Laboratorio di Reti – A (matricole pari)

Autunno 2021,
instructor: Laura Ricci

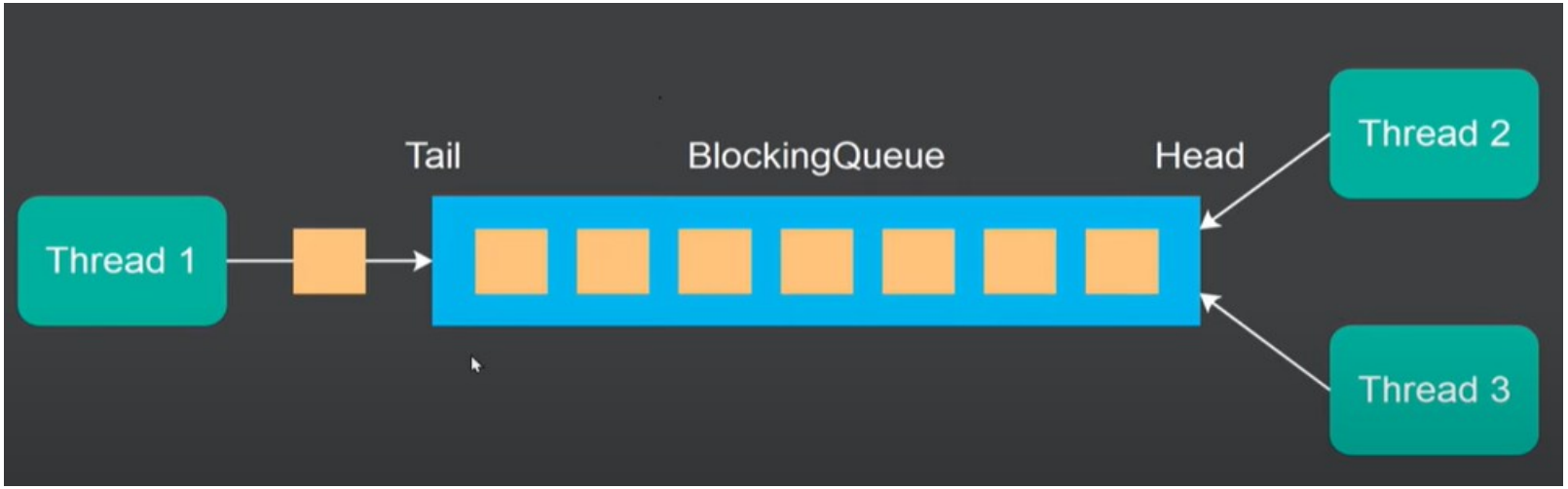
laura.ricci@unipi.it

Lezione 2

BlockingQueue, ThreadPool, Callable,

21/9/2021

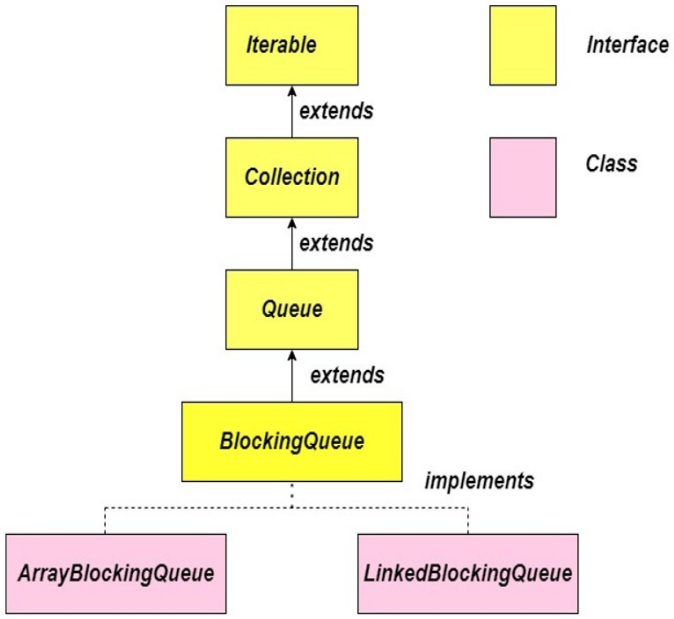
JAVA BLOCKING QUEUE



- `BlockingQueue` (`java.util.concurrent`): una JAVA interface che rappresenta una coda (inserimento alla fine, estrazione all'inizio)
- ...ma quale è la differenza con la interface `Queue<E>` (package `JAVA.UTIL`)?
  - permettere ai thread che inseriscono ed eliminano elementi dalla coda di bloccarsi
    - `Thread1` si blocca se la coda è piena, `Thread2` e `Thread3` se è vuota
  - implementa una corretta sincronizzazione tra thread

