UNIVERSITÀ DI PISA

Programmazione di reti
Corso B

8 Marzo
Lezione 3

Rotonda

- Attenzione al problema dei lock nested
- La **rotonda e rotonda**, come il tavolo dei filosofi
- Se si condividono dei lock, un lock nested risulta in Deadlock!!!
- Chiedetevi se c’è veramente bisogno di fargli nested!

Compiti

- Se sottomettete un compito e vi accorgete di aver sbagliato, la sottomissione può essere riaperta una volta entro il deadline (email me).

- Fare upload al compito sul website non basta, dovete premere il pulsante consegna.

Domanda

```
public synchronized int getC(){
    return this.c;
}
```

```
public int getC(){
    this.cLock.lock();
    int result=this.c;
    this.cLock.unlock();
    return result;
}
```

```
public synchronized int getC(){
    this.cLock.lock();
    try{
       return this.c;
    }finally{
       this.cLock.unlock();
    }
}
```

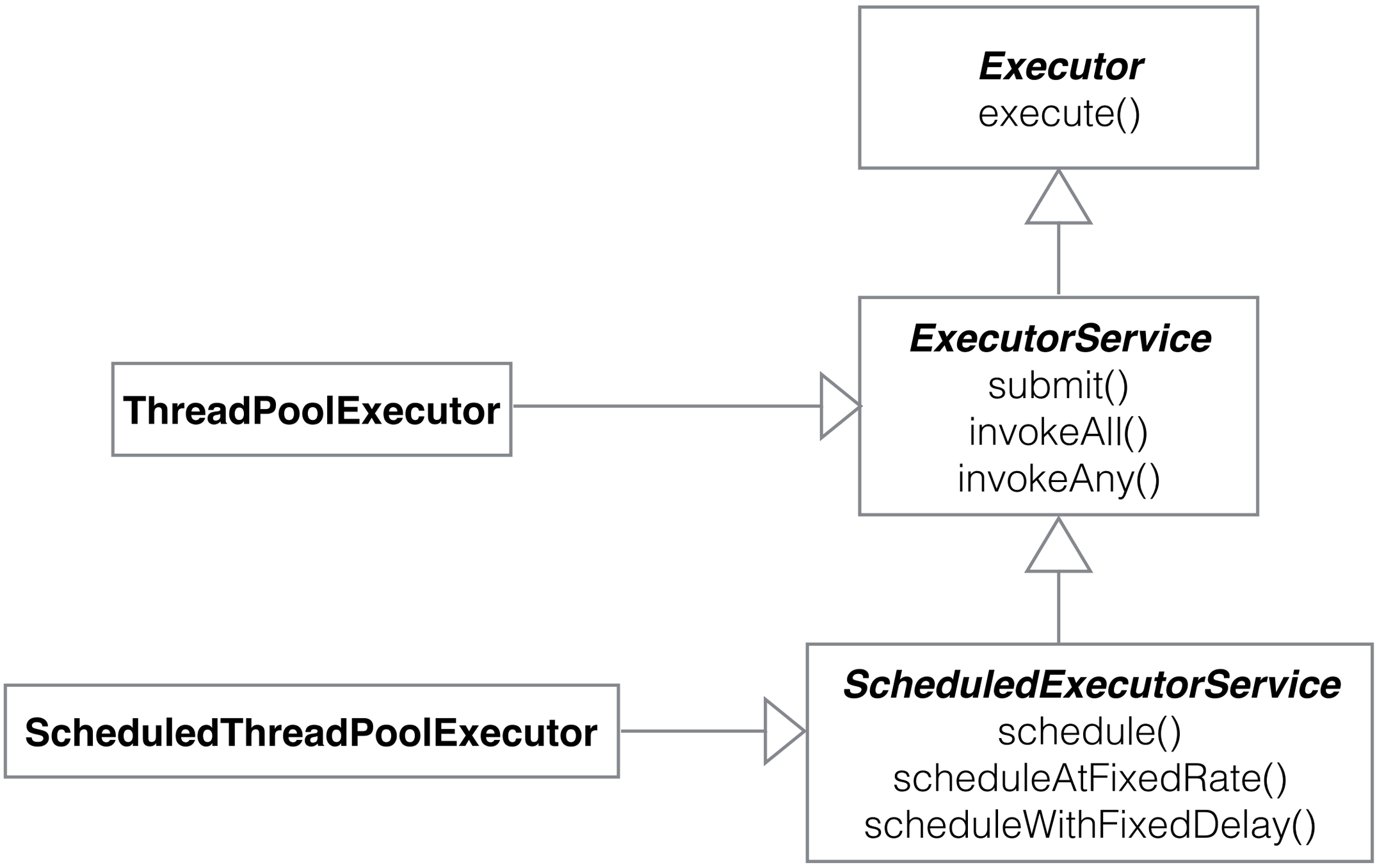
Contenuti

- Esecuzione dei *thread* controllata e indipendente dalla logica dell'applicazione
- Possibilità di restituire un risultato per un *task* e lanciare eccezioni
- *Concurrent collections*
- Variabili `Atomic`

