]
[Thread-1] queue.offer(697)         --> queue = [368, 840, 48, 212, 697]
[Thread-6]     queue.poll() =  368 --> queue = [840, 48, 212, 697]
[Thread-4]     queue.poll() =  840 --> queue = [48, 212, 697]
```

Интерфейс BlockingQueue

Реализация данного интерфейса обеспечивает блокировку потока в двух случаях :

- при попытке получения элемента из пустой очереди
- при попытке размещения элемента в полной очереди.

если поток доступа заполнен(когда очередь ограничена) или становится пустым, очередь блокирует этот поток

Summary of BlockingQueue methods

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	*not applicable*	*not applicable*

`int`	**`drainTo`**`(`**`Collection`**`<? super E> c)` Removes all available elements from this queue and adds them to the given collection.
`int`	**`drainTo`**`(`**`Collection`**`<? super E> c, int maxElements)` Removes at most the given number of available elements from this queue and adds them to the given collection.

BlockingQueue

Summary of BlockingQueue methods

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	*not applicable*	*not applicable*

BlockingQueue методы

BlockingQueue.put()

`put()` метод вставляет элемент в очередь, если доступно место, если достигнут предел емкости очереди, метод блокируется до тех пор, пока не освободится место

BlockingQueue.offer() with Timeout

`offer()` метод вставляет элемент, если в очереди еще есть место. В противном случае метод ожидает указанное время. Если в течение этого времени место становится доступным, элемент вставляется, и метод возвращает значение true, но если время ожидания истекает, а пространство не освобождается, метод возвращает false.

BlockingQueue.take()

`take()` Этот метод берет элемент из начала очереди, при условии, что очередь не пуста. Если очередь пуста, **take**() блокируется до тех пор, пока элемент не станет доступным, а затем возвращает его.

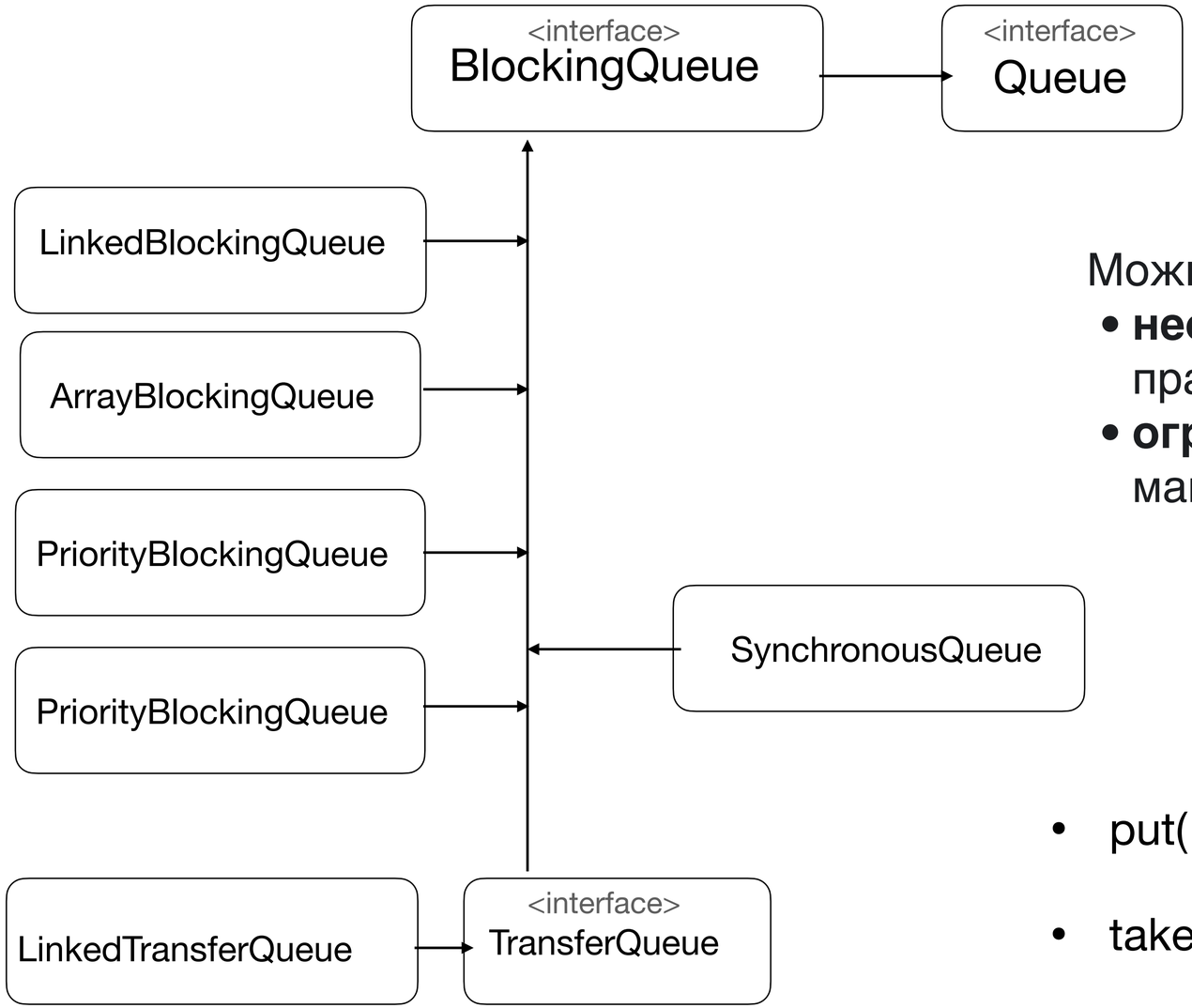
BlockingQueue.poll() with Timeout

`poll()` Этот метод берет элемент из головы очереди, если очередь не пуста. Если очередь пуста, метод ожидает указанное время. Если элемент становится доступным во время ожидания, он возвращается. Если время ожидания истекает без результата, метод возвращает значение null

Метод	Описание
boolean add(E e)	Немедленное добавление элемента в очередь, если это возможно; метод возвращает true при благополучном завершении операции, либо вызывает IllegalStateException, если очередь полная.
boolean contains(Object o)	Проверка наличия объекта в очереди; если объект найден в очереди метод вернет true.
boolean offer(E e)	Немедленное размещение элемента в очереди при наличие свободного места; метод вернет true при успешном завершении операции, в противном случае вернет false.
boolean offer(E e, long timeout, TimeUnit unit)	Размещение элемента в очереди при наличии свободного места; при необходимости определенное ожидание времени, пока не освободиться место.
