Java Pro 6.4

TEL-RAN
by Starta Institute

Атомарные типы. Блокирующие и неблокирующие очереди.

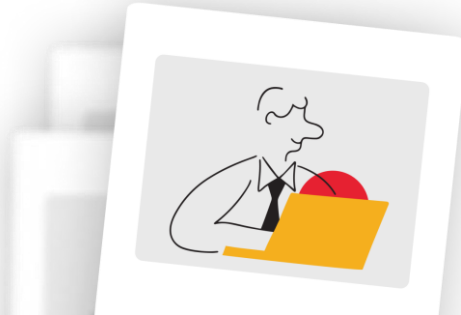
ПРЕПОДАВАТЕЛЬ

TEL-RAN
by Starta Institute

Юрий Костяной

Java/Kotlin backend-разработчик

- 3+ года опыта в коммерческой разработке
- 2+ года опыта в преподавании
- Проекты по интеграции сторонних платформ, CRM
- Проблемно-ориентированный подход в преподавании



ВАЖНО:

- Камера должна быть включена на протяжении всего занятия.
- Если у Вас возник вопрос в процессе занятия, пожалуйста, поднимите руку и дождитесь, пока преподаватель закончит мысль и спросит Вас, также можно задать вопрос в чате или когда преподаватель скажет, что начался блок вопросов.
- Организационные вопросы по обучению решаются с кураторами, а не на тематических занятиях.
- Вести себя уважительно и этично по отношению к остальным участникам занятия.
- Во время занятия будут интерактивные задания, будьте готовы включить камеру или демонстрацию экрана по просьбе преподавателя.

TEL-RAN
by Starta Institute

ПЛАН ЗАНЯТИЯ

1. Повторение
2. Основной блок
3. Вопросы по основному блоку
4. Домашняя работа

TEL-RAN
by Starta Institute

2 ОСНОВНОЙ БЛОК

Введение

TEL-RAN
by Starta Institute

- Неделимый
- Держать очередь

Проблема

TEL-RAN
by Starta Institute

Мы уже знаем, что типы *long* и *double* являются неатомарными из-за своего размера. Но и многие операции с другими типами не являются атомарными (неделимыми). Например, инкремент по своей сути представляет набор операций – взять текущее значение переменной, добавить к нему 1 и снова записать в переменную. Хотя в однопоточной программе мы рассматривали оператор инкремента как неделимый, при его вызове в многопоточной программе может произойти условие, что один поток взял текущее значение переменной, увеличил его, но в это время второй поток уже выполнил инкремент с той же переменной. В итоге первый поток перезапишет неактуальное значение в переменную.

Как решить проблему атомарности операций?

Неделимый

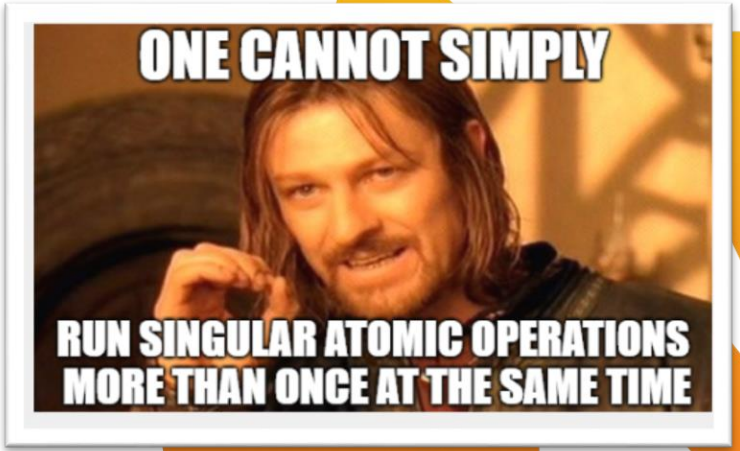
Атомарные классы

TEL-RAN
by Starta Institute

С греческого *atom* переводится как «неделимый».

Атомарные классы – классы пакета *java.util.concurrent.atomic* для выполнения атомарных операций. Операция является атомарной, если её можно безопасно выполнять при параллельных вычислениях в нескольких потоках, не используя при этом ни блокировок, ни синхронизацию *synchronized*.