Thread pools

- Creare un *thread* nuovo introduce un overhead
- Avere un numero di *thread* più grande di una soglia può danneggiare la *performance*
- Può essere utile limitare il numero massimo dei *thread* del mio programma
- In caso i *task* sono molti e corti, può essere utile riusare dei *thread* già creati
- La scelta dipende dal sistema usato (quanti CPU, memoria, etc) e dalla applicazione (e.g. quanti utenti sono previsti)
- In Java 5 sono introdotti gli *executor*, che ci aiutano a creare dei *thread pool*

Gerarchia di *executor*



Interfaccia `Executor`

- Ambiente controllato di esecuzione dei *task*
- Alternativa a creare oggetti `Thread` in tutti i programmi già presentati
- Dei *task* vengono inviati agli *executor* che li gestiscono automaticamente - metodo `void execute(Runnable task)`
- Per ogni *executor* attivo, ci possono essere *task* sottomessi in attesa (una coda), e *task* in esecuzione

```
Runnable task;
….
(new Thread(task)).start()
```

```
Runnable task;
Executor e;
….
e.execute(task);
```

Interfaccia `ExecutorService`

- Vari altri metodi, oltre `execute()`
- Implementata da `ThreadPoolExecutor`
- Gli *executor* devono essere chiusi all fine, altrimenti il programma non si ferma
- metodo `void shutdown()`
  - *L'executor* non accetta *task* nuovi. I *task* gia sottomessi finiscono (sia quelli in coda che quelli in esecuzione).

Interfaccia `ExecutorService`

- metodo `List<Runnable> shutdownNow()`
  - L'*executor* non accetta *task* nuovi e cancella i *task* in coda. Invia un'interruzione ai *task* in esecuzione (non è detto che questi si spengono, devono essere in grado di gestire l'interruzione).
    - restituisce tutti i `Runnable` che non hanno iniziato la loro esecuzione (quelli in coda, cancellati)
- metodo `boolean awaitTermination(long timeout, TimeUnit unit)`
  - si blocca fin che tutti i *task* finiscono dopo aver richiamato `shutdown` o `shutdownNow`, o fin che il *timeout* passa

Classe `Executors`

- *Factory* per vari tipi di *executor*
- `ExecutorService Executors.newFixedThreadPool(int n)` - crea un *pool* con numero di *thread* fisso
- `ExecutorService Executors.newCachedThreadPool()` - crea un *pool* con numero di *thread* variabile - possibilmente molto grande - riusa i *thread* esistenti quando possibile
- `ExecutorService Executors.newSingleThreadExecutor()` - *executor* con un solo *thread* - può sostituire il semplice `Thread`. Può eseguire più *task*.

Vendita biglietti

Sostituiamo il semaforo con un *thread pool* fisso

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TicketSalesMain {

   public static void main(String[] args) {
      ExecutorService ticketCounters= Executors.newFixedThreadPool(5);
      for (int i =0;i<20;i++){
         ticketCounters.execute(new Traveler(i));
      }
      ticketCounters.shutdown();

   }
}
```

```
public class Traveler implements Runnable{
   private int id;
   public Traveler(int id){
      this.id=id;
   }

   private void print(String message){
      System.out.println(System.currentTimeMillis()+" Traveler "+this.id+
            " in thread "+Thread.currentThread().getId()+": "+message);
   }

   @Override
   public void run() {
      Random rand = new Random(System.currentTimeMillis());
      try{
         this.print("I am buying my ticket");
         Thread.sleep(rand.nextInt(1000));
         this.print("Got my ticket");
      }catch (InterruptedException e){
         this.print("Something went wrong");
      }
   }
}
```

ExecutorService

- Metodi `submit()`

```
Future<T> submit(Callable<T> task)
Future<?> submit(Runnable task)
Future<T> submit(Runnable task, T result)
```

- task può essere di due tipi : `Runnable` o `Callable`
- Viene restituito un risultato di tipo `Future`

`Callable<T>`

- Simile a **`Runnable`**, definisce un *task*
- Classe *Template*/parametrica
- Interfaccia: un metodo che sostituisce **`run()`**

```
T call() throws Exception
```

  - restituisce un risultato di tipo `T`
  - può lanciare eccezioni

Future<T>

- Restituito da `ExecutorService.submit()`
- Contiene informazioni sul *status* del *task* e agevola l'interruzione del *task*

```
boolean isDone()

boolean isCancelled()

boolean cancel(boolean mayInterruptIfRunning)
```

Future<T>

- Memorizza anche il risultato del *task*

```
T get()

T  get(long timeout, TimeUnit unit)
```

- Si blocca fin che il *task* finisce
- Se il *task* era un `Callable`, restituisce il risultato del metodo `call()`
- Se il *task* era un `Runnable`
  - `submit(Runnable task, T result)`: `get()` restituisce `result`
  - `submit(Runnable task)` : `get()` restituisce `null`
- Lancia `ExecutionException` se `call()` lancia un'eccezione

Esempio

Calcoliamo x^4+x^3+x^2 in modo distribuito (3 thread)