BLOCKING QUEUE: IMPLEMENTAZIONI

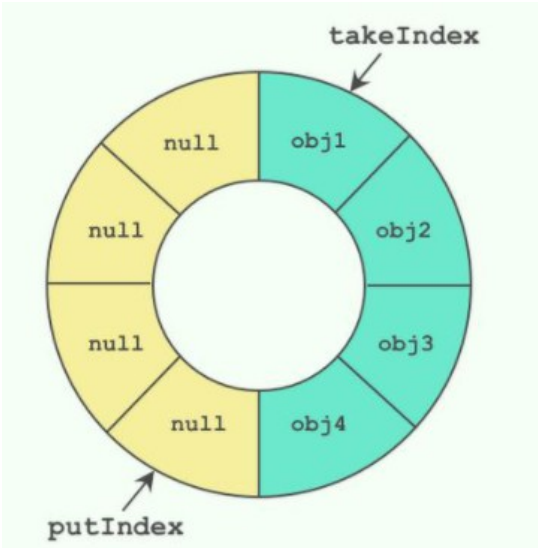
```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class BlockingQueueExample {
        public static void main(String[] args)
            {BlockingQueue arrayBlockingQueue  =
                new  ArrayBlockingQueue(3);
             BlockingQueue linkedBlockingQueue =
                new   LinkedBlockingQueue();

             // java.util.concurrent.DelayQueue
             // java.util.concurrent.LinkedTranserQueue
             // java.util.concurrent.PriorityBlockingQueue
             // java.util.concurrent.SynchronousQueue
             }}
```

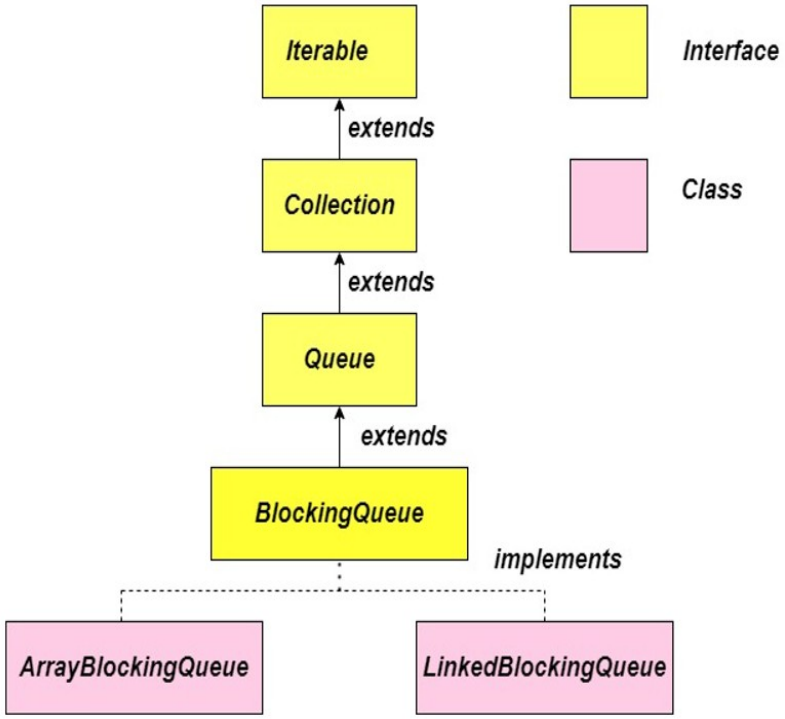


QUALI CODE UTILIZZEREMO MAGGIORMENTE?