Атомарный класс включает метод *compareAndSet*, реализующий механизм *оптимистичной блокировки* и позволяющий изменить значение только в том случае, если оно равно ожидаемому значению. Т.е. если значение атомарного класса было изменено в другом потоке, то оно не будет равно ожидаемому значению, и метод *compareAndSet* не позволит изменить значение.



Неделимый

Атомарные классы

TEL-RAN
by Starta Institute

Ряд архитектур процессоров имеют инструкцию *Compare-And-Swap (CAS)*, которая реализует операцию *compareAndSet*. Таким образом, на уровне инструкций процессора имеется поддержка необходимой атомарной операции. В архитектурах процессоров, где инструкция не поддерживается, операции реализованы иными низкоуровневыми средствами.

Неделимый

Пример устройства атомарного класса

В *AtomicLong* переменная *value* объявлена с модификатором *volatile*, т.е. её значение могут поменять разные потоки одновременно.

Модификатор *volatile* гарантирует выполнение отношения *happens-before*, что ведет к тому, что измененное значение этой переменной увидят все потоки.

Каждый атомарный класс включает метод *compareAndSet*, представляющий механизм *оптимистичной блокировки* и позволяющий изменить значение *value* только в том случае, если оно равно ожидаемому значению (т.е. *current)*

```
private volatile long value;

public final long get() {
    return value;
}

public final long getAndAdd(long delta) {
    while (true) {
        long current = get();
        long next = current + delta;
        if (compareAndSet(current, next))
            return current;
    }
}
```

Устройство атомарного класса *AtomicLong*

Неделимый

Пример устройства атомарного класса

Если значение *value* было изменено в другом потоке, то оно не будет равно ожидаемому значению. Следовательно, метод *compareAndSet* вернет значение *false*, что приведет к новой итерации цикла *while* в методе *getAndAdd*. Таким образом, в очередном цикле в переменную *current* будет считано обновленное значение *value*, после чего будет выполнено сложение и новая попытка записи получившегося значения (т.е. *next*). Переменные *current* и *next* - локальные, и, следовательно, у каждого потока свои экземпляры этих переменных.

```
private volatile long value;

public final long get() {
    return value;
}

public final long getAndAdd(long delta) {
    while (true) {
        long current = get();
        long next = current + delta;
        if (compareAndSet(current, next))
            return current;
    }
}
```

Устройство атомарного класса *AtomicLong*

Неделимый

Пример устройства атомарного класса

Если значение *value* было изменено в другом потоке, то оно не будет равно ожидаемому значению. Следовательно, метод *compareAndSet* вернет значение *false*, что приведет к новой итерации цикла *while* в методе *getAndAdd*. Таким образом, в очередном цикле в переменную *current* будет считано обновленное значение *value*, после чего будет выполнено сложение и новая попытка записи получившегося значения (т.е. *next*). Переменные *current* и *next* - локальные, и, следовательно, у каждого потока свои экземпляры этих переменных.

```
private volatile long value;

public final long get() {
    return value;
}

public final long getAndAdd(long delta) {
    while (true) {
        long current = get();
        long next = current + delta;
        if (compareAndSet(current, next))
            return current;
    }
}
```

Устройство атомарного класса *AtomicLong*

Неделимый

Пример устройства атомарного класса

Основная выгода от атомарных (*CAS*) операций появляется только при условии, когда переключать контекст процессора с потока на поток становится менее выгодно, чем немного покрутиться в цикле *while*, выполняя *метод boolean compareAndSwap(oldValue, newValue)*. Если время, потраченное в этом цикле, превышает 1 квант потока, то, с точки зрения производительности, может быть невыгодно использовать атомарные переменные.

```
private volatile long value;

public final long get() {
    return value;
}

public final long getAndAdd(long delta) {
    while (true) {
        long current = get();
        long next = current + delta;
        if (compareAndSet(current, next))
            return current;
    }
}
```

Устройство атомарного класса *AtomicLong*

Неделимый

Пакет java.util.concurrent.atomic

TEL-RAN
by Starta Institute

Классы

- *AtomicBoolean*
- *AtomicInteger*
- *AtomicLong*
- *AtomicReference*

Atomic-обёртки для *boolean*, *integer*, *long* и ссылочных типов.

Классы этой группы содержат метод *compareAndSet*.

Классы *AtomicInteger* и *AtomicLong* имеют также методы инкремента, декремента и добавления нового значения.

