Concurrency Utilities

- Набор классов, облегчающих написание многопоточных программ
- Пакет java.util.concurrent
  - Многопоточные коллекции
  - Примитивы синхронизации
  - Управление заданиями
- Пакет java.util.concurrent.locks
  - Работа с блокировками
- Пакет java.util.concurrent.atomic
  - Атомарные переменные

Concurrency Utilities. Блокировки

- Интерфейс Lock
- Методы
  - lock() – захватить блокировку
  - boolean tryLock() – захватить блокировку (если свободна)
  - lockInterruptibly() – захватить блокировку (возможность прерывания на ждущем потоке)
  - unlock() – отпустить блокировку
  - newCondition() – создать условие

Реализация блокировки

Класс ReentrantLock

Синтаксис

```
myLock.lock(); // a ReentrantLock object
try
{
   critical section
}
finally
{
   myLock.unlock(); // make sure the lock is unlocked even if an
exception is thrown
}
```

Условия

- Интерфейс Condition
  - newCondition() – получить экземпляр Condition
  - await(time)? – ждать условия
  - awaitUntil(deadline) – ждать условия до времени
  - signal() – подать сигнал
  - signalAll() – подать сигнал всем

Реализация блокировки

Пример: сущность Банк с набором accounts.

- случайным образом генерируются транзакции, которые переводят деньги со счета на счет.
- каждый перевод выполняется в отдельном потоке.

Реализация блокировки с условием

Перевод должен осуществляться только при наличии достаточной суммы на счете

Напрямую сделать нельзя:

```
if (bank.getBalance(from) >= amount)
// thread might be deactivated at this point
bank.transfer(from, to, amount);
```

Класс ReentrantLock

ReentrantLock()

ReentrantLock(boolean fair)

Режим Fairness – по умолчанию false

ReadWriteLock

Интерфейс **ReadWriteLock**
Поддерживает пару связанных блокировок (lock) – для операций чтения или записи.
Read lock могут получить несколько потоков – читателей до тех пор пока нет потоков писателей, write lock - взаимоисключающая.
Поток, получивший read lock видит все изменения, сделанные после предыдущего освобождения write lock.

Методы:
Lock readLock()
Lock writeLock()

ReentrantReadWriteLock

Класс **ReentrantReadWriteLock:**
Не дает предпочтения потокам читателям или писателям, но поддерживает дополнительную опцию - *fairness* policy.

Режим non-fair (по умолчанию).
Порядок read и write блокировок не определен.

Режим Fair.
Порядок read и write блокировок определяется порядком прихода потоков.
Когда текущая блокировка освобождается, то либо дольше всего ждущий поток писатель получает write lock или, если есть группа потоков писателей, ожидающих больше, чем все потоки писатели, то они получают read lock.

ReentrantReadWriteLock

Прерывание потоков:
Потоки могут быть прерваны во время ожидания read lock или write lock.

Поддержка Condition:
Write lock обеспечивает имплементацию Condition, read lock – нет - readLock().newCondition() бросает UnsupportedOperationException.

Семафоры

- Семафор имеет емкость, указываемую при создании

Конструктор: public **Semaphore**(int permits)

- Методы
  - acquire(n?) – получить разрешение
  - release(n?) – отдать разрешение
  - tryAcquire(n?, time?) – попробовать получить разрешение
  - reducePermits(n) – уменьшить количество разрешений