```
import java.util.concurrent.Callable;

public class Power implements Callable<Double> {
   private double parameter;
   private int power;

   public Power(double parameter,int power){
      this.parameter=parameter;
      this.power=power;
   }
   @Override
   public Double call() throws Exception {
      System.out.format("%d Executing power %d of %f in thread %d%n",
            System.nanoTime(),this.power,this.parameter,
            Thread.currentThread().getId());
      return Math.pow(this.parameter,this.power);
   }
}
```

```
public class PowerMain {

   public static void main(String[] args) {
      double parameter= Double.parseDouble(args[0]);
      double result= 0;

      ArrayList<Future<Double>> runningTasks= new ArrayList<Future<Double>>();

      for (int i=2;i<5;i++){
         ExecutorService ex= Executors.newSingleThreadExecutor();
         runningTasks.add(ex.submit(new Power(parameter,i)));
         ex.shutdown();
      }
      try{
         for (Future<Double> t: runningTasks ){
            result+=t.get();
         }
         System.out.println("result is "+ result);
      }catch (ExecutionException e){
         System.out.println("some thread has failed");
      } catch (InterruptedException e) {
            System.out.println("interrupted while waiting");
      }
   }
}
```

```
52372790241848 Executing power 2 of 5.000000 in thread 9
52372790790907 Executing power 3 of 5.000000 in thread 10
52372791262400 Executing power 4 of 5.000000 in thread 11
result is 775.0
```

ExecutorService

- Metodi `invoke`

```
List<Future<T>> invokeAll(Collection<extends
Callable<T>> tasks)
```

Avvia i *task*, aspetta, e restituisce una lista di oggetti `Future` quando tutti i *task* sono completati

```
List<Future<T>> invokeAll(Collection<extends
Callable<T>> tasks, long timeout, TimeUnit unit)
```

Avvia i *task*, aspetta, e restituisce una lista di oggetti Future quando tutti i task sono completati o quando il *timeout* e passato. I *task* che non hanno finito quando il *timeout* e passato vengono annullati (*cancelled*).

ExecutorService

- Metodi `invoke`

```
T invokeAny(Collection<extends Callable<T>> tasks)
```

Avvia i *task,* aspetta, e restituisce il risultato di uno solo, quando un task finisce senza lanciare un'eccezione. Se nessun task finisce correttamente, lancia ExecutionException. Quando un task finisce, gli altri vengono annullati.

```
T invokeAny(Collection<extends Callable<T>> tasks,
long timeout, TimeUnit unit)
```

Come sopra però se nessun *task* finisce correttamente prima del *timeout*, lancia `TimeoutException`

```
import java.util.concurrent.Callable;

public class ExceptionTask implements Callable<Double>{

   @Override
   public Double call() throws Exception {
      Thread.sleep(100);
      System.out.format("%d throwing an exception in thread %d%n",
            System.nanoTime(),Thread.currentThread().getId());
      throw new Exception("I only do exceptions");
   }

}
```

```
public class InvokeAnyMain {

   public static void main(String[] args) {
      ExecutorService ex=Executors.newFixedThreadPool(3);
      ArrayList<Callable<Double>> tasks= new ArrayList<Callable<Double>>();
      for (int i=0;i<5;i++){
         tasks.add(new ExceptionTask());
      }
      tasks.add(new Power(3,2));
      tasks.add(new Power(4,2));
      for (int i=0;i<5;i++){
         tasks.add(new ExceptionTask());
      }
      Double result;
      try {
         result = ex.invokeAny(tasks);
         System.out.println("Result is "+result);
      }catch (InterruptedException e) {
      }catch ( ExecutionException e) {
         System.out.println(“No thread finished correctly");;
      }finally{
         ex.shutdown();
      }
   }
}
```

```
52170453540465 throwing an exception in thread 9
52170453540885 throwing an exception in thread 10
52170453540885 throwing an exception in thread 11
52170479116342 Executing power 2 of 3.000000 in thread 11
52170481707931 Executing power 2 of 4.000000 in thread 11
Result is 9.0
```

Esempio

- Dobbiamo generare una lista di numeri aleatori molto grande
- Generare numeri aleatori è costoso
- Usiamo più *thread* per generare numeri

```java
import java.util.ArrayList;
import java.util.concurrent.Callable;
import java.util.concurrent.ThreadLocalRandom;

public class BigArray implements Callable<ArrayList<Double>> {

   private int size;

   public BigArray( int size){
      this.size=size;
   }
   @Override
   public ArrayList<Double> call() throws Exception {
      ThreadLocalRandom rand= ThreadLocalRandom.current();
      ArrayList<Double> result= new ArrayList<>(this.size);
      for (int i=0;i<this.size;i++){
         result.add(rand.nextDouble()*2-1);
      }
      return result;
   }
}
```

```
public class BigArrayMain {
   public static int THREAD_COUNT=8;
   public static int SLICE=5000000;
   public static int    N=40000000;

      public static void main(String[] args) {
      ExecutorService ex= Executors.newFixedThreadPool(THREAD_COUNT);
      try {
         long startTime=System.currentTimeMillis();