Неделимый

Пакет java.util.concurrent.atomic

Классы

- *AtomicIntegerArray*
- *AtomicLongArray*
- *AtomicReferenceArray*

Atomic-классы для массивов *integer*, *long* и ссылочных типов.

Элементы массивов могут быть изменены атомарно.

Неделимый

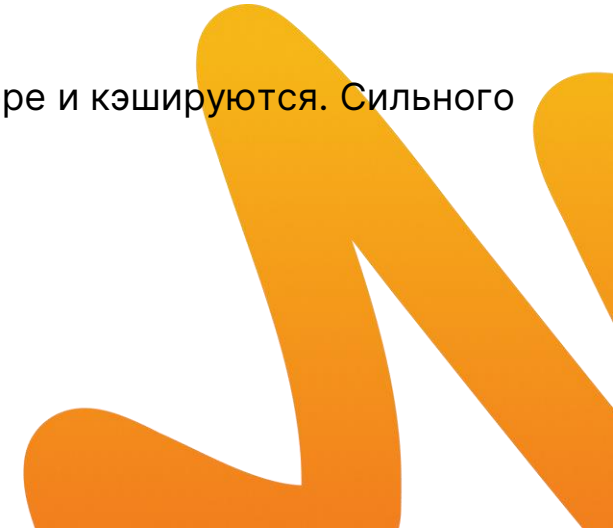
Пакет java.util.concurrent.atomic

Классы

- *AtomicIntegerFieldUpdater*
- *AtomicLongFieldUpdater*
- *AtomicReferenceFieldUpdater*

Atomic-классы для обновления полей по их именам с использованием рефлексии кода (*reflection)*.

Смещения полей для CAS операций определяется в конструкторе и кэшируются. Сильного падения производительности из-за *reflection* не наблюдается.



Неделимый

Пакет java.util.concurrent.atomic

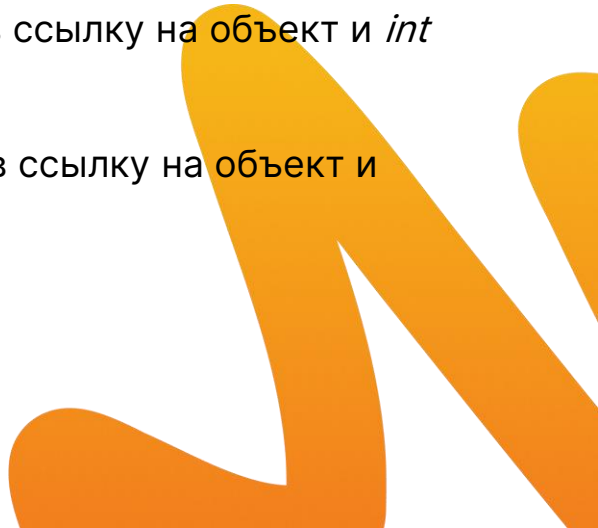
Классы

- *AtomicStampedReference*
- *AtomicMarkableReference*

Atomic-классы для реализации некоторых алгоритмов, (точнее сказать, уход от проблем при реализации алгоритмов).

Класс *AtomicStampedReference* получает в качестве параметров ссылку на объект и *int* значение.

Класс *AtomicMarkableReference* получает в качестве параметров ссылку на объект и битовый флаг (*true/false*).



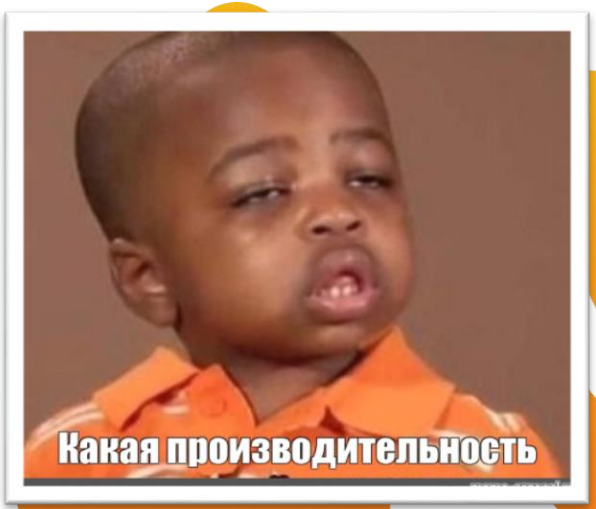
Неделимый

Производительность atomic

TEL-RAN
by Starta Institute

Согласно множеству источников неблокирующие алгоритмы в большинстве случаев более масштабируемы и намного производительнее, чем блокировки. Это связано с тем, что операции CAS реализованы на уровне машинных инструкций, а блокировки тяжеловесны и используют приостановку и возобновление потоков, переключение контекста и т.д. Тем не менее, блокировки демонстрируют лучший результат только при очень «высокой конкуренции», что в реальной жизни встречается не так часто.