`ArrayBlockingQueue`

- dimensione limitata, definita in fase di inizializzazione
- memorizza gli elementi all'interno di un oggetto `Array`
  - nessun ulteriore oggetto creato
  - non sono possibili inserzioni/rimozioni in parallelo (una sola lock per tutta la struttura)



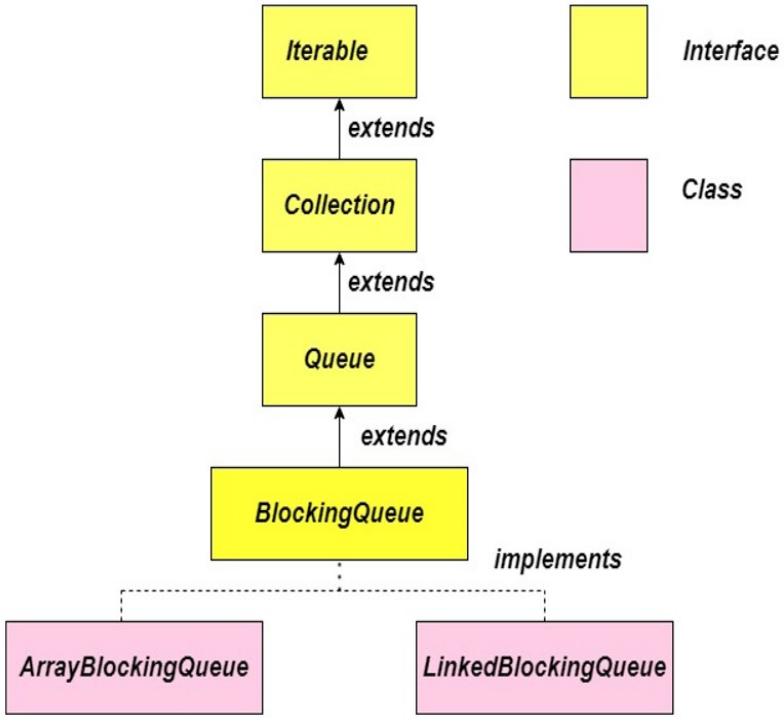


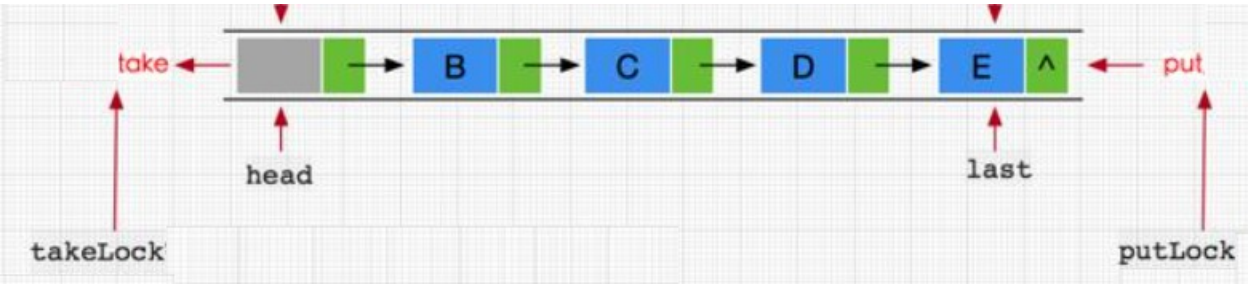
E QUALI CODE UTILIZZEREMO MAGGIORMENTE?

