Corso di Programmazione Concorrente

`java.util.concurrent`

Valter Crescenzi

http://crescenzi.inf.uniroma3.it

Sommario

- Introduzione a **`java.util.concurrent`**
- **`java.util.concurrent.Lock/Condition/Semaphore`**
- Buffer Circolare
- Estensioni al Java Collection Framework
  - **`java.util.`*`Queue`***
  - **`java.util.concurrent.`*`BlockingQueue`***
  - **`java.util.`*`Deque`***
  - **`java.util.concurrent.`*`BlockingDeque`***
- **`java.util.concurrent.-`**
  - **`CyclicBarrier`**
  - **`CountDownLatches`**
  - **`Exchanger`**

`java.util.concurrent`: Motivazioni

- Package ideato da un gruppo di esperti sotto la guida di Doug Lea (JSR166)
- La piattaforma java, nonostante prevedesse dei meccanismi per la PC sin dalla nascita, non ha poi fatto significativi passi in avanti
- I meccanismi “nativi” che mette a disposizione sono piuttosto primitivi e rigidi; necessità di astrazione

- Obbiettivi (“dichiaratamente immodesti”): creare una nuova libreria che rappresenti per la PC, ciò che il *Java Collections Framework* aveva rappresentato per le collezioni

J2SE 5.0 e la PC

- Package di principale interesse
  - **`java.util.concurrent`**
    - framework per la scrittura di codice concorrente
  - **`java.util.concurrent.locks`**
    - usuali strumenti a supporto dei problemi di competizione e cooperazione
  - **`java.util.concurrent.atomic >>`**
    - classi wrapper che offrono operazioni atomiche grazie ad istruzioni del tipo *TestAndSet*, qui chiamate *CompareAndSet* (CAS)
    - Richiesero uno specifico al supporto della JVM

- alcune novità hanno causato cambiamenti ed aggiunte anche in altri package, in particolare **`java.util`**

I Lock:
`java.util.concurrent.locks.`*`Lock`*

Method Summary	
void	**lock**() Acquires the lock.
void	**lockInterruptibly**() Acquires the lock unless the current thread is interrupted.
Condition	**newCondition**() Returns a new Condition instance that is bound to this Lock instance.
boolean	**tryLock**() Acquires the lock only if it is free at the time of invocation.
boolean	**tryLock**(long time, TimeUnit unit) Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted.
void	**unlock**() Releases the lock.

Le Variabili Condizione: `java.util.concurrent.locks.Condition`

Method Summary	
void	**await**() Causes the current thread to wait until it is signalled or interrupted.
boolean	**await**(long time, TimeUnit unit) Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.
long	**awaitNanos**(long nanosTimeout) …
void	**awaitUninterruptibly**() Causes the current thread to wait until it is signalled.
boolean	**awaitUntil**(Date deadline) ….
void	**signal**() Wakes up one waiting thread.
void	**signalAll**() Wakes up all waiting threads.

Implementazioni di Lock e Condition

- **`java.util.concurrent.locks.`*`Lock`*** e **`java.util.concurrent.locks.`*`Condition`*** sono interfacce
- **`java.util.concurrent.locks.ReentrantLock`** è una implementazione di lock rientranti
  - molteplici **`lock()`** dallo stesso thread non causano sospensione
- con il metodo factory **`Lock.newCondition`**() si ottiene una implementazione *anonima* delle variabili condizione

Protocollo di Utilizzo dei Lock

```
Lock l = una implementazione di Lock;
l.lock();
try {
  <<regione critica sulla risorsa condivisa>>
}
finally {
  l.unlock();
}
```

I Semafori: `java.util.concurrent.Semaphore`

Constructor Summary
Semaphore(int permits) Creates a Semaphore with the given number of permits and nonfair fairness setting.