Основной недостаток неблокирующих алгоритмов связан со сложностью их реализации по сравнению с блокировками. Особенно это касается ситуаций, когда необходимо контролировать состояние не одного поля, а нескольких.



Неделимый

Пример использования

TEL-RAN
by Starta Institute

Пусть класс *SequenceGenerator* генерирует последовательность [1, 2, 4, 8, 16, ...] в многопоточной среде. Каждое новое обращение даёт следующее значение из последовательности. Можно сделать доступ к объекту *SequenceGenerator* синхронным, но эффективнее будет обойтись без синхронизации и использовать атомарные классы.

ThreadAtomicExample.zip

Задание

TEL-RAN
by Starta Institute

Создайте класс IdHolder, который выдаёт новые идентификаторы по запросу.

Идентификаторы должны быть уникальны.

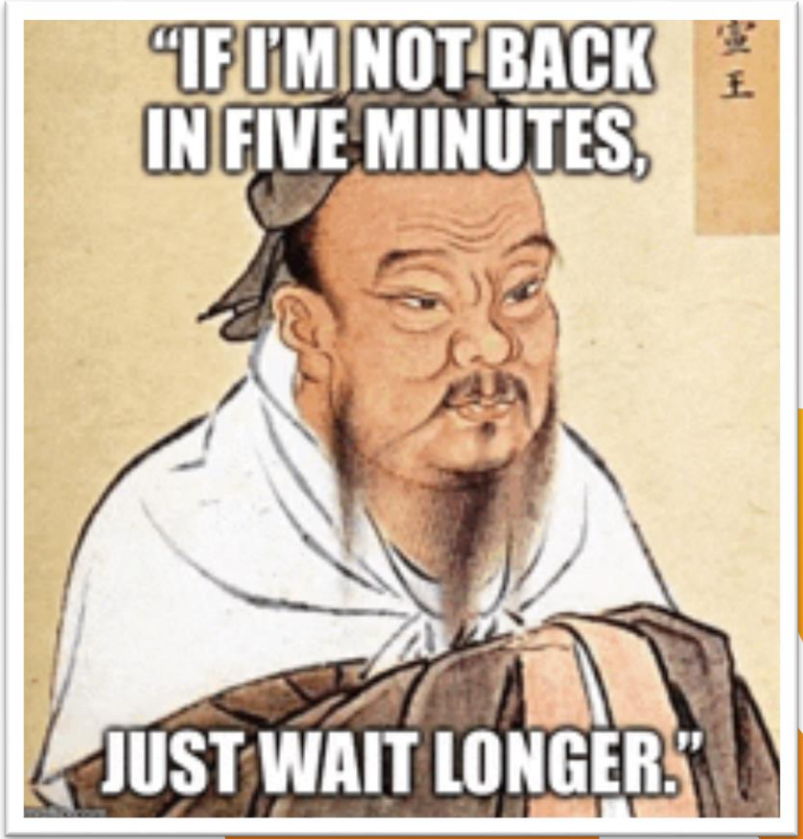
Создайте классы User, ApiUser и TechnicalUser, каждый из которых получает уникальный идентификатор после создания.

Т.к. пользователи регистрируются в параллельном режиме, создайте поток для каждого пользователя, чтобы получить идентификатор из IdHolder.

Проблема

TEL-RAN
by Starta Institute

Для реализации модели поставщиков-потребителей мы использовали методы *wait()* и *notify()*. Т. к. задача является типовой и чаще всего связана с потоком входных запросов, которые нужно обрабатывать в порядке прихода, в пакете *java.util.concurrent* предусмотрены блокирующие и неблокирующие очереди, пригодные для работы в многопоточной среде.