         ArrayList<BigArray> tasks= new ArrayList<>();

for (int i=0;i<N/SLICE;i++){
   tasks.add(new BigArray(SLICE));
}

if(N%SLICE>0){
   tasks.add(new BigArray(N%SLICE));
}
```

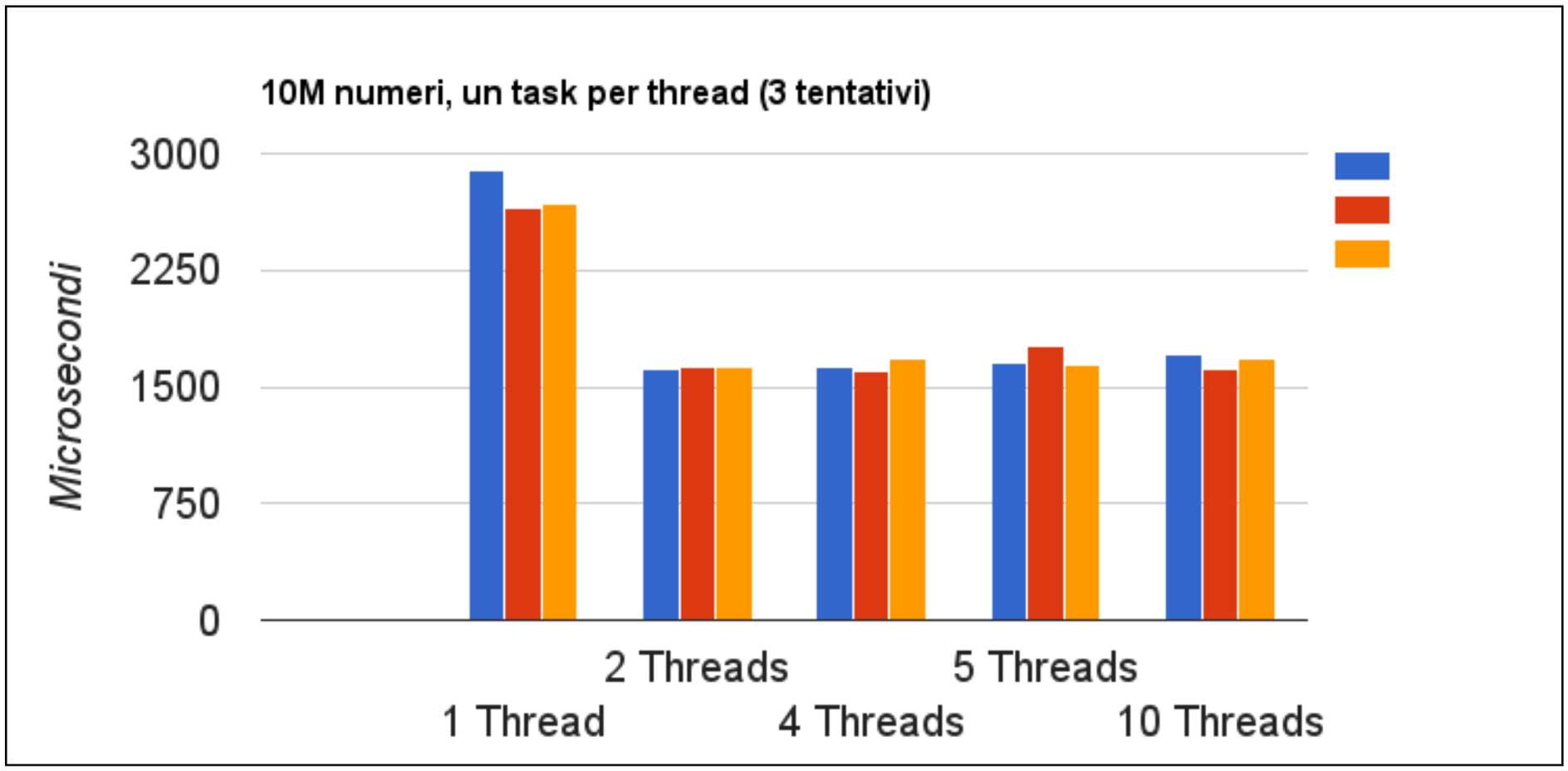
```
List<Future<ArrayList<Double>>> results = ex.invokeAll(tasks);

System.out.format("%d numbers obtained in %d "
      + "mseconds with %d threads%n",N,
      System.currentTimeMillis()-startTime, THREAD_COUNT);

for (Future<ArrayList<Double>> res :results)
{
   ArrayList<Double> array=res.get();
   //process result
         }