`LinkedBlockingQueue`

- può essere limitata o illimitata, se illimitata dimensione = `Integer.MAX_VALUE`.
- mantiene gli elementi in una `LinkedList`
  - maggior occupazione di memoria
  - un nuovo oggetto per ogni inserzione
- possibili inserzioni ed estrazioni concorrenti

  (lock separate per lettura e scrittura), maggior throughput





IL PROBLEMA DEL PRODUTTORE/CONSUMATORE

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ArrayBlockingQueue;
public class ProducerConsumerExample {

    public static void main(String[] args) {
        BlockingQueue<String> blockingQueue =
                                new ArrayBlockingQueue<String>(3);
        Producer producer = new Producer(blockingQueue);
        Consumer consumer = new Consumer(blockingQueue);
        Thread producerThread = new Thread (producer);
        Thread consumerThread = new Thread(consumer);
        producerThread.start();
        consumerThread.start(); } }
```

IL PROBLEMA DEL PRODUTTORE/CONSUMATORE

```
import java.util.concurrent.BlockingQueue;
public class Producer implements Runnable {
     BlockingQueue <String> blockingQueue = null;
     public Producer (BlockingQueue<String> queue) {
           this.blockingQueue = queue;   }
     public void run() {
         while (true) {
            long timeMillis = System.currentTimeMillis();
            try {
                this.blockingQueue.put("" + timeMillis);
            } catch (InterruptedException e) {
            System.out.println("Producer was interrupted"); }
            Sleep(1000); }}
     private static void sleep(long timeMillis) {
           try { Thread.sleep(timeMillis);
           } catch(InterruptedException e) {e.printStackTrace()}} }
```

IL PROBLEMA DEL PRODUTTORE/CONSUMATORE