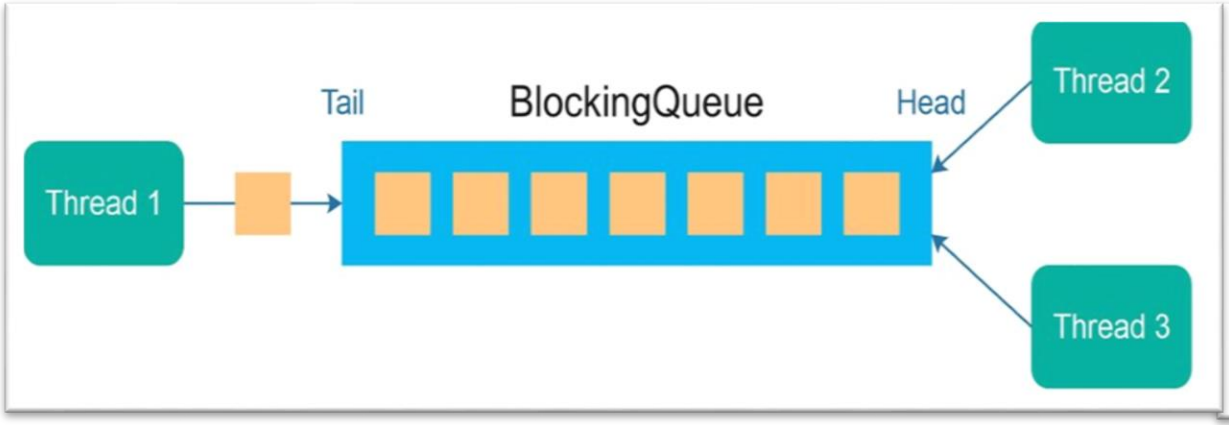


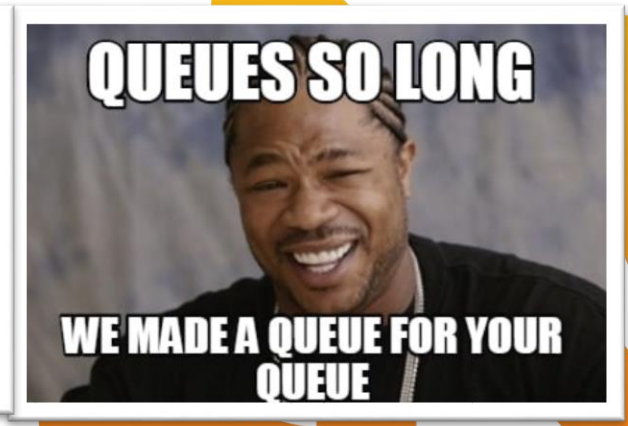
Держать очередь
Concurrent Queues

TEL-RAN
by Starta Institute

Неблокирующие очереди «заточены» на скорость выполнения.

Блокирующие очереди приостанавливают потоки при работе с очередью.





Держать очередь

Блокирующие очереди

TEL-RAN
by Starta Institute

Блокирующие очереди приостанавливают потоки при работе с очередью и дают возможность задать размер очереди и/или условия блокировки.

Блокирующие очереди реализуют интерфейсы

BlockingQueue – хранит элементы в порядке FIFO, гарантирует, что любая попытка извлечь элемент из пустой очереди заблокирует вызывающий поток до тех пор, пока не появится доступный элемент. Аналогично , любая попытка вставить элемент в заполненную очередь заблокирует вызывающий поток до тех пор, пока не освободится место для нового элемента. гарантирует, что элемент, созданный Producer будет в очереди.

BlockingDeque – включает дополнительные методы для двунаправленной блокирующей очереди, у которой данные можно добавлять и извлекать с обоих сторон очереди.

TransferQueue – блокируют поток записи до тех пор, пока другой поток не извлечет элемент. Для этого следует использовать метод *transfer*. Гарантирует, что элемент Producer'a «получает» Consumer.