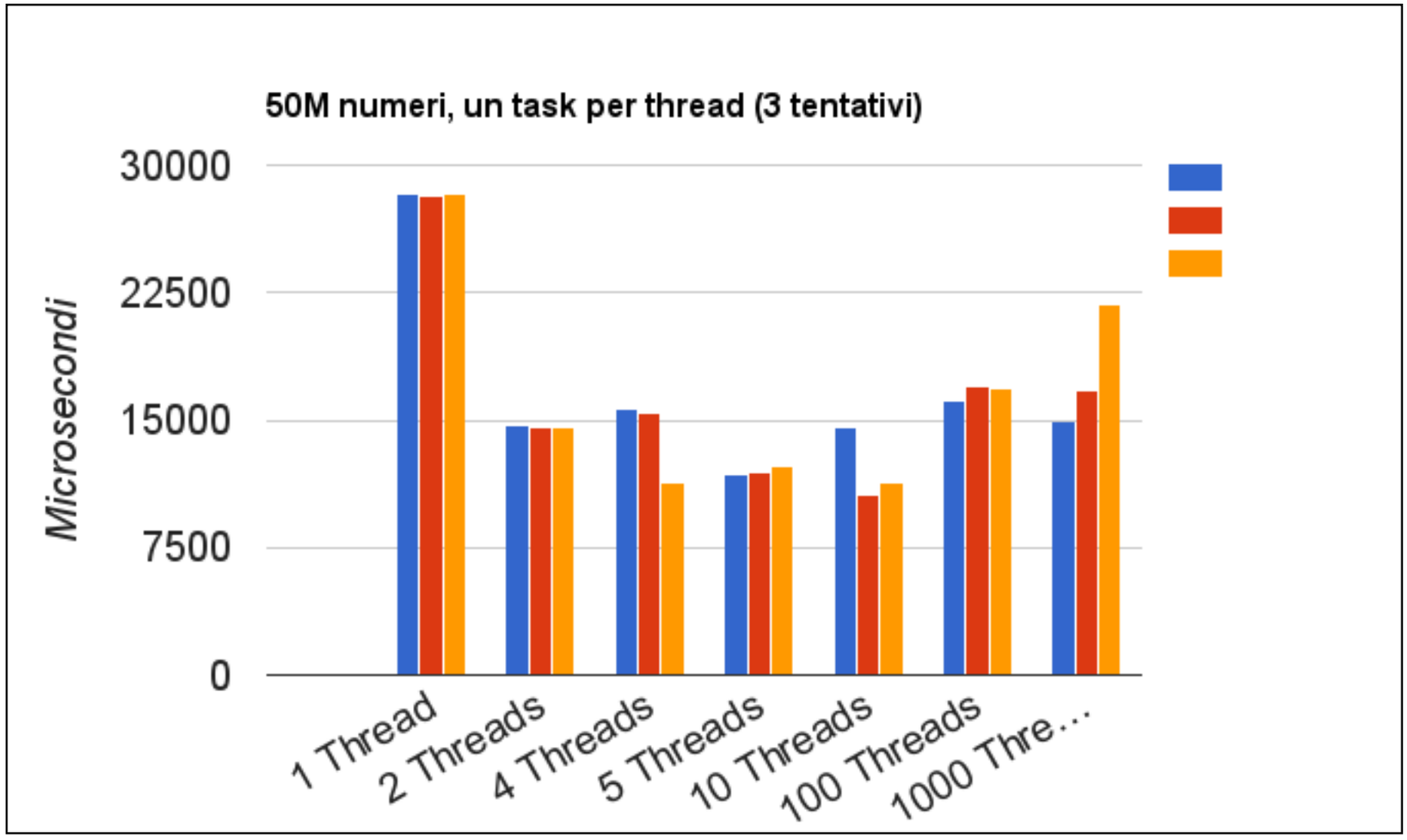
      } catch (InterruptedException e) {
      } catch (ExecutionException e) {
         System.out.println("Some thread failed");
      } finally {
         ex.shutdown();}
   }
}
```

Analisi



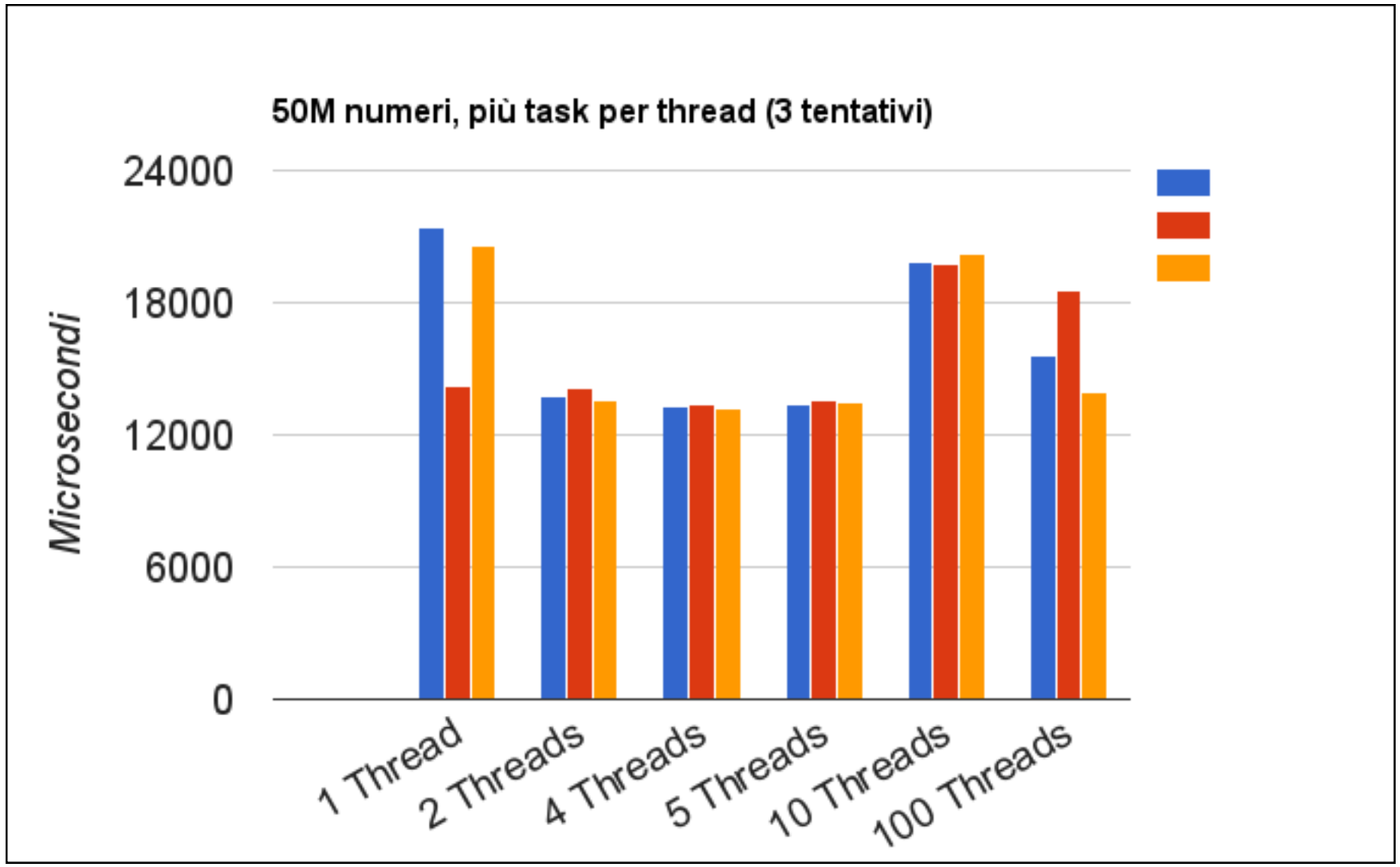
- N=10000000, THREAD_COUNT varia, SLICE=N/THREAD_COUNT
- *Speedup* da 1 a 2 *thread* : T(1)/T(2)=1.68
- No *speedup* dopo 2 *thread*

Analisi



- N=50000000, THREAD_COUNT varia, SLICE=N/THREAD_COUNT
- *Speedup* da 1 a 2 *thread* : T(1)/T(2)=1.92
- Performance migliore raggiunta con più di 4 *thread*
- Con troppi *thread* performance peggiora

Analisi



- N=50000000, SLICE=100000, THREAD_COUNT varia
- Performance migliore per 1 e 2 *thread*, in confronto con il caso SLICE=N/THREAD_COUNT
- Performance peggiora per più di 4 *thread*

Risultati finali

- `invokeAll` utile quando dobbiamo aspettare tutti i risultati intermedi e poi calcolare il risultato finale

- A volte sarebbe utile cominciare a calcolare il risultato finale appena i primi *thread* finiscono

- La classe `ExecutorCompletionService`

`ExecutorCompletionService<T>`

- Gestisce l'esecuzione dei *task* usando un `Executor`
- Usa una coda per mettere i risultati quando i *task* completano
- Costruttore :