Method Summary	
void	**acquire**(int permits) Acquires the given number of permits from this semaphore, blocking until all are available, or the thread is interrupted.
int	**availablePermits**() Returns the current number of permits available
void	**release**(int permits) Releases the given number of permits, …
boolean	**tryAcquire**(int permits) Acquires the given number of permits from this semaphore, only if all are available at the time of invocation.

Buffer Circolare (1)

- Sviluppiamo un esempio di utilizzo dell'API
- Buffer circolare di dimensione finita acceduto da molteplici thread
  - N.B. la classe **`ArrayBlockingQueue`** fornisce le medesime funzionalità
  - Per motivi puramente didattici
    - *Rarissimamente* ha senso reimplementarsi queste strutture dati
- Utilizziamo
  - Lock
  - Variabili Condizione

Buffer Circolare (2)

```
import java.util.concurrent.*;
public class BoundedBuffer<Data> {

  private Data buffer[];
  private int first;
  private int last;
  private int numberInBuffer;
  private int size;
  private Lock lock = new ReentrantLock();
  private final Condition notFull =
                lock.newCondition();
  private final Condition notEmpty =
                lock.newCondition();
  //…
```

Buffer Circolare (3)

```
//…
public BoundedBuffer(int length) {
   size = length;
   buffer = (Data[]) new Object[size];
   last = 0;
   first = 0;
   numberInBuffer = 0;
}
//…
```

Buffer Circolare (4)

```
//…
public void put(Data item)
          throws InterruptedException {
  lock.lock();
  try {
    while (numberInBuffer == size)
            notFull.await();
    last = (last + 1) % size;
    numberInBuffer++;
    buffer[last] = item;
    notEmpty.signal();
  } finally {
    lock.unlock();
  }
}
//…
```

Buffer Circolare (5)

```
    //…
    public Data get()
           throws InterruptedException {
    lock.lock();
    try {
      while (numberInBuffer == 0)
              notEmpty.await();
      first = (first + 1) % size ;
      numberInBuffer--;
      notFull.signal();
      return buffer[first];
    } finally {
        lock.unlock();
    }
   }
}
```

`java.util.Queue`

- Nella pratica si utilizzano direttamente le classi fornite da **`java.util.concurrent`**
- In java 1.5 ha trovato posto una specializzazione dell'interfaccia **`java.util.Collection`** per collezioni che obbediscono a politiche di ins./estr. FIFO
- Estensioni al Java Collection Framework motivata anche dalle esigenze di JSR166

Method Summary	
E	**element**() Retrieves, but does not remove, the head of this queue.
boolean	**offer**(E o) Inserts the specified element into this queue, if possible.
E	**peek**() Retrieves, but does not remove, the head of this queue, returning null if this queue is empty.
E	**poll**() Retrieves and removes the head of this queue, or null if empty.
E	**remove**() Retrieves and removes the head of this queue.

Varianti dei Metodi di Inserimento/Estrazione in ***java.util.Queue***

- 2 varianti

```
public interface Queue<E>
extends Collection<E>
```

A collection designed for holding elements prior to processing. Besides basic `Collection` operations, queues provide additional insertion, extraction, and inspection operations. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either `null` or `false`, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted `Queue` implementations; in most implementations, insert operations cannot fail.

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out). Whatever the ordering used, the *head* of the queue is that element which would be removed by a call to `remove()` or `poll()`. In a FIFO queue, all new elements are inserted at the *tail* of the queue. Other kinds of queues may use different placement rules. Every `Queue` implementation must specify its ordering properties.

