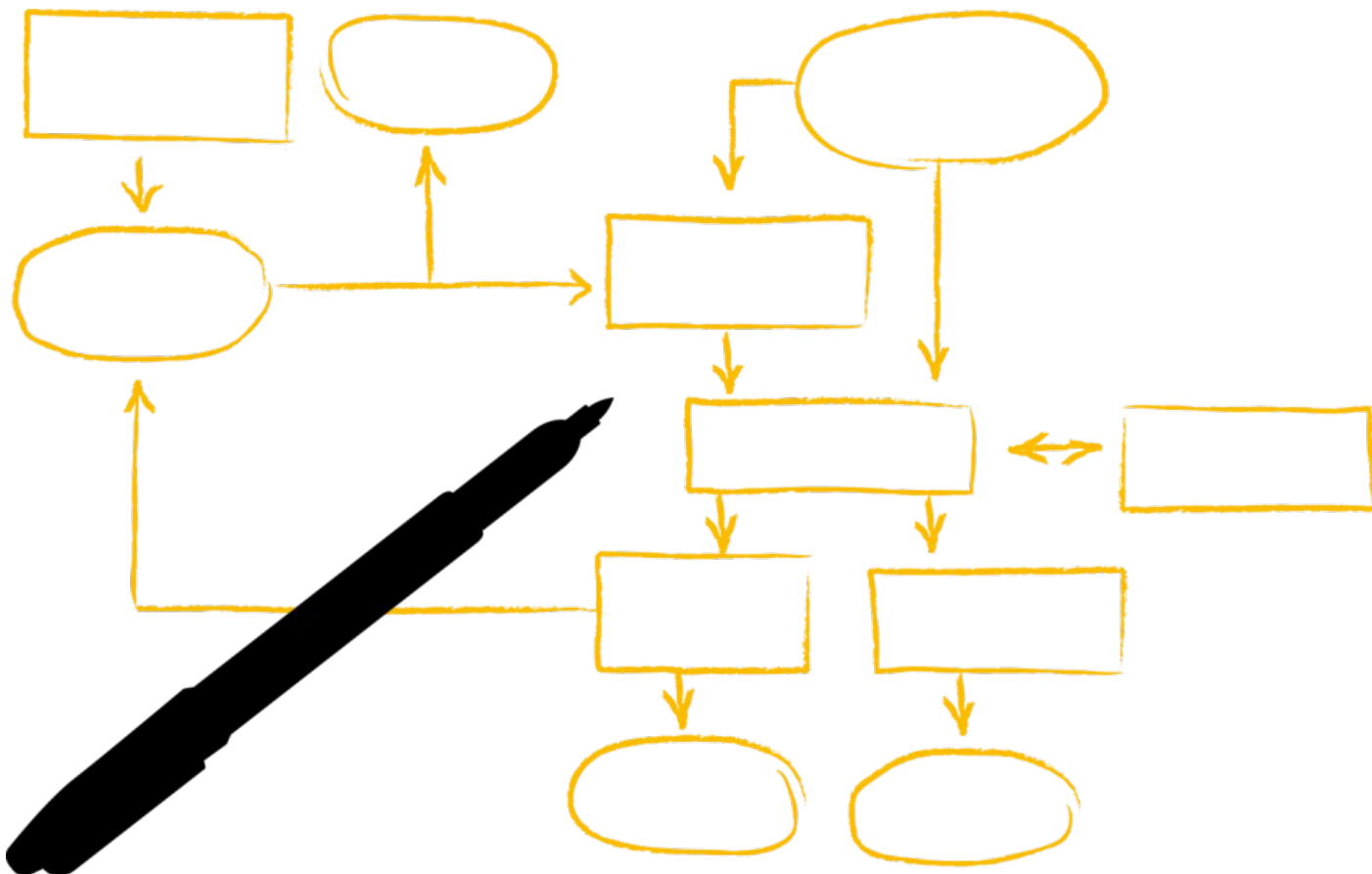


# Многонишково програмиране

## Ключалки, семафори, бариери

Владимир Панов  
Декември, 2014

Public



# Ключалка

`java.util.concurrent.locks.Lock`

---

Интерфейсът **`java.util.concurrent.locks.Lock`** предоставя функционалност, подобна на блок **`synchronized`**.