Держать очередь

Реализации очередей

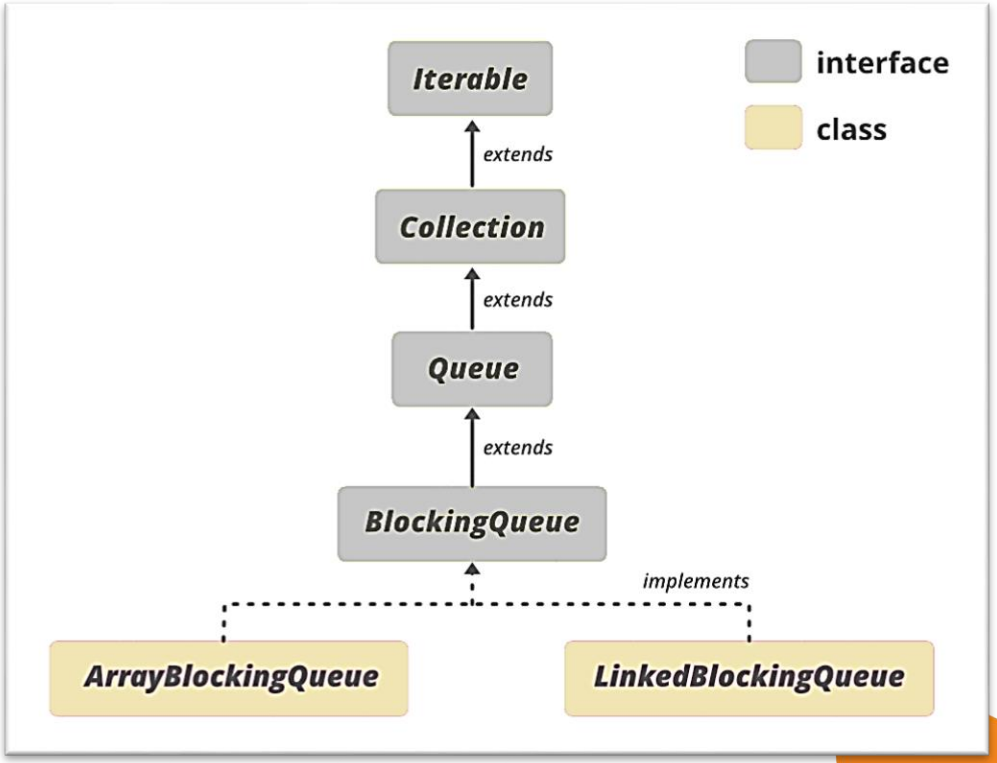
TEL-RAN by Starta Institute

Интерфейсы блокирующих очередей наряду с возможностью определения размера очереди включают методы, по-разному реагирующие на незаполнение или переполнение. Так, например, при добавлении элемента в переполненную очередь, один из методов вызовет *IllegalStateException*, другой вернет *false*, третий заблокирует поток, пока не появится место, четвертый же заблокирует поток на определенное время (таймаут) и вернет *false*, если место так и не появится.

	Вызывает Exception	**Чтение значения**	**Блокировка**	**Чтение с задержкой**
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Проверка	element()	peek()	не применим	не применим

Держать очередь

Реализации блокирующих очередей



Держать очередь

Реализации блокирующих очередей

ArrayBlockingQueue - блокирующая очередь, реализующая классический кольцевой буфер. Параметр *fair* в конструкторе позволяет управлять справедливостью очереди для упорядочивания работы ожидающих потоков производителей (вставляющих элементы) и потребителей (извлекающих элементы).

LinkedBlockingQueue - блокирующая очередь на связанных узлах, реализующая «two lock queue» алгоритм: один lock добавляет элемент, второй извлекает. За счет двух lock'ов данная очередь показывает более высокую производительность по сравнению с *ArrayBlockingQueue*, но и расход памяти повышается. Размер очереди задается через конструктор и по умолчанию равен *Integer.MAX_VALUE*.

LinkedBlockingDeque - двунаправленная блокирующая очередь на связанных узлах, реализованная как простой двунаправленный список с одним локом. Размер очереди задается через конструктор и по умолчанию равен *Integer.MAX_VALUE*.

Держать очередь

Пример ArrayBlockingQueue

ArrayBlockingQueue.zip

Держать очередь

Реализации блокирующих очередей

SynchronousQueue - блокирующая очередь, в которой каждая операция добавления должна ждать соответствующей операции удаления в другом потоке и наоборот. Т.е. очередь реализует принцип «один вошел, один вышел». SynchronousQueue не имеет никакой внутренней емкости, даже емкости в один элемент.

LinkedTransferQueue - блокирующая очередь с реализацией интерфейса **TransferQueue**, который позволяет при добавлении элемента в очередь заблокировать вставляющий поток до тех пор, пока другой поток не заберет элемент из очереди. Блокировка может быть как с таймаутом, так и с проверкой ожидающего потока.

