



Санкт-Петербургский Национальный
Исследовательский университет ИТМО

Факультет Программной Инженерии и Компьютерной Техники

# Программирование на JAVA.
# «синхронизаторы в пакете java.util.concurrent»

Преподаватель - Гаврилов А.В.
Выполнил - Дробыш Д.А. (333219)
Группа - P3110

Санкт-Петербург
2022

# 1. Introduction.

В первую очередь необходимо разобраться с **java.util.concurrent.**

**java.util.concurrent.**
Это пакет, который предназначен для разработки многопоточных программ, поддержания корректной работы кода. Часто разработчики используют только средства обычной синхронизации, но java.util.concurrent предоставляет дополнительные возможности.

java.util.concurrent

- ✓Атомарные переменные
- ✓Неблокирующие коллекции
- ✓Блокирующие коллекции
- ✓Примитивы синхронизации
- ✓Пулы потоков
- ✓CompletableFuture

*Сегодня нас интересуют только примитивы синхронизации, поэтому давайте перейдем к ним.*

# 2. Semaphore.

- Класс java.util.concurrent.Semaphore;
- Ограничивает одновременный доступ к ресурсу;
- Основное отличие от **synchronized**-блока: могут работать несколько потоков, задается ограничение каким-то числом N;
- Основные операции :

  **void acquire();** —> захват.
  **void release();** —> освобождение.

Пример использования:
Пусть у нас есть приложение, которое подключается к базе данных, а нам необходимо обеспечить не более N подключений к базе. Тогда соответственно используем Semaphore и защищаем наш код.

Давайте рассмотрим стандартный шаблон работы с Semaphore.
(Рабочие примеры вы можете найти в папке проекта)

```java
import java.util.concurrent.Semaphore;

public class main {
    public static void main(String[] args) throws InterruptedException {
        Semaphore semaphore = new Semaphore( permits: 7);
        semaphore.acquire();
        try {
            // можно до 7 потоков
            // код может работать одновременно

        }finally {
            semaphore.release();
        }
    }
}
```

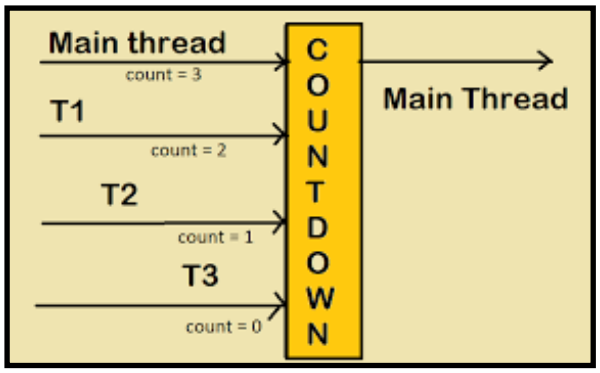
## 3. CountDownLatch.

- Класс java.util.concurrent.CountDownLatch
- Обеспечивает синхронизацию между потоками. N потоков могут дожидаться друг друга, чтобы потом одновременно стартовать
- Основные операции:

```
void await();
void countDown();
```

Вот как-то так и никак иначе… (слезы на щеках, что ты Java плачешь :)))

А почему нельзя иначе?
Предположим, что у нас есть N потоков, которые начали делать какую-то работу, причем работа делится на инициализацию и полезную работу. Мы не знаем, сколько занимает инициализация, а сколько полезная работа, но при этом мы хотим одновременный запуск полезной работы всех потоков (сразу после инициализации последнего потока ). Ну а из-за неизвестности времени инициализации, да и вообще недоверию к распределению между потоками в OS, Timeout мы использовать не можем. Получается, что нам необходим соответсвующий примитив синхронизации. Ну а это CountDownLatch.




На картинке я привел пример работы. В каждом потоке будет CountDown и await. С каждой инициализацией счетчик будет декрементироваться и ждать. Ну, а после инициализации последнего все разом начнут работать.

Вот Шаблон.
(Рабочие примеры вы можете найти в папке проекта)

```
CountDownLatch countDownLatch = new CountDownLatch(7);
// latch.countDown() 7 раз
countDownLatch.await();
```

## 4. CyclicBarrier.

- Класс java.util.concurrent.CyclicBarrier
- Полностью отражает идею CountDownLatch, но допускает повторное ожидание.

## 5. ReentrantLock.

- Класс java.util.concurrent.locks.ReentrantLock
- Обеспечивает взаимное исключение потоков, аналогичное **synchronized**-блокам.
- Основные операции:

```
lock();
unlock();
```

- Отличие от встроенной синхронизации: Синтаксическая конструкция заменилась на объект со своими методами (lock(), unlock()). Это позволяет нам захватить lock в одном

методе, а отпустить в другом, что значительно расширяет стандартное решение.
( **synchronized** блок должен быть внутри одного метода и никак иначе ).

```
Lock locker = new ReentrantLock();

locker.lock();
try {
    //что-то делаем
}finally {
    locker.unlock();
}
```

Вот шаблон.
(Рабочие примеры вы можете найти в папке проекта)

## 5. Condition.

- Класс java.util.concurrent.locks.Condition
- Прямой аналог **wait/notify (await\signal)**
- Привязывается к Lock-у
- У одного Lock-а может быть несколько Conditions.

Зачем это нужно?

Пусть у нас есть очередь, в которую можно добавлять элементы/ извлекать их из нее. При этом операция извлечения блокирует текущий поток, если очередь пуста. (до тех пор, пока не появится ). А операция добавления проверяет на максимум очереди и блокирует поток, если максимум достигнут. По такому принципу после удаления элемента из очереди есть смысл разблокировать потоки с возможностью добавления, обратно ситуация симметрична. Но! Если мы будем решать такую задачу на встроенной синхронизации, то мы будем всегда будить все потоки, а не необходимые нам, следовательно, теряем эффективность. А через Condition и lock мы можем решить эту проблемы. На один lock накинуть 2 condition и раздельно на них вызывать await и signal;

тут как таковой шаблон не очень удобно показывать, поэтому рекомендую посмотреть простой пример в папке.

## 6. ReentrantReadWriteLock.

- Класс java.util.concurrent.locks.ReentrantReadWriteLock
- Поддерживает разделение доступа на чтение и на запись.

Разделяем читателей и писателей. Потоки на чтение никак не мешают друг другу, а вот во время изменения могут возникнуть проблема. Этим способом мы отправляем писателей поочередно (по 1), а потом хоть всех читателей сразу. Главное - писатели и читатели не должны работать одновременно. Выгодно для ситуаций с частым чтением памяти и редкой ее модификацией.

работает практически аналогично прошлому.