## Допълнителни възможности:

- Заклучването и отключването може да се случат на различни места в кода.
- Нишката може да се откаже от заклучването, ако:
  - ключалката е заета;
  - ключалката не може да бъде получена за определено време.
- Нишката може да бъде прекъсната докато чака за ключалката.

## Недостатъци:

- Отговорността за отключването пада изцяло върху програмиста.

В JDK има само една директно използвана имплементация на този интерфейс: **`java.util.concurrent.locks.ReentrantLock`**.

# Ключалка

java.util.concurrent.locks.Lock - методи

---

**void lock()**

Като влизане в **synchronized** блок. Нишката ще бъде блокирана, докато получи ключалката. Нишката не може да бъде прекъсната, докато е блокирана.

**boolean tryLock()**

Ако ключалката е свободна, то нишката я получава и методът връща **true**. В противен случай методът връща **false**. Никога не блокира.

**void lockInterruptibly() throws InterruptedException**

Като **lock()**, но нишката може да бъде прекъсната, докато е блокирана.

**boolean tryLock(long time, TimeUnit unit) throws InterruptedException**

Като **tryLock()**, но блокира за ограничено време, ако ключалката не е свободна. Нишката може да бъде прекъсната, докато е блокирана.

**void unlock()**

Освобождава ключалката. Трябва да бъде извикан по веднъж за всяко успешно извикване на някой от заключващите методи.

# Условие

`java.util.concurrent.locks.Condition`

---

Интерфейсът **`java.util.concurrent.locks.Condition`** предоставя функционалност, подобна на **`Object.wait(...)`** и **`Object.notify*()`**.

## Допълнителни възможности:

- Повече от едно събитие на една ключалка.
- Нишката може да чака за уведомяването без да бъде прекъсвана.
- Нишката може да чака за уведомяването до определен момент.

Всяко условие изисква ключалка.

=> Единственият начин за конструиране на ново условие е чрез метода:

**`Condition Lock.newCondition()`**

Условията също могат да доведат до фалшиво събуждане.

# Условие

java.util.concurrent.locks.Condition - методи

---

`void await() throws InterruptedException`

Като `Object.wait()`.

`boolean await(long time, TimeUnit unit) throws InterruptedException`

`long awaitNanos(long nanosTimeout) throws InterruptedException`

`boolean awaitUntil(Date deadline) throws InterruptedException`

КАТО `Object.wait(long timeout)` И `Object.wait(long timeout, int nanos)`, С  
НЯКОИ ДОПЪЛНИТЕЛНИ УДОБСТВА.

`void awaitUninterruptibly()`

КАТО `await()`, НО НИШКАТА НЕ МОЖЕ ДА БЪДЕ ПРЕКЪСНАТА, ДОКАТО Е БЛОКИРАНА.

`void signal()`

КАТО `Object.notify()`.

`void signalAll()`

КАТО `Object.notifyAll()`.

# Ключалка за четене и запис

`java.util.concurrent.locks.ReadWriteLock`

---

Често се налага един ресурс за бъде заключван отделно за четене и запис така, че да бъде позволено конкурентно четене:

- Ресурсът може да бъде заключен за четене, ако не е заключен за запис.
  - Ако ресурсът е заключен за четене от поне една нишка, то той може да бъде заключен за четене от други нишки, но не може да бъде заключен за запис.
- Ресурсът може да бъде заключен за запис, ако не е заключен нито за запис, нито за четене.
  - Ако ресурсът е заключен за запис от една нишка, то той не може да бъде заключен нито за четене, нито за запис от друга нишка.

За целта може да използвате интерфейса **`java.util.concurrent.locks.ReadWriteLock`**. Той има само два метода – за получаване на свързаните помежду си ключалки за четене и запис.

В JDK има само една имплементация на този интерфейс:  
**`java.util.concurrent.locks.ReentrantReadWriteLock`**.

# Семафор

`java.util.concurrent.Semaphore`

---

Семафорът е синхронизационен механизъм, който управлява  $n$  разрешителни.

- Ако дадено парче код се нуждае от разрешително за да свърши някаква работа, то се опитва да получи такова от семафора. Ако всичките  $n$  разрешителни са раздадени, то нишката ще блокира, докато не се освободи някое.
- След като парчето код си свърши работата разрешителното трябва да бъде върнато на семафора.

Възможно е едно разрешително да бъде получено в една нишка, а да бъде върнато в друга.

