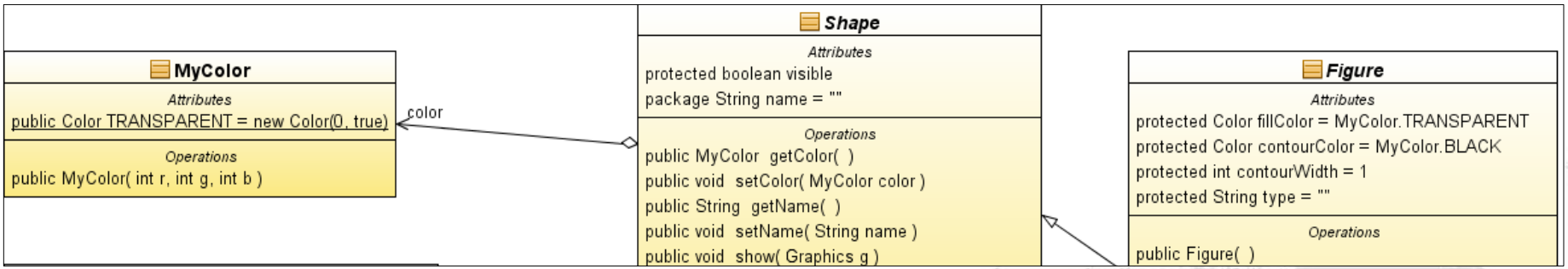
Паралелни процеси в езика Java™, споделяне на ресурси и синхронизация, блокиране, приоритети



Траян Илиев

IPT – Intellectual Products & Technologies
e-mail: tiliev@iproduct.org
web: http://www.iproduct.org

Съдържание

Паралелни процеси – множество нишки

- Процес – самостоятелна програма със собствено адресно пространство (клас ***java.lang.ProcessBuilder***)
- Нишка (***Thread***) – "олекотени процеси", могат да споделят помежду си ресурси в рамките на процеса, включително памет и отворени файлове
- Multi – threading реализация в съвременните ОС
- Предимства от използване на множество нишки
  - реагиращи потребителски интерфейси
  - не-блокиращ (асинхронен) достъп до ресурс
  - увеличаване на пропусквателната способност
  - опашки и буфериране
- Message Oriented Middleware (MOM)

Реализация на нишки в езика Java™

- Нишки в езика Java™ - начин на реализация
  - Клас **Thread**
  - Методи **run()** и **start()**
- По важни методи на класа **Thread**
  - sleep()
  - yield()
  - setPriority()
  - wait()
  - join()
- Нишки демони - setDaemon(true)

JSR 166: Concurrency Utilities

- Рамкови класове за реализиране на фино-гранулярна конкурентност – ***Executor framework*** (пакет ***java.util.concurrent***) за стандартизирано извикване, отложено изпълнение и контрол на асинхронни задачи на базата на политики, с автоматично пулиране на нишки
- Високоефективни конкурентни колекции и асоциативни списъци: ***BlockingQueue, ConcurrentMap, ConcurrentNavigableMap***
- Атомарни променливи за ефективни паралелни алгоритми, броячи и генератори на номера - ***java.util.concurrent.atomic***
- Synchronizers: *semaphores, mutexes, barriers, latches, exchangers*
- Locks: по-ефективно и гъвкаво заключване на ресурси сравнено с synchronized – ***java.util.concurrent.locks***
- Наносекундна прецизност на тайминга на задачите

Предимства на JSR 166: Concurrency Utilities

- По-лесно и бързо програмиране чрез използване на вече доказано ефективни и тествани конструкции и класове в пакета **java.util.concurent** и неговите под-пакети
- По-добра продуктивност и гъвкавост при избор на алтернативи
- Идиоми за заключване (Locking), които опростяват много конкурентни приложения
- High-level API за изпълнение и управление на задачи в **Thread Pools** с използване на **Executors**
- По-лесно управление на колекции от данни редуциращи нуждата от синхронизация
- По-добра поддръжка на кода в бъдеще