E poll(long timeout, TimeUnit unit)	Чтение и удаление элемента из очереди в течение определенного времени (таймаута).
void put(E e)	Размещение элемента в очереди, ожидание при необходимости освобождения свободного места.
int remainingCapacity()	Получения количества элементов, которое можно разместить в очереди без блокировки, либо Integer.MAX_VALUE при отсутствии внутреннего предела.
boolean remove(Object o)	Удаление объекта из очереди, если он в ней присутствует.
E take()	Получение с удалением элемента из очереди, при необходимости ожидание пока элемент не станет доступным.

Блокирующие очереди

Collections Framework



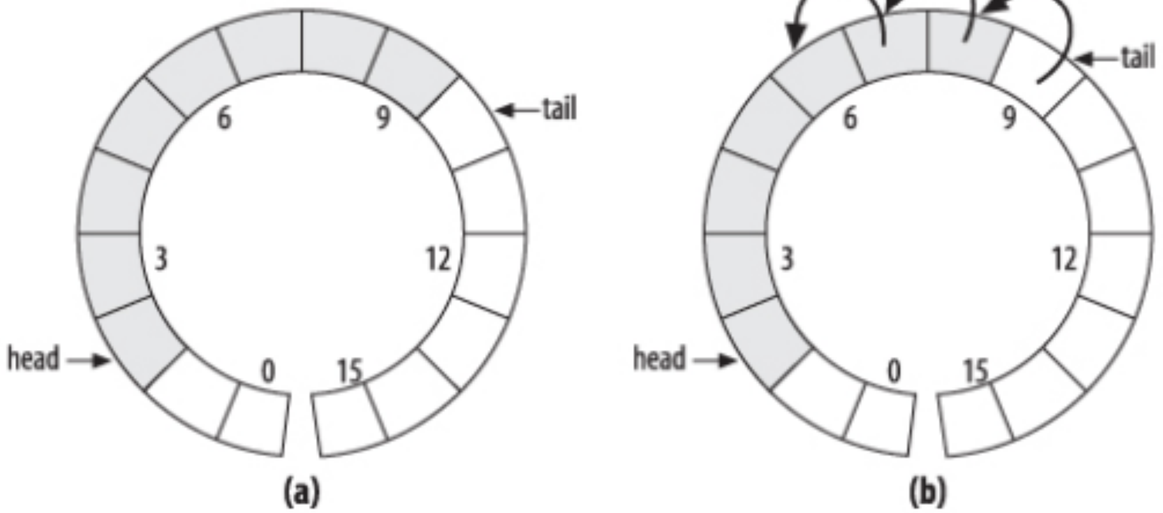
Можно выделить два типа **BlockingQueue** :

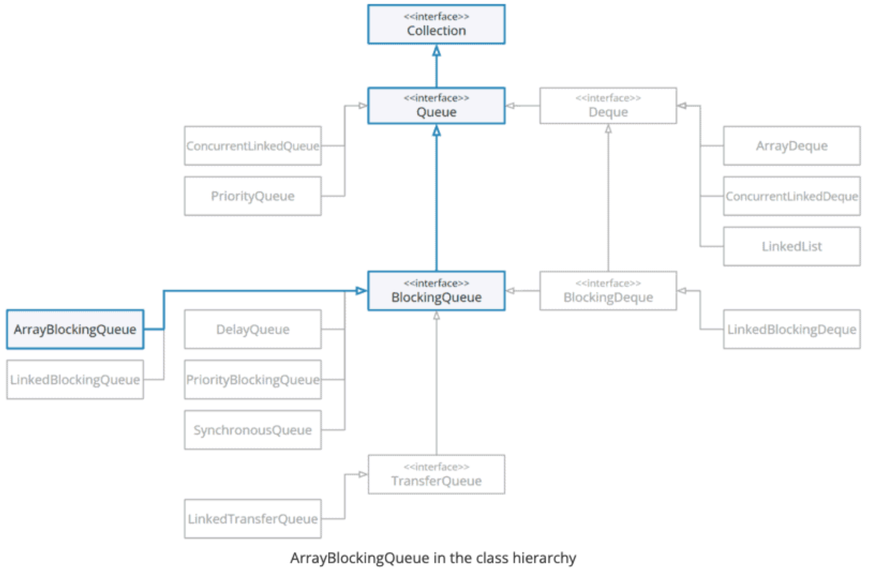
- **неограниченная очередь** – может расти практически бесконечно
- **ограниченная очередь** – с заданной максимальной емкостью

- put( ) - блокирует если нет места
- take( ) - блокирует, если пустая

Очередь ArrayBlockingQueue

(потокобезопасная с ограниченным размером (capacity) на основе ограниченного массива или как его называют кольцевой масив)





ArrayBlockingQueue in the class hierarchy

Достигнув конца массива, мы продолжаем с его начала. Это относится как к операциям вставки, так и к операциям удаления из очереди.

Класс блокирующей очереди, построенный на классическом кольцевом буфере. Помимо размера очереди, доступна возможность управлять «честностью» блокировок. Если fair=false (по умолчанию), то очередность работы потоков не гарантируется. Более подробно о «честности» можно посмотреть в описании ReentrantLock’a.
Но если очередь создана с «fair=true», реализация класса **ArrayBlockingQueue** предоставляет доступ потоков в порядке FIFO. Справедливость обычно уменьшает пропускную способность, но также снижает изменчивость и предупреждает исчерпание ресурсов.

AArrayBlockingQueue(int capacity)
ArrayBlockingQueue(int capacity, boolean fair)
ArrayBlockingQueue(int capacity, boolean fair,
Collection<? extends E> c)

```
[ 0,0s] [pool-1-thread-1 ]     Вызов queue.take() (queue = [])...
[ 3,0s] [pool-1-thread-6 ]     Вызов queue.take() (queue = [])...