Estensioni al Java Collection Framework

- *java.util.Queue*
  - *java.util.concurrent.BlockingQueue*
    - java.util.concurrent.ArrayBlockingQueue
    - java.util.concurrent.LinkedBlockingQueue
    - java.util.concurrent.PriorityBlockingQueue
    - java.util.concurrent.SyncronousQueue
    - java.util.concurrent.DelayQueue
  - java.util.concurrent.ConcurrentLinkedQueue
  - java.util.PriorityQueue
  - java.util.LinkedList
- *java.util.Deque*
  - java.util.ArrayDeque
  - *java.util.concurrent.BlockingDeque*
    - java.util.concurrent.LinkedBlockingDeque

java.util.concurrent.BlockingQueue

- Estende ***java.util.Queue***
- Principale interfaccia per le code bloccanti da utilizzarsi in un contesto multi-thread
- E' l'interfaccia per tutte le code bloccanti e prevede, oltre ai metodi di ***Queue***, due nuove varianti delle stesse funzionalità:
  - metodi con time-out
  - metodi bloccanti >>

Metodi Bloccanti di *java.util.concurrent.BlockingQueue*

Method Summary	
boolean	**offer**(E o, long timeout, TimeUnit unit) Inserts the specified element into this queue, waiting if necessary up to the specified wait time for space to become available.
E	**poll**(long timeout, TimeUnit unit) Retrieves and removes the head of this queue, waiting if necessary up to the specified wait time if no elements are present on this queue.
boolean	**add**(E o) Adds the specified element to this queue if it is possible to do so immediately, returning true upon success, else throwing an IllegalStateException.
void	**put**(E o) Adds the specified element to this queue, waiting if necessary for space to become available.
E	**take**() Retrieves and removes the head of this queue, waiting if no elements are present on this queue.

Varianti dei Metodi di Inserimento/Estrazione in *java.util.concurrent.BlockingQueue*

```
public interface BlockingQueue<E>
extends Queue<E>
```

A `Queue` that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

`BlockingQueue` methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future: one throws an exception, the second returns a special value (either `null` or `false`, depending on the operation), the third blocks the current thread indefinitely until the operation can succeed, and the fourth blocks for only a given maximum time limit before giving up. These methods are summarized in the following table:

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	*not applicable*	*not applicable*

Implementazioni di ***BlockingQueue***

- **`java.util.concurrent.ArrayBlockingQueue`**
  - coda di dimensione finita basata su un array
- **`java.util.concurrent.LinkedBlockingQueue`**
  - coda illimitata (opzionalmente finita) basata su una rappresentazione a lista collegata
- **`java.util.concurrent.SynchronousQueue`**
  - una coda senza buffer per la gestione di *hand-off*, ovvero trasferimento di oggetti tra diversi thread
  - non ha nemmeno la capacità di ospitare un elemento: il metodo **`peek()`** perde di significato
- **`java.util.concurrent.PriorityBlockingQueue`**
  - versione bloccante di **`java.util.PriorityQueue`**
  - coda di priorità per inserimenti ed estrazioni “ordinate”
- **`java.util.concurrent.DelayQueue`**
  - coda illimitata i cui elementi possono essere rimossi solo dopo che un “delay” associato all'elemento è trascorso

java.util.Deque

- Double Ended Queue (“Deck”)
  - Coda che supporta inserimento ed estrazione efficienti da ambo gli estremi
- La versione bloccante è molto utile per le tecniche di *work stealing*>>

- 2 varianti x 2 estremi:

	First Element (Head)		Last Element (Tail)	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

java.util.concurrent.BlockingDeque

- Estende ***java.util.Deque***
- Implementata da `java.util.concurrent.LinkedBlockingDeque`
- 4 varianti x 2 estremi (della coda)

First Element (Head)				
	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`addFirst(e)`	`offerFirst(e)`	`putFirst(e)`	`offerFirst(e, time, unit)`
Remove	`removeFirst()`	`pollFirst()`	`takeFirst()`	`pollFirst(time, unit)`
Examine	`getFirst()`	`peekFirst()`	*not applicable*	*not applicable*
Last Element (Tail)				
	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`addLast(e)`	`offerLast(e)`	`putLast(e)`	`offerLast(e, time, unit)`
Remove	`removeLast()`	`pollLast()`	`takeLast()`	`pollLast(time, unit)`
Examine	`getLast()`	`peekLast()`	*not applicable*	*not applicable*