```
ExecutorCompletionService<T>(Executor executor)
```

- Avviare *task* (simile ai `ExecutorService`):

```
submit(Callable task);

submit(Runnable task, T result);
```

ExecutorCompletionService<T>

- Processare risultati:

```
Future<T> take();
```

Si blocca fin che un *task* finisce, poi restituisce il `Future` del *task* finito

```
Future<T> poll();
```

Non si blocca, restituisce un `Future` o `null` se nessun *task* ha finito.

```
Future<T> poll(long timeout, TimeUnit unit);
```

Si blocca fin che un task finisce o il *timeout* e trascorso, poi restituisce un `Future` o `null` se nessun *task* ha finito.

```
public class BigArrayMainEcs {

  public static int THREAD_COUNT=5;
  public static int SLICE=100000;
  public static int      N=50000000;

  public static void main(String[] args) {
      ExecutorService ex= Executors.newFixedThreadPool(THREAD_COUNT);
      ExecutorCompletionService<ArrayList<Double>> ecs =
                                              new ExecutorCompletionService<>(ex);
      try {
         for (int i=0;i<N/SLICE;i++){
            ecs.submit(new BigArray(SLICE));
         }
         if(N%SLICE>0){
            ecs.submit(new BigArray(N%SLICE));
         }
         for (int i=0;i<(N/SLICE+Math.signum(N%SLICE));i++){
            ArrayList<Double> array=ecs.take().get();
            //process result
         }
    } catch (InterruptedException e) {
    } catch (ExecutionException e) {System.out.println("Some thread failed");
    } finally {ex.shutdown();}}}
```

`ScheduledExecutorService`

- Interfaccia che estende `ExecutorService`
- Possibilità di avviare i *task* con ritardo
- Possibilità di avviare un *task* alla volta a intervalli di tempo definiti
- Metodi *factory*:
  - `Executors.newScheduledThreadPool(int corePoolSize)`
  - `Executors.newSingleThreadScheduledExecutor()`

ScheduledExecutorService

```
ScheduledFuture<V> schedule(Callable<V> task, long delay, TimeUnit
unit)
ScheduledFuture<?> schedule(Runnable task, long delay, TimeUnit unit)
```

Avvia il *task*, che diventa attivo dopo il `delay`

```
ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long
initialDelay, long period, TimeUnit unit)
```

Il *task* è avviato dopo `initialDelay`, poi riavviato dopo `initialDelay`+`period`, `initialDelay`+2*`period`, etc. Se il *task* precedente non è finito, riavvio più tardi (i *task* non eseguono in contemporanea).