[ 3,5s] [pool-1-thread-2 ] Вызов queue.put(0) (queue = [])...
[ 3,5s] [pool-1-thread-1 ]     queue.take() returned 0 (queue = [])
[ 3,5s] [pool-1-thread-2 ] queue.put(0) returned (queue = [])
[ 4,5s] [pool-1-thread-4 ] Вызов queue.put(1) (queue = [])...
[ 4,5s] [pool-1-thread-4 ] queue.put(1) returned (queue = [])
[ 4,5s] [pool-1-thread-6 ]     queue.take() returned 1 (queue = [])
[ 5,5s] [pool-1-thread-10] Вызов queue.put(2) (queue = [])...
[ 5,5s] [pool-1-thread-10] queue.put(2) returned (queue = [2])
[ 6,0s] [pool-1-thread-8 ]     Вызов queue.take() (queue = [2])...
[ 6,0s] [pool-1-thread-8 ]     queue.take() returned 2 (queue = [])
[ 6,5s] [pool-1-thread-3 ] Вызов queue.put(3) (queue = [])...
[ 6,5s] [pool-1-thread-3 ] queue.put(3) returned (queue = [3])
[ 7,5s] [pool-1-thread-7 ] Вызов queue.put(4) (queue = [3])...
[ 7,5s] [pool-1-thread-7 ] queue.put(4) returned (queue = [3, 4])
[ 8,5s] [pool-1-thread-5 ] Вызов queue.put(5) (queue = [3, 4])...
[ 8,5s] [pool-1-thread-5 ] queue.put(5) returned (queue = [3, 4, 5])
```

Два потока консьюмера уже должны быть заблокированы и ожидают записи элементов в очередь.

Поскольку мы заполняем очередь быстрее, чем читаем, очередь вскоре должна достичь предела своей емкости. С этого момента потоки producers должны блокироваться до тех пор, пока потоки чтения не догонят его.

Очередь LinkedBlockingQueue (потокобезопасная)

LinkedBlockingQueue()

LinkedBlockingQueue(int capacity)

LinkedBlockingQueue(Collection<? extends E> c)

"У соединенных на узлах очереди обычно более высокая пропускная способность, чем у основанной на массиве очереди, но менее предсказуемая производительность в большинстве многопоточных приложений."

Первый конструктор создает пустую очередь фиксированной емкости.

Второй конструктор создает очередь с фиксированной емкостью capacity.

Третий конструктор создает очередь с набором элементов. Если используется конструктор без указания емкости очереди, то используется значение по умолчанию Integer.MAX_VALUE.

Underlying data structure	Thread-safe?	Blocking/ non-blocking	Fairness policy	Bounded/ unbounded	Iterator type
Linked list	Yes (pessimistic locking with two locks)	Blocking	Not available	Bounded	Weakly consistent[1]

Пример LinckedBlockingQueue

```
[ 0.0s] [pool-1-thread-1 ]     Calling queue.take() (queue = [])...
[ 3.0s] [pool-1-thread-2 ]     Calling queue.take() (queue = [])...
[ 3.6s] [pool-1-thread-5 ] Calling queue.put(0) (queue = [])...
[ 3.6s] [pool-1-thread-5 ] queue.put(0) returned (queue = [])
[ 3.6s] [pool-1-thread-1 ]     queue.take() returned 0 (queue = [])
[ 4.6s] [pool-1-thread-3 ] Calling queue.put(1) (queue = [])...
[ 4.6s] [pool-1-thread-3 ] queue.put(1) returned (queue = [1])
[ 4.6s] [pool-1-thread-2 ]     queue.take() returned 1 (queue = [])
[ 5.6s] [pool-1-thread-10] Calling queue.put(2) (queue = [])...
[ 5.6s] [pool-1-thread-10] queue.put(2) returned (queue = [2])
[ 6.0s] [pool-1-thread-9 ]     Calling queue.take() (queue = [2])...
[ 6.0s] [pool-1-thread-9 ]     queue.take() returned 2 (queue = [])
[ 6.6s] [pool-1-thread-7 ] Calling queue.put(3) (queue = [])...
[ 6.6s] [pool-1-thread-7 ] queue.put(3) returned (queue = [3])
[ 7.6s] [pool-1-thread-6 ] Calling queue.put(4) (queue = [3])...
[ 7.6s] [pool-1-thread-6 ] queue.put(4) returned (queue = [3, 4])
[ 8.6s] [pool-1-thread-4 ] Calling queue.put(5) (queue = [3, 4])...
[ 8.6s] [pool-1-thread-4 ] queue.put(5) returned (queue = [3, 4, 5])
[ 9.0s] [pool-1-thread-8 ]     Calling queue.take() (queue = [3, 4, 5])
[ 9.0s] [pool-1-thread-8 ]     queue.take() returned 3 (queue = [4, 5])
[ 9.6s] [pool-1-thread-5 ] Calling queue.put(6) (queue = [4, 5])...
[ 9.6s] [pool-1-thread-5 ] queue.put(6) returned (queue = [4, 5, 6])
[10.6s] [pool-1-thread-1 ] Calling queue.put(7) (queue = [4, 5, 6])...
```

Поскольку мы начинаем писать только после того, как два потока уже вызвали take(), эти первые две попытки чтения блокируются через 0,0 и 3,0 с (потоки 1 и 2).

Через 3,5 с записывается первый элемент (поток 5). Это пробуждает поток 1, и метод take( ) немедленно снова удаляет этот элемент из очереди.

Через 4,5 с записывается второй элемент (поток 3). Поток 2 просыпается и снова берет этот элемент из очереди.

Программа пишет быстрее, чем читает. Через 10,5 с поток записи (поток 1) впервые блокируется при попытке записать 7 в очередь, которая в это время заполнена. Через 11,5 с поток 4 также блокирует попытку записи 8 в очередь. …

Потокобезопасность **LinkedBlockingQueue** гарантируется блокировкой с использованием двух отдельных ReentrantLocks для операций записи и чтения. Это предотвращает конфликты (конфликты доступа) между потоками producer и consumer.