Executors Framework I

- **Executor** интерфейси дефинирани в пакета **java.util.concurrent**:
  - **Executor** – базов интерфейс поддържащ изпълнение на нови задачи
  - **ExecutorService** – подинтерфейс на Executor добавящ методи за управление на жизнения цикъл на задачите
  - **ScheduledExecutorService** – подинтерфейс на ExecutorService добавящ поддръжка на отложено и периодично изпълнение на задачи

Executors Framework II

- Типове ExecutorService:
  - CachedThreadPool – нишки се създават при необходимост
  - FixedThredPool – фиксиран брой нишки
  - SingleThredPool –една нишка-без синхронизация
  - ScheduledThreadPool – отложено и периодично изпълнение на задачи
- Асинхронно връщане на резултат- Callable & Future
- Прекъсване на задачи и Executors – методи: shutdown(), shutdownNow(), awaitTermination()

Връщане на асинхронен резултат с използване на Callable & Future I

```
Callable<List<ImageData>> task =
        new Callable<List<ImageData>>() {
            public List<ImageData> call() {
                List<ImageData> result = new ArrayList<ImageData>();
                for (ImageInfo imageInfo : imageInfos)
                    result.add(imageInfo.downloadImage());
                return result;
            }
        };
Future<List<ImageData>> future = executor.submit(task);
```

Връщане на асинхронен резултат с използване на Callable & Future II

```
try {
    List<ImageData> imageData = future.get();
    for (ImageData data : imageData)
        renderImage(data);
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    future.cancel(true);
} catch (ExecutionException e) {
    throw Throwable(e.getCause());
} }
```

Canceling ExecutorSevice and Tasks

```
void shutdownAndAwaitTermination(ExecutorService executor) {
  executor.shutdown();   // Disable tasks submission
  try {
     if (!executor.awaitTermination(120, TimeUnit.SECONDS)) {
        executor.shutdownNow();   // Cancel tasks currently submitted
        if (!executor.awaitTermination(120, TimeUnit.SECONDS))
           System.err.println("Executor did not finish successfully");
     }
  } catch (InterruptedException ie) {
     executor.shutdownNow();     Thread.currentThread().interrupt();
  }}
```

Опции за реализация на нишки

- Вътрешен клас наследник на **Thread**
- Анонимен вътрешен клас, наследник на **Thread**
- Клас имплементиращ интерфейса **Runnable**
- Анонимен вътрешен клас, имплементиращ интерфейса **Runnable**
- Стартиране на нишка в отделен метод

Споделяне на общи ресурси между множество нишки

- Проблеми при достъп до общи ресурси от множество нишки
- Средства за синхронизация
  - синхронизирани методи
  - критични секции (synchronized blocks)
  - семафори – методи: acquire(), release()
  - монитори – class Object: wait(), notify, notifyAll()
  - канали – PipedInputStream, PipedOutputStream,PipedReader, PipedWriter
- Обекти, по които синхронизираме достъпа на нишките
- Атомарни операции и **volatile** атрибути

Атомарни операции и Volatile атрибути I

- volatile ключовата дума не грантира заключване (++ операцията не е атомарна в Java)
- При честа промяна на стойностите поради нужда от обновяване на стойностите
- volatile може да бъде използвана когато:
  - Полето не трябва да поддържа инвариантни условия включващи други полета
  - Записа на стойност в полето не зависи от предишната му стойност
  - Никоя нишка никога не записва невалидна стойност в полето
  - Действията на прочитащите стойността нишки не зависят от стойности на други non-volatile полета

Атомарни операции и Volatile атрибути II

- Атомарност – достъпа и промяната на стойност на полета от примитивен тип без long и double са атомарни
- Достъпът дори до long и double полета е атомарен, ако те са декларирани volatile
- Атомарността сама по себе си не гарантира, че разполагаме с последните стойности на полетата
- volatile гарантира, че промените в полето направени от една нишка ще бъдат видими за всички други нишки незабавно (synchronized има същия memory effect)