Java Collection Framework & *java.util.Map/Set*

- Classi non completamente sincronizzate:
- ***java.util.Map***
  - ***java.util.concurrent.ConcurrentMap>>***
    - **java.util.concurrent.ConcurrentHashMap>>**
    - ***java.util.SortedMap***
      - ***java.util.concurrent.ConcurrentNavigableMap***
        - **java.util.concurrent.ConcurrentSkipListMap**
- ***java.util.Set***
  - ***java.util.SortedSet***
    - ***java.util.NavigableSet***
      - **java.util.concurrent.ConcurrentSkipListSet**

Altri Strumenti di Sincronizzazione in `java.util.concurrent`

- `CountDownLatches`
- `CyclicBarrier`
- `Exchanger`

java.util.concurrent.CountDownLatches (1)

- Consente ad uno o più thread di attendere il completamento di un insieme di operazioni da parte di altri thread

Constructor Summary
CountDownLatch(int count)   Constructs a CountDownLatch initialized with the given count.

Method Summary	
void	**await()** Causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted.
boolean	**await(**long timeout**, TimeUnit** unit) Causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted, or the specified waiting time elapses.
void	**countDown()** Decrements the count of the latch, releasing all waiting threads if the count reaches zero.
long	**getCount()** Returns the current count.
String	**toString()** Returns a string identifying this latch, as well as its state.

CountDownLatches (2)

- C'è un **Driver** e diversi thread **Worker**
  - **startSignal** serve ad impedire che i **Worker** procedano prima che il **Driver** è pronto
  - **doneSignal** serve al **Driver** per attendere che tutti i **Worker** abbiano terminato

```
class Driver { // ...
  void main() throws InterruptedException {
    CountDownLatch startSignal = new CountDownLatch(1);
    CountDownLatch doneSignal = new CountDownLatch(N);

    for (int i = 0; i < N; ++i)//create/start threads
       new Thread(
           new Worker(startSignal,doneSignal)).start();

    doSomethingElse();       // don't let run yet
    startSignal.countDown();// let all threads proceed
    doSomethingElse();
    doneSignal.await();      // wait for all to finish
 }
```

CountDownLatches (3)

```
class Worker implements Runnable {
  private final CountDownLatch startSignal;
  private final CountDownLatch doneSignal;
  Worker(CountDownLatch startSignal,
         CountDownLatch doneSignal) {
     this.startSignal = startSignal;
     this.doneSignal = doneSignal;
  }
  public void run() {
     try {
       startSignal.await();
       doWork();
       doneSignal.countDown();
     } catch (InterruptedException ex) {}// return;
  }

  void doWork() { ... }
}
```

`java.util.concurrent.CyclicBarrier` (1)

- Consente di fermare e sincronizzare dei thread presso una *barriera* in attesa che *tutti* i partecipanti la raggiungono
- Utile per la decomposizione di problemi in sottoproblemi da risolvere concorrentemente

Constructor Summary
CyclicBarrier(int parties) Creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and does not perform a predefined action upon each barrier.
CyclicBarrier(int parties, **Runnable** barrierAction) Creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and which will execute the given barrier action when the barrier is tripped, performed by the last thread entering the barrier.

`CyclicBarrier` (2)

- I sottoproblemi si affidano a thread distinti
- La barriera serve per sincronizzarsi sulla loro fine e ricomporne i risultati

Method Summary	
int	**await()** Waits until all parties have invoked await on this barrier.
int	**await**(long timeout, **TimeUnit** unit) Waits until all parties have invoked await on this barrier.
int	**getNumberWaiting()** the n. of parties currently waiting at the barrier.
int	**getParties()** Returns the n. of parties required to trip this barrier.
boolean	**isBroken()** Queries if this barrier is in a broken state.
void	**reset()** Resets the barrier to its initial state.