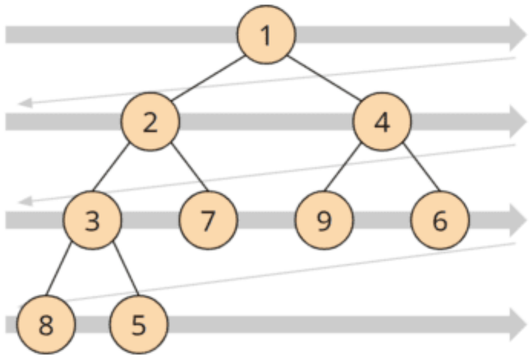
```
Lock held by take, poll, etc
private final ReentrantLock takeLock = new ReentrantLock();
Lock held by put, offer, etc
private final ReentrantLock putLock = new ReentrantLock();
```

Отличия от других очередей:

В **ConcurrentLinkedQueue** безопасность потоков обеспечивается за счет блокировки посредством сравнения и установки, что приводит к повышению производительности при низкой и умеренной конкуренции.

ArrayBlockingQueue защищен только одним ReentrantLock, поэтому возможны конфликты доступа между потоками producer и consumer.

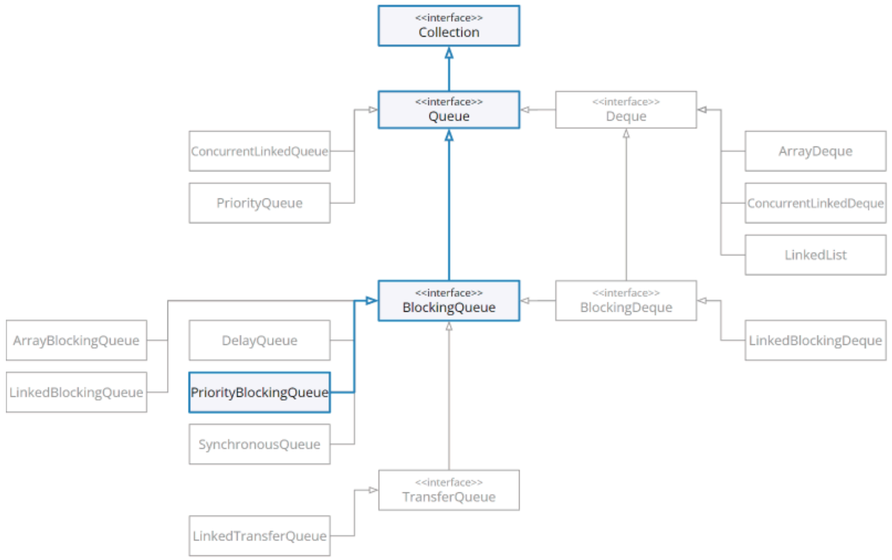
Очередь PriorityBlockingQueue (потокобезопасная с не ограниченным размером создана на основе дерева (min-heap))



Mapping a min-heap to an array

“элемент с наивысшим приоритетом всегда упорядочивался первым”

“Дерево в котором каждый узел больше, либо равен своим дочерним элементам («max heap»), меньше или равен своим дочерним элементам («min heap»).



PriorityBlockingQueue in the class hierarchy

- 1 меньше чем 2 и 4;
- 2 меньше чем 3 и 7;
- 4 i меньше чем 9 и 6;
- 3 меньше чем n 8 и 5.

Основные тезисы

Неограниченная очередь, в которой элементы упорядочены по приоритету

Хоть эта очередь логически неограничена, попытки добавления могут завершиться неудачей из-за исчерпания ресурсов (вызывая OutOfMemoryError).
Этот класс не допускает null элементов.

Как и в случае с PriorityQueue, элементы хранятся в массиве, представляющем минимальную кучу.
Итератор перебирает элементы в соответствующем порядке.

Одиночный ReentrantLock обеспечивает безопасность потоков.

PriorityBlockingQueue реализует интерфейс BlockingQueue , который дает нам несколько дополнительных методов, позволяющих **блокироваться при удалении из пустой очереди** .

В отличие от стандартной очереди, вы не можете просто добавить элемент любого типа в PriorityBlockingQueue. Есть два варианта:

1. Добавление элементов, реализующих Comparable

2. Добавление элементов, которые не реализуют Comparable , при условии, что вы также предоставляете Comparator

```
[ 0.0s] [pool-1-thread-1 ]     Calling queue.take() (queue = [])...
[ 3.0s] [pool-1-thread-2 ]     Calling queue.take() (queue = [])...
[ 3.5s] [pool-1-thread-6 ] Calling queue.put(56) (queue = [])...
[ 3.5s] [pool-1-thread-6 ] queue.put(56) returned (queue = [])
[ 3.5s] [pool-1-thread-1 ]     queue.take() returned 56 (queue = [])
[ 4.5s] [pool-1-thread-9 ] Calling queue.put(83) (queue = [])...
[ 4.5s] [pool-1-thread-9 ] queue.put(83) returned (queue = [])
[ 4.5s] [pool-1-thread-2 ]     queue.take() returned 83 (queue = [])
[ 5.5s] [pool-1-thread-10] Calling queue.put(88) (queue = [])...
[ 5.5s] [pool-1-thread-10] queue.put(88) returned (queue = [88])
[ 6.0s] [pool-1-thread-4 ]     Calling queue.take() (queue = [88])...
[ 6.0s] [pool-1-thread-4 ]     queue.take() returned 88 (queue = [])
[ 6.5s] [pool-1-thread-8 ] Calling queue.put(99) (queue = [])...
[ 6.5s] [pool-1-thread-8 ] queue.put(99) returned (queue = [99])
[ 7.5s] [pool-1-thread-3 ] Calling queue.put(43) (queue = [99])...
[ 7.5s] [pool-1-thread-3 ] queue.put(43) returned (queue = [43, 99])
[ 8.5s] [pool-1-thread-7 ] Calling queue.put(62) (queue = [43, 99])...
[ 8.5s] [pool-1-thread-7 ] queue.put(62) returned (queue = [43, 99, 62])
[ 9.0s] [pool-1-thread-5 ]     Calling queue.take() (queue = [43, 99, 62])...
[ 9.0s] [pool-1-thread-5 ]     queue.take() returned 43 (queue = [62, 99])
[ 9.5s] [pool-1-thread-6 ] Calling queue.put(45) (queue = [62, 99])...