DelayQueue - специфичный вид очереди, позволяющий извлекать элементы только после некоторой задержки, определенной в каждом элементе через метод *getDelay* интерфейса *Delayed*.

PriorityBlockingQueue - является многопоточной оберткой интерфейса PriorityQueue.

Задание

TEL-RAN
by Starta Institute

Пользователи чата шлют сообщение (текст, дата и время отправки, отправитель) всем участникам чата. Сервер принимает сообщения в многопоточном режиме. Т.к. задержка в канале у пользователей отличается, то сообщения могут приходить не в хронологическом порядке. Но выводится сообщения должны не в порядке получения на сервере, а с учётом даты отправки. Реализуйте процесс получения сообщений на сервере и вывод их в консоль в хронологическом порядке.

Проблема

TEL-RAN
by Starta Institute

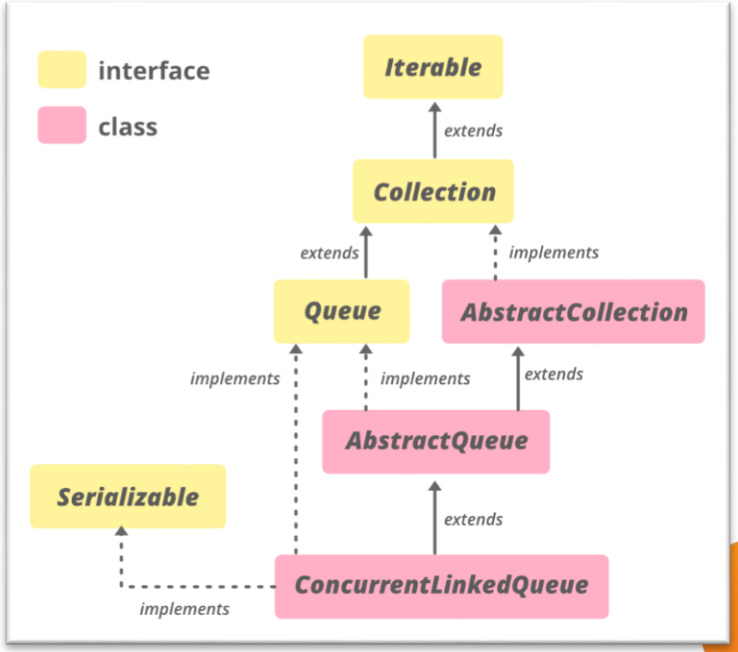
Чтобы достичь максимальной производительности работы очередей в многопоточной среде, *java.util.concurrent* предусмотрены неблокирующие очереди. Скорость их работы в основном упирается в возможности железа – сетевого соединения и оперативной памяти.

Держать очередь

Неблокирующие очереди

TEL-RAN
by Starta Institute

Неблокирующие очереди «заточены» на скорость выполнения и не приостанавливают потоки при работе с очередью. Любое количество потоков может работать с такой очередью благодаря реализации интерфейсов Queue и Deque на связанных узлах (*linked nodes*).



Держать очередь

Неблокирующие очереди

TEL-RAN
by Starta Institute

ConcurrentLinkedQueue - неограниченная по емкости и ориентированная на многопоточное исполнение очередь.

ConcurrentLinkedDeque – аналог *ConcurrentLinkedQueue*, реализует интерфейс *Deque*.

Итераторы классов отражают состояние очереди на определенный момент времени его создания и не вызывают *ConcurrentModificationException*. Содержавшиеся в очереди элементы с начала создания iterator'a, будут возвращены точно по запросу.
Аккуратно используйте метод *size*, который в отличие от большинства наборов, не является *constant-time operation*. Из-за асинхронной природы этой очереди, определение текущего количества элементов может быть неточным и требует обхода элементов, если этот набор изменяется во время обхода.
Объемные операции *addAll*, *removeAll*, *retainAll*, *containsAll*, *equals* и *toArray* не гарантируют, что будут выполнены атомарно.

Держать очередь

Конструкторы неблокирующих очередей

```
ConcurrentLinkedQueue()
ConcurrentLinkedQueue(Collection<? extends E> c)
```

```
ConcurrentLinkedDeque()
ConcurrentLinkedDeque(Collection<? extends E> c)
```

Держать очередь

Методы неблокирующих очередей

Очередь *ConcurrentLinkedQueue, ConcurrentLinkedDeque* и их *iterator'ы* реализуют все дополнительные методы интерфейсов *Queue/Deque* и *Iterator*. Также в очередях присутствуют методы, бросающие исключение *NoSuchElementException* вместо возврата *null,* когда мы пытаемся получить элемент из пустой очереди.