Състояния на нишка, блокиране

- Правилни и неправилни начини за взаимно блокиране / разблокиране на нишки
  - wait() - notify() - освобождават заключванията върху ресурси
  - stop(), suspend(), resume() - не освобождават заключванията върху ресурси - deprecated
- Начини за правилно спиране на нишки
  - чрез флаг
  - с прекъсване - interrupt()

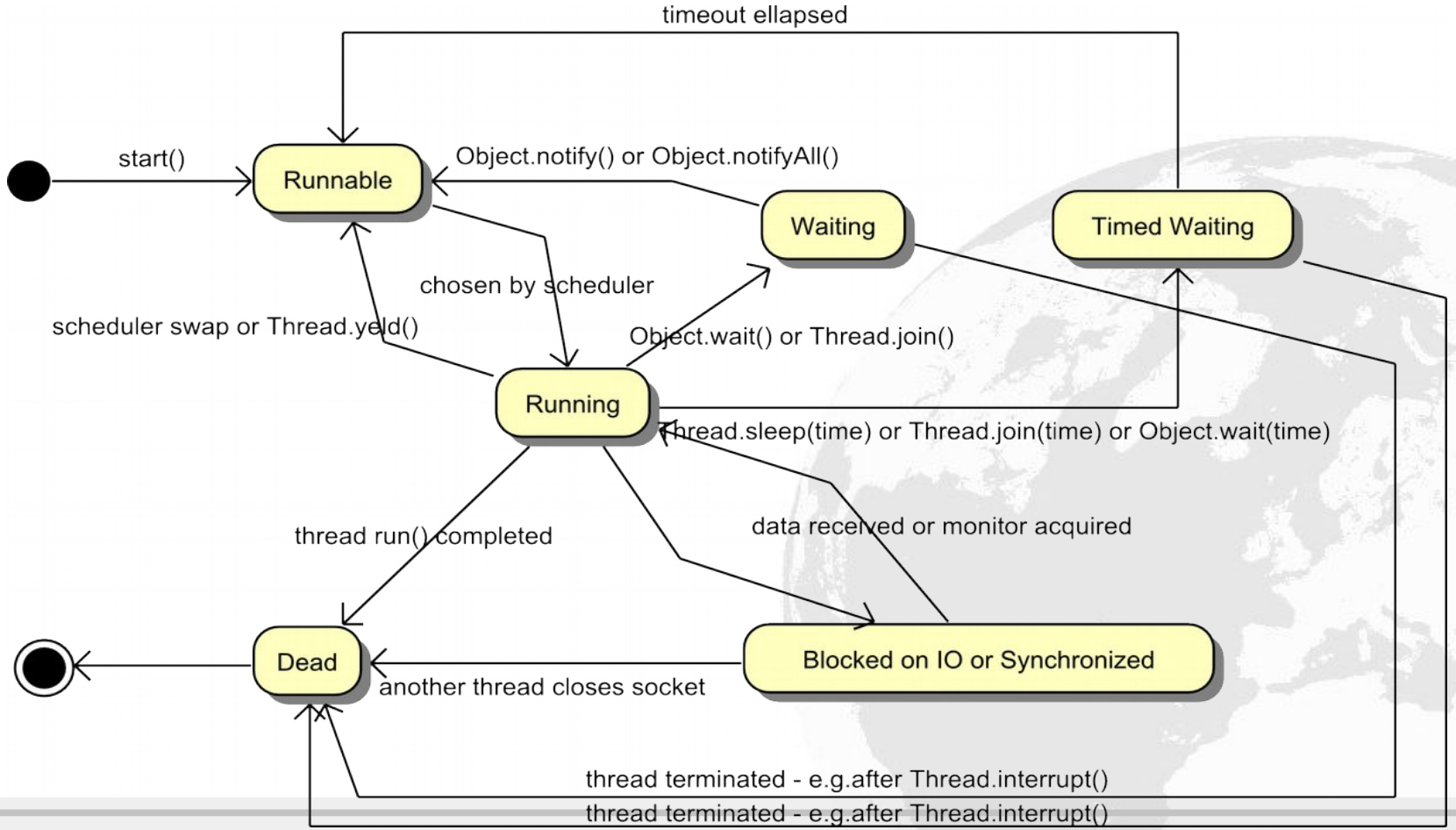
Прекъсване на нишки

- Прекъсване на Thread – thread.interrupt();
- Методи генериращи InterruptedException:
  - Object.wait()
  - Thread.join()
  - Thread.sleep(long msecs)
- IOStream блокиращите методи не генерират InterruptedException
- Thread.interrupted() - връща и изчиства interruption status