[ 9.5s] [pool-1-thread-6 ] queue.put(45) returned (queue = [45, 99, 62])
```

У нас есть очередь, которая блокируется, ожидая добавления элементов. Затем мы добавляем множество элементов одновременно, а затем показываем, что они будут обрабатываться в порядке приоритета.

Очередь DelayQueue (потокобезопасная с не ограниченным размером)

“когда потребитель хочет взять элемент из очереди, он может взять его только тогда, когда истечет задержка для этого конкретного элемента”

```
public class DelayQueue<E extends Delayed> extends AbstractQueue<E>
        implements BlockingQueue<E> {
```

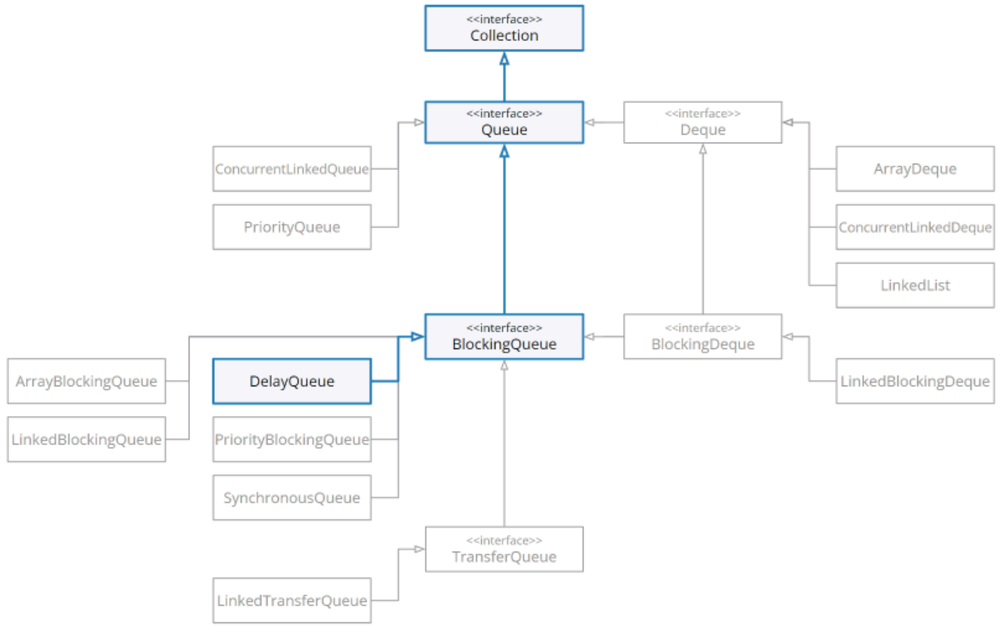
```
interface Delayed extends Comparable<Delayed> {
    long getDelay(TimeUnit unit);
}
```

Он не удаляет элемент, который находился в очереди дольше всего. Вместо этого элемент может быть взят, когда время ожидания («задержка»), назначенное этому элементу, истекло.

Следовательно, элементы должны реализовывать интерфейс **java.util.concurrent.Delayed** и его метод **getDelay()**. Этот метод **возвращает оставшееся время ожидания, которое должно пройти, прежде чем элемент можно будет удалить из очереди.**

DelayQueue является блокирующим, но неограниченным, поэтому он может содержать любое количество элементов и блокируется только при удалении (пока не истечет время ожидания), а не при вставке.

Безопасность потоков достигается за счет блокировки с помощью одного ReentrantLock.



DelayQueue in the class hierarchy

```
элементы хранятся в приоритетной очереди, отсортированной по запланированному времени (самый ранний срок действия находится в начале очереди).
```

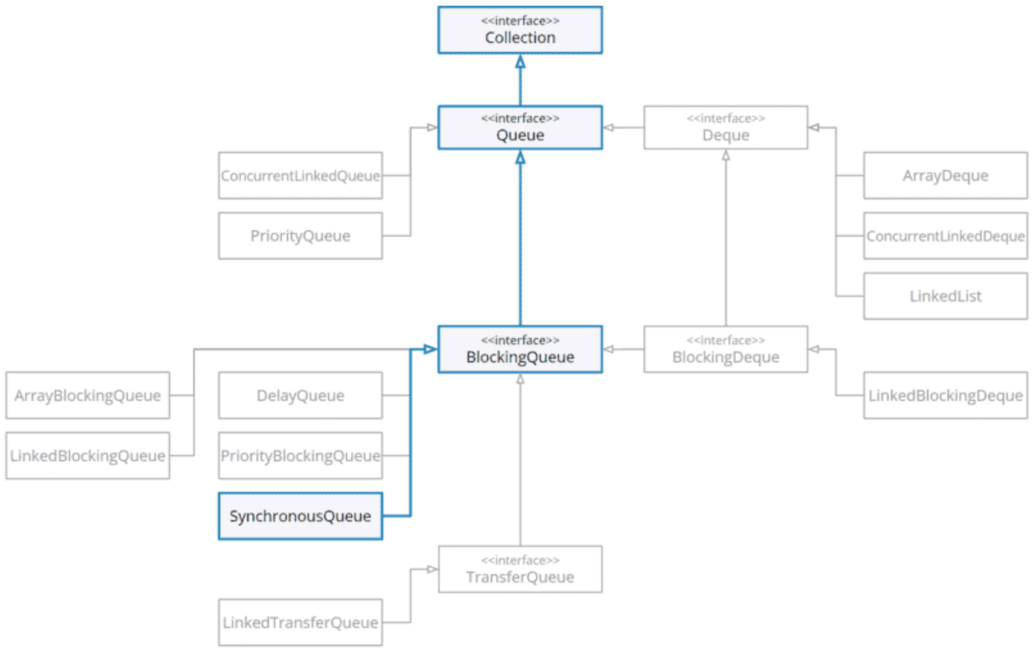
```
[  3ms] queue.offer({30, 879ms})   --> queue = [{30, 879ms}]
[ 68ms] queue.offer({83, 409ms})   --> queue = [{83, 409ms}, {30, 879ms}]
[ 69ms] queue.offer({72, 729ms})   --> queue = [{83, 409ms}, {30, 879ms}, {72, 729ms}]
[ 70ms] queue.offer({89, 869ms})   --> queue = [{83, 409ms}, {30, 879ms}, {72, 729ms}, {89, 869ms}]
[ 72ms] queue.offer({14, 615ms})   --> queue = [{83, 409ms}, {14, 615ms}, {72, 729ms}, {89, 869ms}, {30, 879ms}]
[ 73ms] queue.offer({13, 368ms})   --> queue = [{13, 368ms}, {14, 615ms}, {83, 409ms}, {89, 869ms}, {30, 879ms}, {72, 729ms}]
[ 73ms] queue.offer({10, 556ms})   --> queue = [{13, 368ms}, {14, 615ms}, {83, 409ms}, {89, 869ms}, {30, 879ms}, {72, 729ms}, {10, 556ms}]
[444ms] queue.poll() = {13, 368ms} --> queue = [{83, 409ms}, {14, 615ms}, {10, 556ms}, {89, 869ms}, {30, 879ms}, {72, 729ms}]
[478ms] queue.poll() = {83, 409ms} --> queue = [{10, 556ms}, {14, 615ms}, {72, 729ms}, {89, 869ms}, {30, 879ms}]
[633ms] queue.poll() = {10, 556ms} --> queue = [{14, 615ms}, {30, 879ms}, {72, 729ms}, {89, 869ms}]
[687ms] queue.poll() = {14, 615ms} --> queue = [{72, 729ms}, {30, 879ms}, {89, 869ms}]