Семафор с едно разрешително прилича на ключалка, с две съществени разлики:

- Не е рекурсивен (една нишка може да се самоблокира, ако притежава разрешителното и се опита да получи ново) – може да се получи мъртва хватка само с една нишка.
- Отключването (връщането на разрешителното) може да се случи в друга нишка.

# Семафор

java.util.concurrent.Semaphore - методи

---

**Semaphore(int n)**

Създава семафор с **n** разрешителни.

**void acquire() throws InterruptedException**

**void acquireUninterruptibly()**

Получава едно разрешително от семафора. Нишката ще бъде блокирана, докато се освободи разрешително. **acquireUninterruptibly()** не позволява нишката да бъде прекъсната, докато е блокирана.

**void acquire(int k) throws InterruptedException**

**void acquireUninterruptibly(int k)**

Като **acquire\*()**, но получава **k** разрешителни наведнъж.

# Семафор

java.util.concurrent.Semaphore - МЕТОДИ

---

```
boolean tryAcquire()  
boolean tryAcquire(long timeout, TimeUnit unit)  
    throws InterruptedException  
boolean tryAcquire(int k)  
boolean tryAcquire(int k, long timeout, TimeUnit unit)  
    throws InterruptedException
```

Подобно на `Lock.tryLock(...)` - получават 1 или k разрешителни, само ако те са налични веднага или до изтичането на зададен таймаут.

```
void release()  
void release(int k)
```

Връща 1, респ. k, разрешителни на семафора.

# Бариера

java.util.concurrent.CyclicBarrier

---

Бариерата е синхронизационен механизъм, който дава възможност на **n** нишки да достигнат до обща точка, след което да продължат изпълнението си едновременно.

Една бариера с предел **n** блокира всяка нишка, чакаща на нея, докато броят на чакащите нишки не стане **n**. Тогава бариерата се вдига и всичките **n** нишки преминават в състояние *изпълнима* едновременно.

Една бариера се счупва, ако се случи едно от следните неща:

- някоя от чакащите нишки бъде прекъсната. В този случай прекъснатата нишка ще получи **InterruptedException**, а всички останали – **BrokenBarrierException**.
- таймаутът на някоя от чакащите нишки изтече. В този случай тази нишка ще получи **TimeoutException**, а всички останали – **BrokenBarrierException**.
- бариерата бъде преинициализирана от външна нишка. В този случай всички чакащи на бариерата нишки ще получат **BrokenBarrierException**.

# Барьера

java.util.concurrent.CyclicBarrier - методи

---

**CyclicBarrier(int n)**

Създава бариера с предел **n**.

**int getParties()**

Връща **n**.

**int await() throws InterruptedException, BrokenBarrierException**

**int await(long timeout, TimeUnit unit)**

**throws InterruptedException, BrokenBarrierException, TimeoutException**

Нишката се блокира докато бариерата не бъде вдигната (респ. без или с таймаут).

Връща индекса на пристигане; първата пристигнала нишка има индекс **n-1**, а последната (която не се блокира) – **0**.

**boolean isBroken()**

Проверява дали бариерата е счупена.

**void reset()**

Ако бариерата е здрава, първо я счупва. След това я преинициализира за нова употреба. Ако една бариера е счупена, тя може да се използва само след преинициализиране.

# Резе с обратно броене

`java.util.concurrent.CountDownLatch`

---

Резето с обратно броене е синхронизационен механизъм, който дава възможност на няколко нишки да изчакат извършването на **n** операции от някакъв тип.

Едно резе с начална бройка **n** блокира всяка нишка, чакаща на него, докато не бъде уведомено за извършването на **n** операции. Тогава резето се отключва и всички нишки, чакащи на него, преминават в състояние *изпълнима* едновременно.

Веднъж отключено, едно резе не може да бъде заключено пак:

- Всяка нишка, която се опита да чака на вече отключено резе, няма да бъде блокирана изобщо.
- Резетата не са преизползваеми.

# Резе с обратно броене

java.util.concurrent.CountDownLatch - методи

---

**CountDownLatch(int n)**

Създава резе с обратно броене с начална бройка **n**.

**void await() throws InterruptedException**

**boolean await(long timeout, TimeUnit unit)**

**throws InterruptedException**

Нишката се блокира докато резето не бъде вдигнато (респ. без или с таймаут).

**void countDown()**

Уведомява резето, че е извършена една операция. След **n**-тото извикване всички нишки, чакащи на резето (вкл. бъдещи такива), ще преминат в състояние *изпълнима*.



**Благодаря за вниманието!**