Семафоры

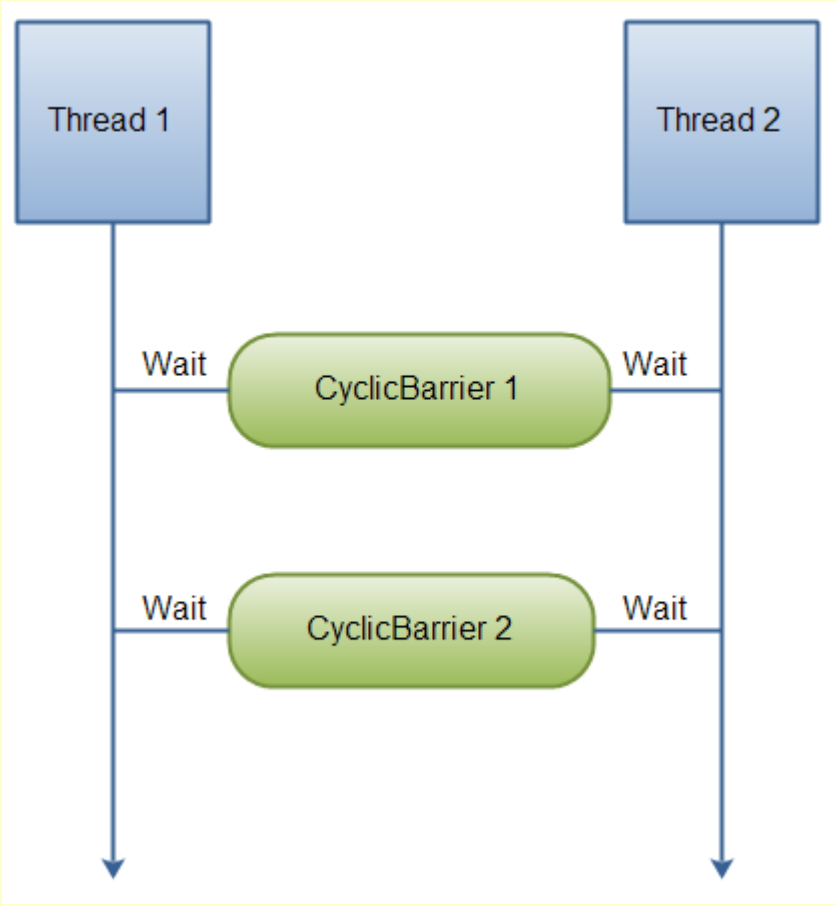
Пример:

• семафор предназначен для управления въездом в туннель с односторонним движением;

• по соображениям безопасности ограниченное число машин может въехать в туннель

Барьер (CyclicBarrier)

- Количество потоков и действие на барьере указывается при создании
- Методы
  - await(time?) – барьер.
  - reset() – возвращает барьер в исходное состояние
  - isBroken() – “сломан” ли барьер
  - статистика



Защелка (CountDownLatch)

- Высота защелки указывается при создании
- Методы
  - await(time?) – ждет спуска защелки.
  - countDown() – опускает защелку на единицу.
  - getCount() – текущая высота защелки.

Exchanger

Класс Exchanger предназначен для упрощения процесса обмена данными между двумя потоками исполнения. Принцип действия класса Exchanger: он ожидает до тех пор, пока два отдельных потока исполнения не вызовут его метод exchange ().

Как только это произойдет, он произведет обмен данными, предоставляемыми обоими потоками.

Phaser

- назначение — синхронизировать потоки исполнения, которые представляют одну или несколько стадий (или фаз) выполнения действия;
- поддерживает несколько фаз. Класс Phaser позволяет определить объект синхронизации, ожидающий завершения определенной фазы. Затем он переходит к следующей фазе и снова ожидает ее завершения.

  Phaser ()

  Phaser (int *количество сторон)*

Phaser

Для регистрации стороны после создания объекта класса Phaser следует вызвать метод register (). В итоге он возвратит номер регистрируемой фазы.

```
int register();
```

Синхронизатор фаз ожидает до тех пор, пока все зарегистрированные стороны не завершат фазу. Сторона извещает об этом, вызывая один методов:

int **arrive**();

int **arriveAndAwaitAdvance** ();

int **arriveAndDeregister**()

Phaser

int **arrive** (); - метод сообщает, что сторона (поток исполнения) завершила некоторую задачу (или ее часть). Он возвращает текущий номер фазы или отрицательное значение, если синхронизатор фаз завершил свою работу. Метод arrive () не приостанавливает исполнение вызывающего потока.

int **arriveAndAwaitAdvance** (); - метод ожидает до тех пор, пока все стороны не достигнут данной фазы, а затем возвращает номер следующей фазы или отрицательное значение, если синхронизатор фаз завершил свою работу.

int **arriveAndDeregister**() - метод возвращает номер текущей фазы или отрицательное значение, если синхронизатор фаз завершил свою работу. Он не ожидает завершения фазы.

final int **getPhase** (); - возвращает номер текущей фазы.

Phaser

Как только все стороны достигнут данной фазы, она считается завершенной, и синхронизатор фаз может перейти к следующей фазе (если она имеется) или завершить свою работу.

protected boolean **onAdvance** (int *фаза,* int *количество_сторон);* - метод вызывается исполняющей средой, когда синхронизатор фаз переходит от одной фазы к следующей.

Для того чтобы завершить работу синхронизатора фаз, метод onAdvance () должен возвратить логическое значение true.