Състояния на нишка, блокиране

- Състояния на нишка
  - **New** – нишката все още не е стартирана
  - **Runnable** – нишката е готова за изпълнение от JVM
  - **Blocked** – нишката чака да получи монитор или IO
  - **Waiting** – нишката освобождава мониторите и чака
  - **Timed Waiting** – нишката чака максимум (timeout)
  - **Dead** – нишката е завършила своето изпълнение
- Причини за блокиране
  - входно-изходни операции
  - синхронизирани методи
  - изчакване и заспиване – **sleep(), join()** и **wait()**

Състояния на нишка - UML



Състояния на нишка и промяна на състоянието – wait(), notify(), notifyAll() I

- Правилни/неправилни начини за блокиране/разбликиране
  - Правилни: wait() - notify() - освобождава ресурсите преди да блокира нишката
  - wait() трябва винаги да бъде в: while (!condition) object.wait();
  - wait() и notify() / notifyAll() трябва да бъдат извиквани в синхронизиран блок по същия обект
  - Неправилно: stop(), suspend(), resume() - не освобождават заетите ресурси => Deadlock - deprecated

Състояния на нишка и промяна на състоянието – wait(), notify(), notifyAll() II

```
TaskA:
synchronized(sharedMonitor) {
    <setup condition for TaskB>
    sharedMonitor.notify();
}
TaskB:
synchronized(sharedMonitor) {
    while(!condition)
        sharedMonitor.wait();
} }
```

Atomic lock-free thread-safe програмиране чрез променлива – java.util.concurent.atomic I

- java.util.concurent.atomic – набор от класове, които поддържат thread-safe програмиране без заключване (lock-free) чрез използване на една единствена променлива
- Класовете разширяват концепцията за volatile стойности и масиви до такива, които предоставят операция за атомарно условно обновяване:

```
boolean compareAndSet(expectedValue, updateValue);
```

```
class SerialNumberGenerator{
  private final AtomicLong serialNumber = new AtomicLong(0);
  public long next() {
    return serialNumber.getAndIncrement();
  } }
```

Atomic lock-free thread-safe програмиране чрез променлива – java.util.concurent.atomic II

- AtomicBoolean
- AtomicInteger
- AtomicLong
- AtomicReference
- AtomicMarkableReference
- AtomicStampedReference
- AtomicIntegerArray
- AtomicLongArray
- AtomicReferenceArray
- AtomicIntegerFieldUpdater
- AtomicLongFieldUpdater
- AtomicReferenceFieldUpdater

Locks и Conditions: ReentrantLock, ReentrantReadWriteLock, Condition I

- java.util.concurent.lock – осигурява рамка за заключване и изчакване по дадено условие != synchronized + монитори
- Позволява по-голяма гъвкавост при използването на заключвания и условия (locks и conditions)
- Locks, които не се блокират при опит за заключване:
  - tryLock() - не се блокира ако заключването не може да се осъществи веднага или за определен период (timeout)
  - lockInterruptibly() - разблокира се, когато друга нишка прекъсне текущата, преди да се осъществи заключване
- newCondition() - създава условия (Conditions), по които се изчаква wait/signal (обектът трябва да притежава заключването както при wait()/notify())

Locks и Conditions: ReentrantLock, ReentrantReadWriteLock, Condition II

- Типичен шаблон за използване:

```
class MyClass {
    private final ReentrantLock lock = new ReentrantLock();
    public void doSomething() {
        lock.lock();  // block until condition succeeds
        try {
            // method body operations
        } finally {
            lock.unlock();
        }
    }
}
```

Locks и Conditions: ReentrantLock, ReentrantReadWriteLock, Condition III

- ReadWriteLock – осигурява двойка от свързани заключвания (locks), едното от които е за четене (споделяемо – shareable), а другото за писане (несподеляемо – exclusive)
- Позволява по постигане на по-високи нива на конкурентност при достъпа до споделени данни, отколкото при използване на обикновено заключване (mutual exclusion lock)
- Пример:

  ...