ConcurrentLinkedQueue:

Действие	**Вызов исключения**	**Возврат значения**
Вставить	add(e)	offer(e)
Удалить	remove()	poll ()
Исследовать	element()	peek ()

ConcurrentLinkedDeque:

Действие	**Первый элемент (голова)**		**Последний элемент (хвост)**	
	С исключением	*Получение значения*	*С исключением*	*Получение значения*
Вставить	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Удалить	removeFirst(e)	pollFirst(e)	removeLast(e)	pollLast(e)
Исследовать	getFirst(e)	peekFirst(e)	getLast(e)	peekLast(e)

Держать очередь

Использование ConcurrentLinkedQueue

ThreadConcurrentLinkedQueueExample.zip

Держать очередь

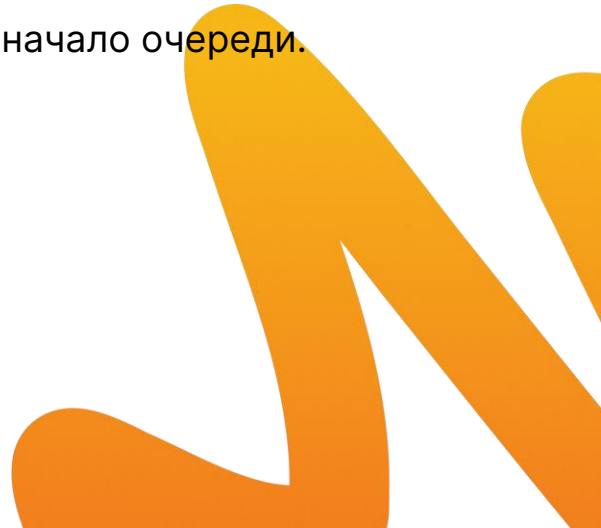
Использование ConcurrentLinkedDeque

ThreadConcurrentLinkedDequeExample.zip

Задание

TEL-RAN
by Starta Institute

1 Используйте ConcurrentLinkedQueue для передачи данных между производителями и потребителями. Производители должны генерировать случайные числа в диапазоне [1, 1000] и добавлять их в очередь, а потребители должны извлекать числа, выводить их в консоль и ожидать столько миллисекунд, сколько получили в виде числа из очереди.
2 Доработайте программу так, чтобы числа больше или равные 500 всегда помещались в конец очереди (долгие операции), а числа меньше 500 всегда в начало очереди.



TEL-RAN
by Starta Institute

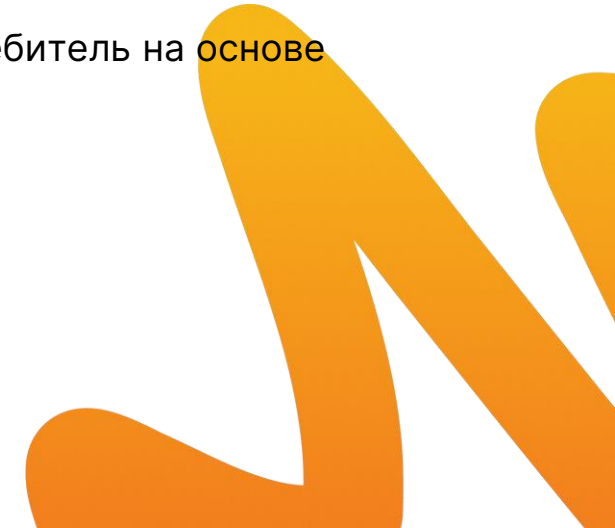
3 Домашнее задание

Домашнее задание

TEL-RAN
by Starta Institute

1 Создайте 10 потоков, каждый из которых «делает вычисления» (генерирует случайное число, ждёт сгенерированное число миллисекунд, добавляет сгенерированное число в общую для всех потоков переменную). Используя AtomicBoolean, создайте флаг, указывающий на возможность завершения приложения. Когда потоки накопят в общей переменной число 1_000_000 флаг становится true.

2 Создайте реализацию программы Поставщик – Склад - Потребитель на основе блокирующей очереди.



ЗАКЛЮЧЕНИЕ

TEL-RAN
by Starta Institute