[798ms] queue.poll() = {72, 729ms} --> queue = [{30, 879ms}, {89, 869ms}]
[882ms] queue.poll() = {30, 879ms} --> queue = [{89, 869ms}]
[939ms] queue.poll() = {89, 869ms} --> queue = []
```

Можно заметить, что очередь не сортируется, но элемент с наименьшим временем ожидания всегда находится в начале (слева) и что элементы берутся (приблизительно) после истечения соответствующего времени ожидания:

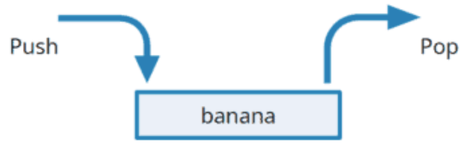
если в очередь вставить два элемента с одинаковым временем ожидания, они будут удалены в случайном порядке относительно друг друга.

Очередь SynchronousQueue (потокобезопасная, блокирующая реализована на основе Stack ())

Эта очередь работает по принципу **один вошел, один вышел**. Каждая операция вставки блокирует «Producer» поток до тех пор, пока «Consumer» поток не вытащит элемент из очереди и наоборот, «Consumer» будет ждать пока «Producer» не вставит элемент



SynchronousQueue in the class hierarchy



Underlying data structure	Thread-safe?	Blocking/ non-blocking	Fairness policy	Bounded/ unbounded	Iterator type
Stack (implemented with a linked list)	Yes (optimistic locking through compare-and-set)	Blocking	Optional	Unbounded	The iterator is always empty.

```
[Thread-0 ] Вызов queue.put(0) (queue = [])...
[Thread-1 ] Вызов queue.put(1) (queue = [])...
[Thread-2 ] Вызов queue.put(2) (queue = [])...
[Thread-3 ]     Вызов queue.take() (queue = [])...
[Thread-3 ]     queue.take() returned 2 (queue = [])
[Thread-2 ] queue.put(2) returned (queue = [])
[Thread-4 ]     Вызов queue.take() (queue = [])...
[Thread-4 ]     queue.take() returned 1 (queue = [])
[Thread-1 ] queue.put(1) returned (queue = [])
[Thread-5 ]     Вызов queue.take() (queue = [])...
[Thread-5 ]     queue.take() returned 0 (queue = [])
[Thread-0 ] queue.put(0) returned (queue = [])
[Thread-6 ]     Вызов queue.take() (queue = [])...
[Thread-7 ]     Вызов queue.take() (queue = [])...
[Thread-8 ]     Вызов queue.take() (queue = [])...
[Thread-9 ] Вызов queue.put(3) (queue = [])...
[Thread-9 ] queue.put(3) returned (queue = [])
```

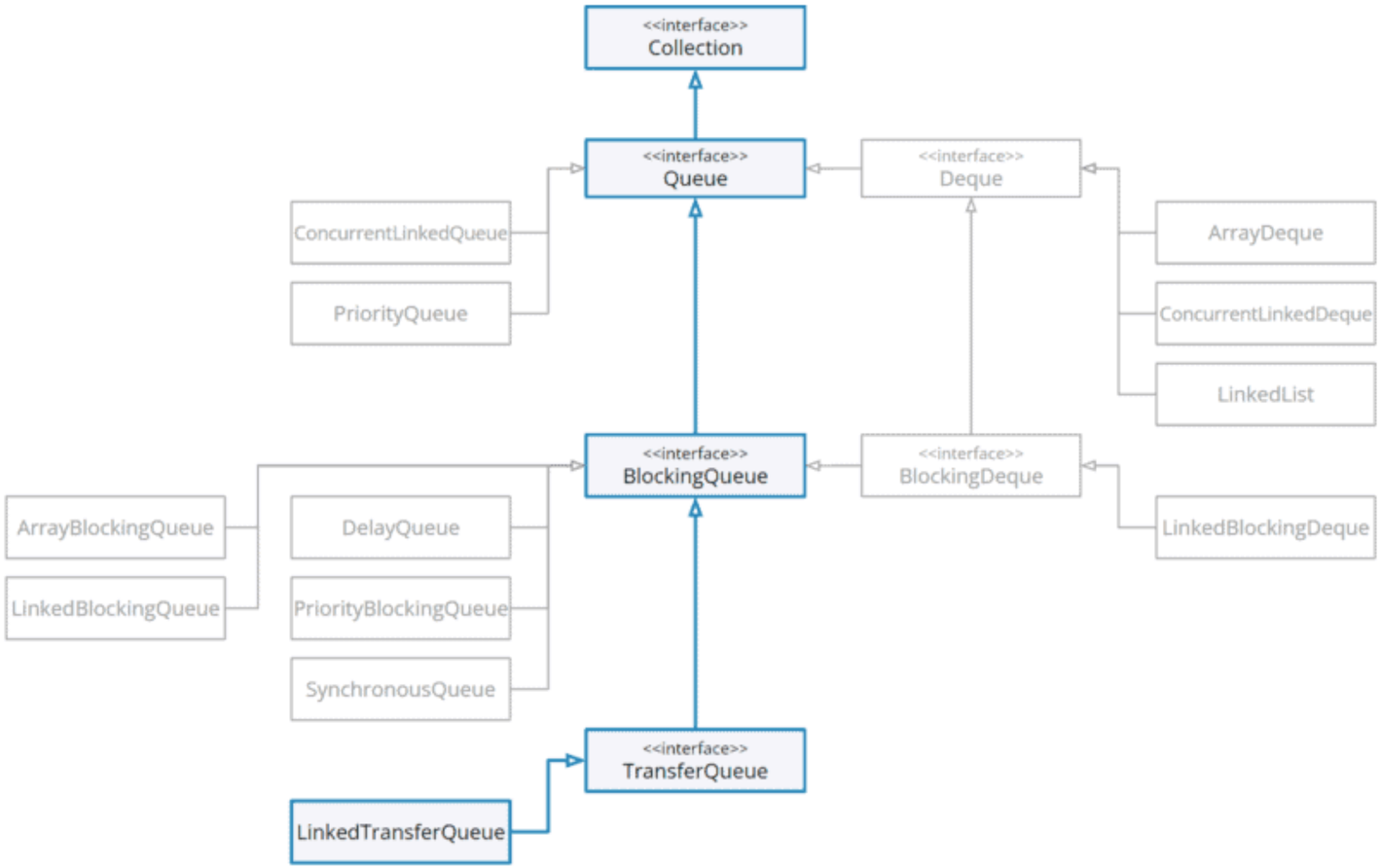
Вывод показывает, как первые три вызова put() (потоками 0, 1 и 2) блокируются до тех пор, пока вставленные элементы не будут извлечены с помощью take() (потоками 3, 4 и 5) в обратном порядке.

После этого три следующих вызова take() (потоки 6, 7, 8) блокируются до тех пор, пока в очередь не будут записаны еще три элемента с помощью put() (потоки 9, 10, 11).

Очередь все время остается пустой

Если установить для константы FAIR значение true, то все элементы берутся в порядке FIFO, а не в порядке LIFO.

Очередь LinkedTransferQueue (потокобезопасная, блокирующая)



LinkedTransferQueue in the class hierarchy