Новости в Java™ 7: Fork – Join Framework

```
if (размер_на_задачата < прагова_стойност)
    изпълняваме_задачата_директно;
else {
    разделяме_задачата_на_две_или_повече_подадачи;
    fork-ваме_отделните_подзадачи;
    join-ваме_когато_подзадачите_завършат;
}
```

- Основни класове: **ForkJoinPool.*invoke(*ForkJoinTask*)***
- **ForkJoinTask** се наследява от **RecursiveTask** (връща резултат) и **RecursiveAction** (**не** връща резултат)

Новости в Java™ 8: **parallelStream()** (uses Fork – Join Framework)

- В **Java 8** използването на **ForkJoin** става напълно прозрачно за програмиста - например:

```
Pattern pat = Pattern.compile(keywordsRegex);
itemsList.parallelStream().flatMap((Item i) -> {
    Map<Item, Integer> matches = new HashMap<>();
    Matcher m = pat.matcher(i.getDescription());
    while(m.find())
        matches.put(i, m.start());
    return matches.entrySet().stream();
}).forEach(System.out::println);
```

Задача за производители и консуматори (Producer-Consumer Problem)

- Задачата е да се синхронизира работата на един или повече производители (Producers) с тази на един или повече консуматори (Consumers) на някакви елементи или съобщения
- Съществуват различни подходи за решаване на задачата:
  - с използване на wait(), notify(), notifyAll()
  - Locks and Conditions (java.util.concurent.lock)
  - BlockingQueues / BlockingDequeues
  - Semaphores
  - Channels (Pipes) – PipedInputStream, PipedOutputStream, PipedReader, PipedWriter

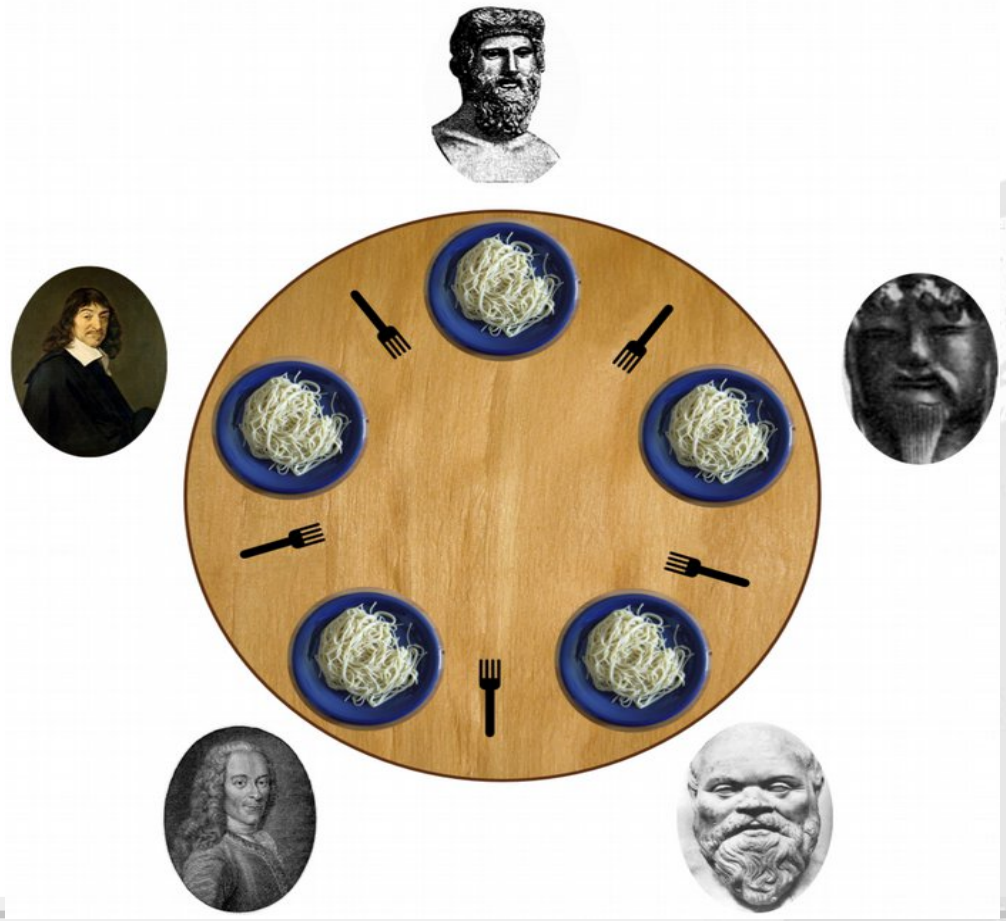
Инструменти за конкурентно програмиране в java.unitl.concurrent