В версии по умолчанию метод onAdvance() возвращает логическое значение true, чтобы завершить работу синхронизатора фаз, если зарегистрированных сторон больше нет.

Исполнители

Интерфейс **Executor**, в котором определяется метод:

void **execute** (Runnable поток);

метод запускает указанный поток на исполнение.

Интерфейс **ExecutorService** расширяет интерфейс Executor, дополняя его методами, помогающими управлять исполнением потоков и контролировать их.

void **shutdown** (); - метод останавливает все потоки исполнения, находящиеся в данный момент под управлением экземпляра интерфейса ExecutorService.

Интерфейс **ScheduledExecutorService** расширяет интерфейс ExecutorService и позволяет планировать выполнение команд после определенной задержки или с заданной периодичностью.

schedule(Runnable command, long delay, TimeUnit unit);

scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)

Исполнители

Предопределенные классы исполнителей:

ThreadPoolExecutor,

ScheduledThreadPoolExecutor.

Класс **ThreadPoolExecutor** реализует интерфейсы Executor и ExecutorService и обеспечивает поддержку управляемого пула потоков исполнения.

Класс **ScheduledThreadPoolExecutor** также реализует интерфейс ScheduledExecutorService для поддержки планирования пула потоков исполнения.

Исполнители

static ExecutorService **Executors.newCachedThreadPool()** - данная реализация применяется в тех случаях, когда заранее неизвестно, какое количество потоков будет передаваться исполнителю.

static ExecutorService **Executors.newFixedThreadPool(int) -** реализация применяется если количество потоков заранее известно. Это дает большой выигрыш в быстродействии, так как все потоки создаются сразу.

static ExecutorService **Executors.newSingleThreadExecutor() -** реализация применяется если необходимо передавать исполнителю только один объект класса, Если при использовании данной реализации исполнителю передается несколько потоков, то они попадут в очередь, и каждый из них будет запускаться только после завершения работы предыдущего.

Интерфейсы Callable и Future

Интерфейс **Callable** представляет поток исполнения, возвращающий значение:

interface Callable<V>; где параметр V обозначает тип данных, возвращаемых потоком исполнения.

В интерфейсе Callable определяется единственный метод call ():

V **call**() throws Exception

В теле метода call () определяется задача, которую требуется выполнить. Когда она будет выполнена, возвращается результат. Если результат нельзя вычислить, метод call () генерирует исключение.

Для выполнения задачи типа Callable вызывается метод **submit** (), определенный в интерфейсе ExecutorService:

<T> Future<T> **submit** (Callable <T> *задача*)

Интерфейсы Callable и Future

Интерфейс **Future** является обобщенным интерфейсом и представляет значение, возвращаемое объектом типа Callable.

interface Future<V>;

Чтобы получить значение, следует вызвать метод get () из интерфейса Future:

V **get** () throws InterruptedException, ExecutionException

V **get** (long *ожидание*, TimeUnit *единица_времени*) throws InterruptedException, ExecutionException, TimeoutException

CompletionStage

public interface **CompletionStage**<T> - этап возможного асинхронного вычисления, на котором выполняется действие или вычисляется значение после завершения другого CompletionStage.

Этап завершается после окончания вычислений, но это, в свою очередь, может инициировать другие зависимые этапы.

Вычисление, выполняемое на очередной стадии, может быть выражено как **Function**, **Consumer** или **Runnable** (с использованием методов с именами *apply*, *accept*, *run* соответственно).

CompletableFuture<T> implements Future<T>, CompletionStage<T>

CompletableFuture

CompletableFuture<String>completableFuture =
new **CompletableFuture** <String>();

Получить результат этого **CompletableFuture**:

String result = completableFuture.**get**();

Завершить CompletableFuture вручную:

completableFuture.**complete**("Результат Future");

Все клиенты, ожидающие этот Future, получат указанный результат, а последующие вызовы completableFuture.complete() будут игнорироваться.

Перечисление TimeUnit

Перечисление TimeUnit служит для обозначения *степени разрешения* синхронизации.