```
ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long
initialDelay, long delay, TimeUnit unit)
```

Il *task* è avviato dopo `initialDelay`. Quando finisce, un nuovo *task* è riavviato dopo `delay`, etc. La sequenza si ripete fino che il *task* viene cancellato o l'*executor* viene chiuso.

`ScheduledFuture`

- Come `Future`, però con un metodo addizionale

```
long getDelay(TimeUnit unit)
```

- Restituisce il delay rimasto per il *task*

Collections

- **Java 1.0 :**
  - `Vector, Hashtable` sono sincronizzate - *thread safe*
    - ogni accesso *read/write* ad elementi e proprietà avviene dopo che si chiude il monitor della collezione stessa, automaticamente
  - `ArrayList, HashMap`, etc non sincronizzate - non *thread safe*
    - se usiamo in un programma *multithreaded* dobbiamo proteggere manualmente

Collections

- **Java 1.2 :**
  - Classe factory `Collections`
    - crea *wrapper* sincronizzati di collezioni non-sincronizzati

```
List<T> Collections.synchronizedList(List<T> list)

Map<K,T> Collections.synchronizedMap(Map<K,T> map)

Set<T> Collections.synchronizedSet(Set<T> set)

SortedMap<K,T> Collections.synchronizedSortedMap(SortedMap<K,T> map)

SortedSet<T> Collections.synchronizedSortedSet(SortedSet<T> set)
```

Collections

- Collezioni sincronizzate: funzionano in programmi *multithreaded* senza dover implementare i meccanismi manualmente ogni volta (reinventare la ruota)

    - lo stato della collezione non diventa mai invalido

    - Se usiamo l'iteratore di una collezione sincronizzata, un'eccezione unchecked viene lanciata quando si osserva una modifica esterna durante l'iterazione

    - attenzione però alle operazioni composite

```
Vector<Integer> v= new Vector<>();
int size= v.size();
int lastElement=v.get(size-1);
```

Possibile che lanci eccezione se nel frattempo un elemento e stato rimosso

dobbiamo sincronizzare esternamente usando `synchronized(v)`

Collections

- A volte le collezioni sincronizzate diventano molto lente
    - se le operazioni composite prendono molto tempo perdiamo in concorrenza (altri *thread* si bloccano)

- Java 5 : *ConcurrentCollections*

Concurrent Collections

- Progettate specialmente per il vero *multithreading*
- `ConcurrentHashMap` - versione concorrente di `HashMap`
- `CopyOnWriteArrayList` - versione concorrente di `List`
- `CopyOnWriteArraySet` - versione concorrente di `Set`
- Molto più veloci delle collezioni sincronizzate nel ambiente *multithreading*
- `BlockingQueue` - una coda bloccante, utile per il problema *Producer/Consumer*

ConcurrentHashMap

- Operazioni sincronizzate usando *lock striping*
  - *Lock* dettagliato (16 *lock* diversi)
    - Tanti *reader* possono leggere in contemporanea
    - Un numero ridotto di *writer* possono scrivere in contemporanea (16)
    - *Reader* e *writer* possono convivere

ConcurrentHashMap

- L'iteratore non lancia eccezioni, però non e garantito che venga aggiornato se la collezione è cambiata da un altro *thread*

- I metodi `size()` e `isEmpty()` diventano approssimativi

- Non può essere usato in modo esclusivo (`synchronized` non blocca l'accesso degli altri che non lo usano)

- In compenso, ci sono delle operazioni atomiche già implementate: `remove, replace, putIfAbsent`

CopyOnWriteArrayList(Set)

- Usano una copia non-modificabile della lista (del set) su cui tutti i *thread* lavorano
- Quando c'e una modifica, questa copia viene cambiata
- L'iteratore non lancia eccezioni però non viene aggiornato
- Molto utile quando l'operazione di base e l'iterazione, e le modifiche sono poche

BlockingQueue

- Coda bloccante
- `T take()`
  - Restituisce il prossimo elemento in coda. A coda vuota, si blocca fino che un elemento arriva.
- `void put(T element)`
  - Mette un elemento in coda. A coda piena, si blocca fino che un elemento viene rimosso (solo per code *bounded*). La coda non accetta elementi `null`

BlockingQueue

- `boolean offer(T element)`
  - Prova di mettere un'elemento in coda, se c'è spazio restituisce `true`, altrimenti restituisce `false`
- `T poll(long timeout, TimeUnit unit)`
  - Prova di prendere il prossimo elemento dalla coda. Se non c'e nessuno, aspetta fino a timeout. restituisce l'elemento o null se ha raggiunto il timeout.
- Molto utile per il problema *Producer/Consumer*

`BlockingQueue`

- Implementata da:
  - `ArrayBlockingQueue` : usa un *array* per memorizzare gli elementi. *Bounded*.
  - `LinkedBlockingQueue` : usa una *linked list* per gli elementi. Può essere *bounded* o *unbounded*.
  - `PriorityBlockingQueue` : coda con priorità. Elementi con la stessa priorità non seguono il principio *FIFO*. *Unbounded*.
  - `SynchronousQueue` : ogni `put` aspetta un `take` corrispondente e viceversa. Utile per sincronizzare *task*.

La classe `ThreadPoolExecutor`

- E indicato usare, quando possibile, la classe *factory* `Executors` per creare gli *executor*

- Quando si vuole un comportamento diverso, si può usare direttamente il costruttore della classe `ThreadPoolExecutor`