CyclicBarrier (3)

```
class Solver {
  final int N;
  final float[][] data;
  final CyclicBarrier barrier;

  class Worker implements Runnable {
    int myRow;
    Worker(int row) { myRow = row; }
    public void run() {
      while (!done()) {
        processRow(myRow);
        try { barrier.await(); }
        catch (InterruptedException ex){ return; }
        catch (BrokenBarrierException ex){ return;}
      }
    } //…
```

CyclicBarrier (4)

```
  //…
  public Solver(float[][] matrix) {
    data = matrix;
    N = matrix.length;
    barrier =
       new CyclicBarrier(N,
                         new Runnable() {
                           public void run() {
                             mergeRows(...);
                           }
                         });
    for (int i = 0; i < N; ++i)
       new Thread(new Worker(i)).start();
    waitUntilDone();
  }
}
```

`CountDownLatches`, `CyclicBarrier`, e Testing Concorrente

- Entrambi gli strumenti possono risultare utili per realizzare degli scenari di testing con molteplici f.d.e.
- I flussi, una volta creati, si bloccano in attesa di uno stesso “segnale di via” prima di procedere
- Il segnale viene impartito dal driver del test-case
- Il test-case ha maggiori probabilità di generare sequenze di interleaving più variegate
  - N.B. In caso contrario, i f.d.e. potrebbero risultare favorite le seq. di interleaving in cui avanzano per primi i flussi creati prima

`java.util.concurrent.Exchanger` (1)

- Realizza un punto di sincronizzazione in cui due thread scambiano un oggetto
- Realizza un *hand-off bidirezionale*

Constructor Summary

Exchanger() Creates a new Exchanger.

Method Summary

V	**exchange**(V x) Waits for another thread to arrive at this exchange point (unless the current thread is interrupted), and then transfers the given object to it, receiving its object in return.
V	**exchange**(V x, long timeout, TimeUnit unit) Waits for another thread to arrive at this exchange point (unless the current thread is interrupted or the specified waiting time elapses), and then transfers the given object to it, receiving its object in return.

`Exchanger` (2)

- Si utilizza un **`Exchanger`** per lo scambio di buffers tra due thread, dei quali uno riempe il buffer con dati già elaborati e l'altro usufruisce dei dati e lo svuota

```
class FillAndEmpty {
  class FillingLoop implements Runnable {...}
  class EmptyingLoop implements Runnable {...}
   void start() {
     new Thread(new FillingLoop()).start();
     new Thread(new EmptyingLoop()).start();
   }
 }
```

Exchanger (3)

```
class FillAndEmpty {
   Exchanger<DataBuffer> exchanger = new Exchanger<DataBuffer>();
   DataBuffer initialEmptyBuffer = ... a made-up type
   DataBuffer initialFullBuffer = ...

   class FillingLoop implements Runnable {
     public void run() {
       DataBuffer currentBuffer = initialEmptyBuffer;
       try {
         while (currentBuffer != null) {
           addToBuffer(currentBuffer);
           if (currentBuffer.isFull())
             currentBuffer = exchanger.exchange(currentBuffer);
         }
       } catch (InterruptedException ex) { ... handle ... }
     }
   }
   class EmptyingLoop implements Runnable {…}}
```

Exchanger (4)

```
class FillAndEmpty {
   Exchanger<DataBuffer> exchanger = new Exchanger<DataBuffer>();
   DataBuffer initialEmptyBuffer = ... a made-up type
   DataBuffer initialFullBuffer = ...

   class FillingLoop implements Runnable {...}

   class EmptyingLoop implements Runnable {
     public void run() {
       DataBuffer currentBuffer = initialFullBuffer;
       try {
         while (currentBuffer != null) {
           takeFromBuffer(currentBuffer);
           if (currentBuffer.isEmpty())
             currentBuffer = exchanger.exchange(currentBuffer);
         }
       } catch (InterruptedException ex) { ... handle ...}
     }
   }...}
```