- DAYS
- HOURS
- MINUTES
- SECONDS
- MICROSECONDS
- MILLISECONDS
- NANOSECONDS

(нет гарантии того, что система сможет работать с заданным разрешением)

Перечисление TimeUnit

В перечислении TimeUnit определяются различные методы, выполняющие преобразование единиц:

long **convert** (long время, TimeUnit *единица_времени)*

long **toMicros** (long время)

long **toMillis** (long время)

long **toNanos** (long время)

long **toSeconds** (long время)

long **toDays** (long время)

long **toHours** (long время)

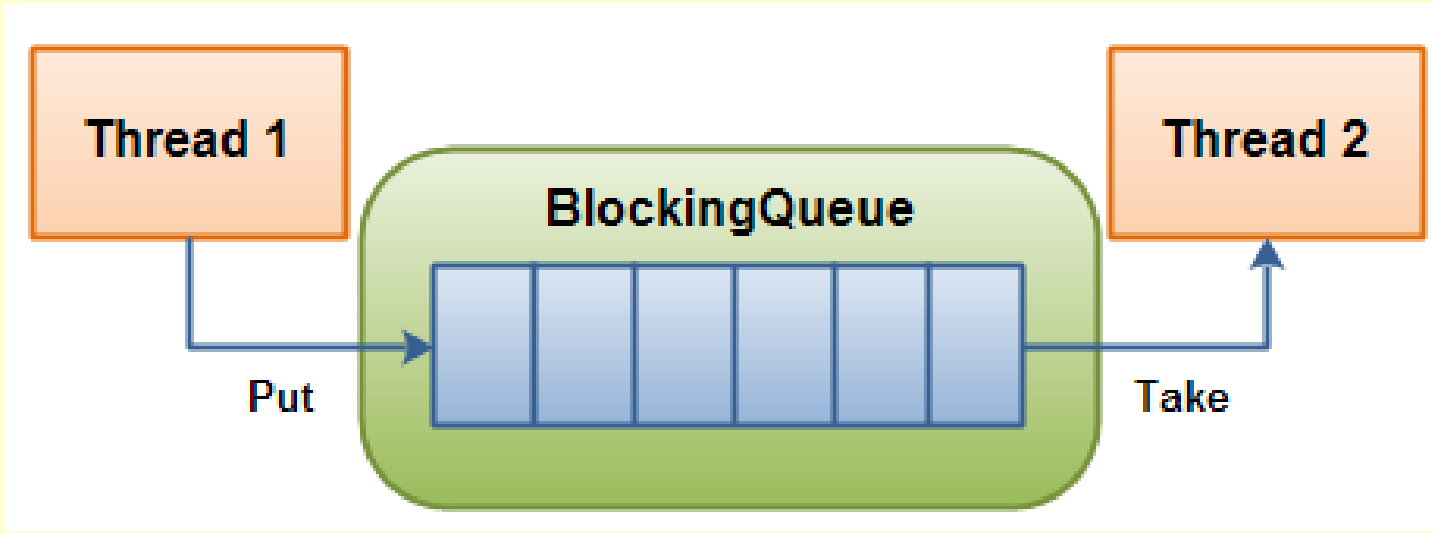
long **toMinutes** (long время)

Метод convert () преобразует заданное *время* в единицы времени, обозначаемые параметром *единица_времени* и возвращает результат.

Многопоточные очереди

- Интерфейс BlockingQueue
- Отличие от обычных очередей
  - Возможно ограничение размера
  - Методы добавления могут бросасать IllegalStateException
  - Дополнительные операции

Многопоточные очереди



Дополнительные операции

- Добавление элемента
  - put(E) – блокируется до появление места.
  - offer(E, timeout, unit) – блокируется на время.
- Удаление элемента
  - take() – блокируется до появления элемента.
  - poll(E, timeout, unit) – блокируется на время.
- Получение нескольких элементов
  - drainTo(Collection) – удаление всех элементов.
  - drainTo(Collection, max) – удаление max количества элементов.

Реализации BlockingQueue

- ArrayBlockingQueue – очередь на массиве
- LinkedBlockingQueue – очередь на списке
- PriorityBlockingQueue – очередь с приоритетами
- DelayQueue – очередь элементов в задержкой
- SynchronousQueue – очередь без "внутренности"

Многопоточные отображения

- Представляются интерфейсом ConcurrentMap<K, V>
- Дополнительные методы
  - putIfAbsent(K, V) – добавляет отображение K, если его ранее не было
  - remove(K, V) – удаляет K, если он отображен на V
  - replace(K, V) – заменяет отображение K
  - replace(K, oldV, newV) – условная замена отображения для ключа K

Другие многопоточные коллекции

- Коллекции с многопоточным доступом
  - ConcurrentHashMap – отображение
  - ConcurrentLinkedQueue – очередь на списке
- Коллекции, “самокопирующиеся” при записи
  - CopyOnWriteArrayList – список на массиве
  - CopyOnWriteArraySet – множество на массиве