TransferQueue

Данный интерфейс может быть интересен тем, что при добавлении элемента в очередь существует возможность заблокировать вставляющий «Producer» поток до тех пор, пока другой поток «Consumer» не вытащит элемент из очереди. Блокировка может быть как с таймаутом, так и вовсе может быть заменена проверкой на наличие ожидающих «Consumer»ов. Тем самым появляется возможность реализации механизма передачи сообщений с поддержкой как синхронных, так и асинхронных сообщений.

Реализация TransferQueue на основе алгоритма Dual Queues with Slack. Активно использует CAS и потоки, когда они находятся в режиме ожидания.

transfer(E e) – передает элемент в поток, который ожидает элемент с помощью take() или **poll**(). Если такого потока не существует, метод блокируется до тех пор, пока другой поток не вызовет take() или poll().

```
[Thread-0] Calling queue.transfer(1)...
[main    ]                              --> queue = [1]
[Thread-1] Calling queue.transfer(2)...
[main    ]                              --> queue = [1, 2]
[main    ] Calling queue.put(3)...
[main    ] queue.put(3) returned        --> queue = [1, 2, 3]
[Thread-2] Calling queue.transfer(4)...
[main    ]                              --> queue = [1, 2, 3, 4]
[Thread-3] Calling queue.transfer(5)...
[main    ]                              --> queue = [1, 2, 3, 4, 5]
[main    ]     Calling queue.take() (queue = [1, 2, 3, 4, 5])...
[main    ]     queue.take() returned 1 --> queue = [2, 3, 4, 5]
[Thread-0] queue.transfer(1) returned   --> queue = [2, 3, 4, 5]
[main    ]     Calling queue.take() (queue = [2, 3, 4, 5])...
[main    ]     queue.take() returned 2 --> queue = [3, 4, 5]
[Thread-1] queue.transfer(2) returned   --> queue = [3, 4, 5]
[main    ]     Calling queue.take() (queue = [3, 4, 5])...
[main    ]     queue.take() returned 3 --> queue = [4, 5]
[main    ]     Calling queue.take() (queue = [4, 5])...
[main    ]     queue.take() returned 4 --> queue = [5]
[Thread-2] queue.transfer(4) returned   --> queue = [5]
[main    ]     Calling queue.take() (queue = [5])...
[main    ]     queue.take() returned 5 --> queue = []
[Thread-3] queue.transfer(5) returned   --> queue = []
```

вначале transfer() вызывается дважды (но не возвращает), как затем put() вызывается один раз (и возвращает) затем transfer() вызывается еще два раза (и не возвращает результат).

После этого мы видим, как берется первый элемент, а затем возвращается и transfer(1).

Затем берется второй элемент и возвращается функция transfer(2).

Удаление 3 не приводит к каким-либо дальнейшим действиям, поскольку оно было записано в очередь с помощью put().

После удаления 4 и 5 вы снова можете увидеть, как возвращается соответствующий вызов transfer().

Так какую очередь выбрать?