- BlockingQueues
- BlockingDeques
- PriorityBlockingQueue
- DelayQueue
- SynchronousQueue
- CountDownLatch
- CyclicBarrier
- Exchanger
- Semaphore
- Comparing mutex technologiesMotivations

Deadloack и Livelock

- Дефиниция: Ако няколко процеса се опитват да влязат в критичните си секции, тогава един от тях трябва евентуално да успее. В противен случай имаме deadlock, където някои процеси са блокирани взаимно завинаги.
- Deadlock е ситуация, при която две или повече конкурентни дейности се чакат една друга да завършат и така никоя не успява. Пример за „кокошката или яйцето“.
- Ситуация, при която една нишка чака да бъде събудена по дадено условие, но събуждайки се открива, че друга нишка е инвертирала условието отново. Първата нишка е принудена да изчака отново. Когато това се случва безкрайно, нишката е в Livelock

Задача за обядващите философи



Източник: Wikipedia – http://commons.wikimedia.org/wiki/File:Dining_philosophers.png
Автор: Benjamin D. Esham и др., Лиценз: Creative Commons Attribution-Share Alike 3.0 Unported

Литература и интернет ресурси I

- Java Concurrency in Practice by Brian Goetz, with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea. Addison-Wesley, 2006 – http://jcip.net/
- Object Oriented System Development, by Dennis de Champeaux, Doug Lea, and Penelope Faure, originally published in 1993 by Addison-Wesley – http://gee.cs.oswego.edu/dl/oosdw3/index.html
- Concurrent Programming in Java: Design Principles and Patterns, (second edition) published November 1, 1999 by Addison-Wesley – http://gee.cs.oswego.edu/dl/cpj/index.html
- Thinking in Java. 3-rd ed., Eckel, B., Prentice Hall, 2002 – http://www.mindview.net/Books/TIJ
- Thinking in Java. 4-th ed., Eckel, B., Prentice Hall, 2006 – http://mindview.net/Books/TIJ4

Литература и интернет ресурси II

- The Java™ Language Specification (third edition), James Gosling, Bill Joy, Guy Steele and Gilad Bracha, Addison-Wesley, 2005 – http://java.sun.com/docs/books/jls/
- Oracle®/ Sun Microsystems Java™ Technologies webpage – http://java.sun.com/
- Effective Java Second Edition, Bloch, J., Sun Microsystems, 2008
- Principles of Concurrent and Distributed Programming (Second edition), Ben-Ari, M., Addison-Wesley, 2006
- Concurrent Programming with J2SE 5.0 http://java.sun.com/developer/technicalArticles/J2SE/concurrency/
- Tutorial - Concurrency Lesson – http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html

Благодаря Ви за вниманието!

Въпроси?