- ```
  ThreadPoolExecutor(int corePoolSize,
          int maximumPoolSize,
          long keepAliveTime, TimeUnit unit,
          BlockingQueue<Runnable> workQueue)
  ```
  ```

La classe `ThreadPoolExecutor`

- `int corePoolSize` : numero minimo di *thread* da costruire - non necessariamente costruiti subito alla creazione dell'*executor*

- `int maximumPoolSize` : numero massimo di *thread* - *thread* addizionali costruiti se ci sono più task di `corePoolSize`

- `long keepAliveTime, TimeUnit unit` : per quanto tempo i *thread* addizionali saranno mantenuti *alive* dopo che hanno finito il loro *task*

- `BlockingQueue<Runnable> workQueue` : coda per i `Runnable` sottomessi

Variabili `Atomic`

- Classi che permettono operazioni atomiche del tipo *read-modify-write* (e.g. i++) su variabili di vari tipi senza necessità di sincronizzare

- Usano le abilità del *hardware* moderno per raggiungere l'atomicità, quindi eliminano l'uso dei *lock* (per la maggior parte delle piattaforme)

- L'idea è di fare degli update senza *locking*, e verificare se c'è stata una *collision* - in qual caso l'operazione fallisce

- Molto veloci se la probabilità di *collision* non è molto grande

- Progettate per sistemi multi-processore

Variabili `Atomic`

- `AtomicInteger, AtomicLong, AtomicBoolean, AtomicReference`
  - `get` e `set` - simile a *read/write* su variabili volatili
  - atomic `compareAndSet`: restituisce `true` se set è stato realizzato
  - atomic `add, increment, decrement`
  - mutabili (a differenza di `Integer, Long`, etc)
- Molto utili per implementare `Counter`

```java
import java.util.concurrent.atomic.AtomicInteger;

public class Counter {
   private AtomicInteger count;

   public Counter(){
      this.count=new AtomicInteger(0);
   }

   public void increment(){
      this.count.incrementAndGet();
   }

   public int getCount() {
      return count.get();
   }
}
```

Conclusioni

- Abbiamo parlato di nuovi meccanismi per usare *thread* - solo le funzionalità di base - consultare la documentazione per la lista completa di classi e metodi
- I nuovi meccanismi semplificano la sincronizzazione esplicita (in molti casi diventa intrinseca al oggetto)
  - Usando `Callable` non serve sincronizzare il risultato
  - Usando classi `Atomic` non serve sincronizzare le operazioni definite da loro come atomiche
  - Usando *synchronized collections* non serve sincronizzare operazioni semplici su collezioni
  - Usando *concurrent collections* non serve sincronizzare ne operazioni semplici ne composite su collezioni
- Programmi molto più puliti, performance e scalabilità migliorate
- Bisogna solo sapere quando usare ogni classe

Esercizi

- Scrivere un programma che usa *multithreading* per moltiplicare due matrici molto grandi. Il *thread* main
  - genera 2 matrici *random* molto grandi (tipo 1000 X 1000)
    - Costruisce degli *task* che calcolano una riga della moltiplicazione ogni uno (data come parametro al worker)
    - Usa un *thread pool* per avviare tutti i *task*
- Analizzare, sul vostro computer, quanti *thread* deve avere la *thread pool* per ottenere il risultato più veloce (simile al corso) - sottomettere l'analisi con i *file* di codice.

Esercizi

- *Producer-consumer:* la mensa.

- Simulare il lavaggio dei piatti alla mensa. Gli studenti portano i loro piatti a lavare dopo aver mangiato (mangiano in 1-5 secondi). Ci sono due azioni diverse da fare per ogni piatto: lavare e risciacquare. Ci sono 7 persone che lavorano alla mensa, 3 lavano, 4 risciacquano. Ogni uno di loro ha bisogno di 1-5 secondi per lavare/risciacquare un piatto.

- Simulate l'arrivo di 100 studenti, a intervalli di tempo aleatori. Il programma si ferma quando tutti i piatti sono lavati.

- Attenzione! Il programma non deve usare ne `synchronized` ne *locking* esplicito.