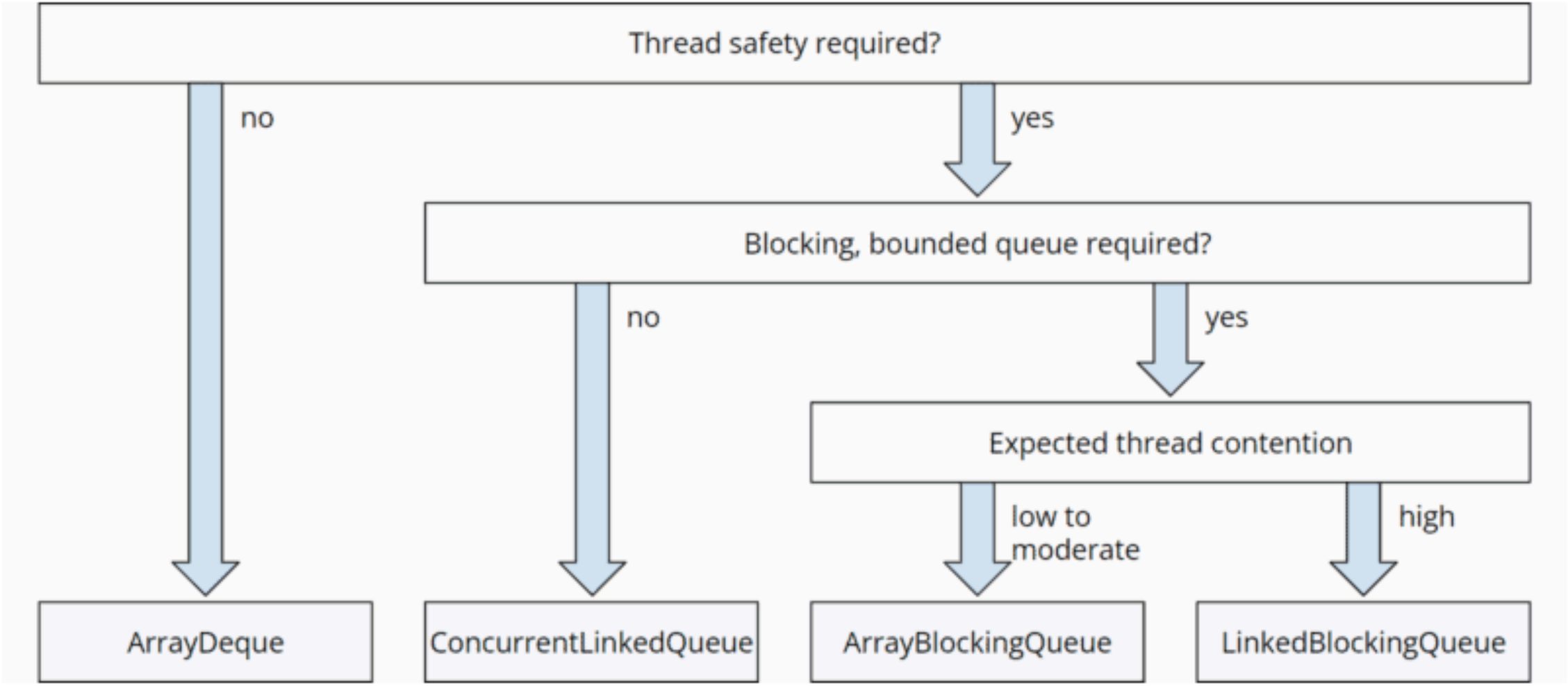
Class	Base data structure	Thread-safe?	Blocking/ non-blocking	Fairness policy	Bounded/ unbounded	Iterator type
ConcurrentLinkedQueue	Linked list	Yes (optimistic locking through compare-and-set)	Non-blocking	—	Unbounded	Weakly consistent[1]
PriorityQueue	Min-heap (stored in an array)	No	Non-blocking	—	Unbounded	Fail-fast[2]
LinkedBlockingQueue	Linked list	Yes (pessimistic locking with two locks)	Blocking	Not available	Bounded	Weakly consistent[1]
ArrayBlockingQueue	Array	Yes (pessimistic locking with one lock)	Blocking	Optional	Bounded	Weakly consistent[1]
PriorityBlockingQueue	Min-heap (stored in an array)	Yes (pessimistic locking with one lock)	Blocking (nur dequeue)	Not available	Unbounded	Weakly consistent[1]
DelayQueue	Priority queue	Yes (pessimistic locking with one lock)	Blocking (nur dequeue)	Not available	Unbounded	Weakly consistent[1]
SynchronousQueue	Stack (implemented with a linked list)	Yes (optimistic locking through compare-and-set)	Blocking	Optional	Unbounded	The iterator is always empty.
LinkedTransferQueue	Linked list	Yes (optimistic locking through compare-and-set)	Blocking (only transfer and dequeue)	Not available	Unbounded	Weakly consistent[1]

Weakly consistent:: все элементы, существующие на момент создания итератора, проходятся им ровно один раз. Изменения, которые происходят после этого, могут, но не обязательно, отражаться итератором.
Fail-fast:Итератор выдает исключение ConcurrentModificationException, если элементы добавляются в очередь или удаляются из нее во время итерации.

Когда следует использовать какую реализацию очереди?
***Рекомендации**

ArrayDeque для однопоточных приложений.
ConcurrentLinkedQueue как потокобезопасная, неблокирующая и неограниченная очередь.
ArrayBlockingQueue как потокобезопасная, блокирующая, ограниченная очередь, если ожидается низкий или средний уровень конкуренции между потоками производителя и потребителя.
LinkedBlockingQueue как потокобезопасная, блокирующая, ограниченная очередь, если вы ожидаете высокой конкуренции между потоками производителя и потребителя (лучше всего проверить, какая реализация более производительна для вашего варианта использования).
Обязательно делайте тесты на производительность для своих задач!

*Рекомендации



Decision tree Java Queue implementations

Источники

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent
https://for-each.dev/
https://sky.pro/wiki/java/
https://www.happycoders.eu/
https://howtodoinjava.com/java/collections/
https://deepakvadgama.com/blog/
https://java-online.ru/
https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%B1%D0%BB%D0%BE%D0%BA%D0%B8%D1%80%D1%83%D1%8E%D1%89%D0%B0%D1%8F_%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F
https://java-online.ru/concurrent-queue-noblock.xhtml
https://java-online.ru/concurrent-queue-block.xhtml
https://jug-lviv.blogspot.com/2013/05/javautilconcurrent.html
https://senior.ua/articles/mnogopotochnost-v-java-lekciya-3-blokirovki-i-klassy-sinhronizacii-potokov
https://www.youtube.com/watch?v=g7ynfDFoCL4
https://www.youtube.com/watch?v=t0psp84lyms
https://stackoverflow.com/questions/7316810/linkedlist-vs-concurrentlinkedqueue
Книга: Maurice Naftalin and Philip Wadler Java Generics and Collections
Книга: Брайман Гетц и др. Java Concurrency