```
import java.util.concurrent.BlockingQueue;
public class Consumer implements Runnable {
     BlockingQueue<String> blockingQueue = null;
     public Consumer (BlockingQueue <String> queue) {
        this.blockingQueue = queue; }
     public void run() {
         while (true) {
            try {
                String element =
                        this.blockingQueue.take();
                System.out.println("consumed: "+ element);
            } catch (InterruptedException e) {e.printStackTrace();}
       }
      }
    }
```

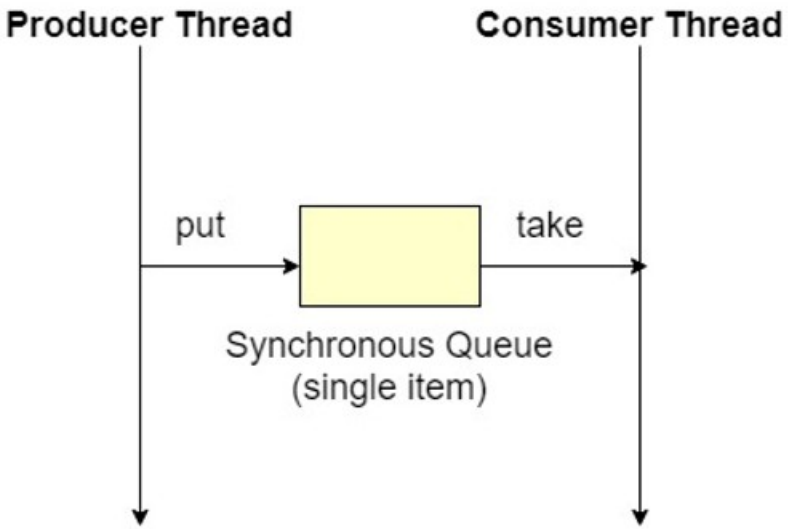
BLOCKINGQUEUE: OPERAZIONI

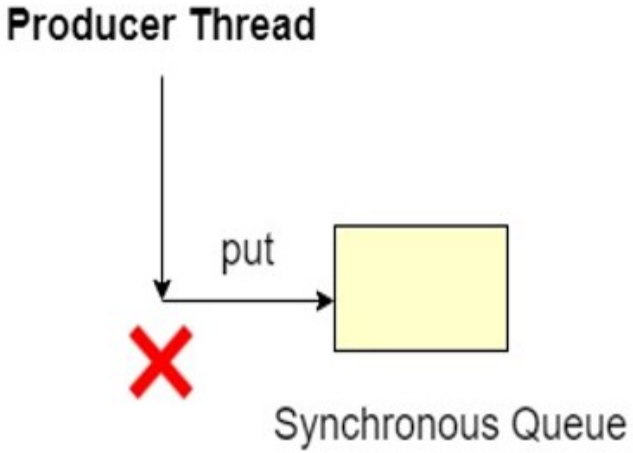
- 4 metodi diversi, rispettivamente, per inserire, rimuovere, esaminare un elemento della coda
- ogni metodo ha un comportamento diverso relativamente al caso in cui l'operazione non possa essere svolta

	Throws Exception	Special Value	Blocks	Times Out
Insert	`add(o)`	`offer(o)`	`put(o)`	`offer(o, timeout, timeunit)`
Remove	`remove(o)`	`poll()`	`take()`	`poll(timeout, timeunit)`
Examine	`element()`	`peek()`		

SYNCHRONOUS QUEUE: UNA CONTRADDIZIONE?

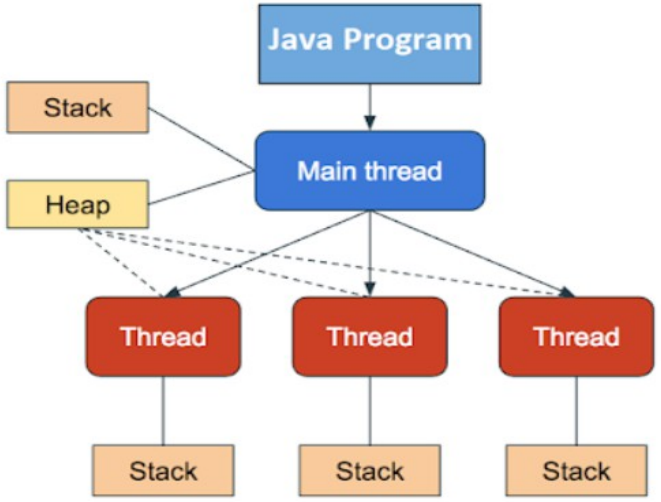
- è una `BlockingQueue` di dimensione uno
  - se il produttore tenta di inserire un elemento nella coda, anche se la coda è vuota, il produttore inserisce l'elemento e poi si blocca
- se il consumatore è pronto: "direct handoff" tra produttore e consumatore
  - in questo caso la coda non serve



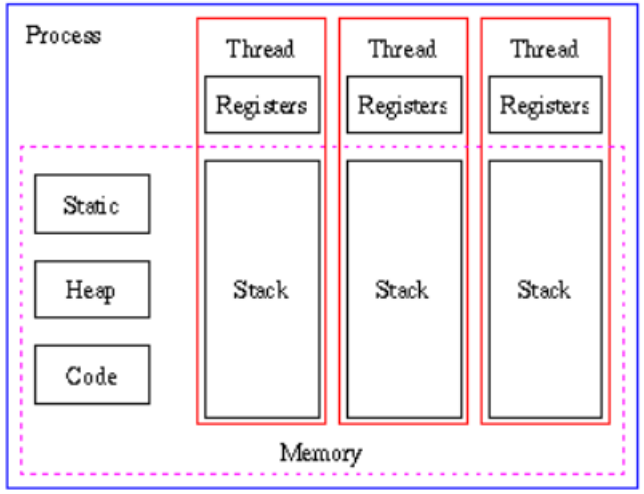


THREAD OVERHEAD

- attivazione/eliminazione di thread
  - richiede interazione tra JVM e SO
  - impatto sulle prestazioni variabile a seconda del SO
  - mai trascurabile, specie per richieste di servizio frequenti e 'lightweight'



- resource consumption
  - alloca uno stack per ogni thread
  - garbage collector stress
  - alcuni SO limitano per questo max numero di thread per programma

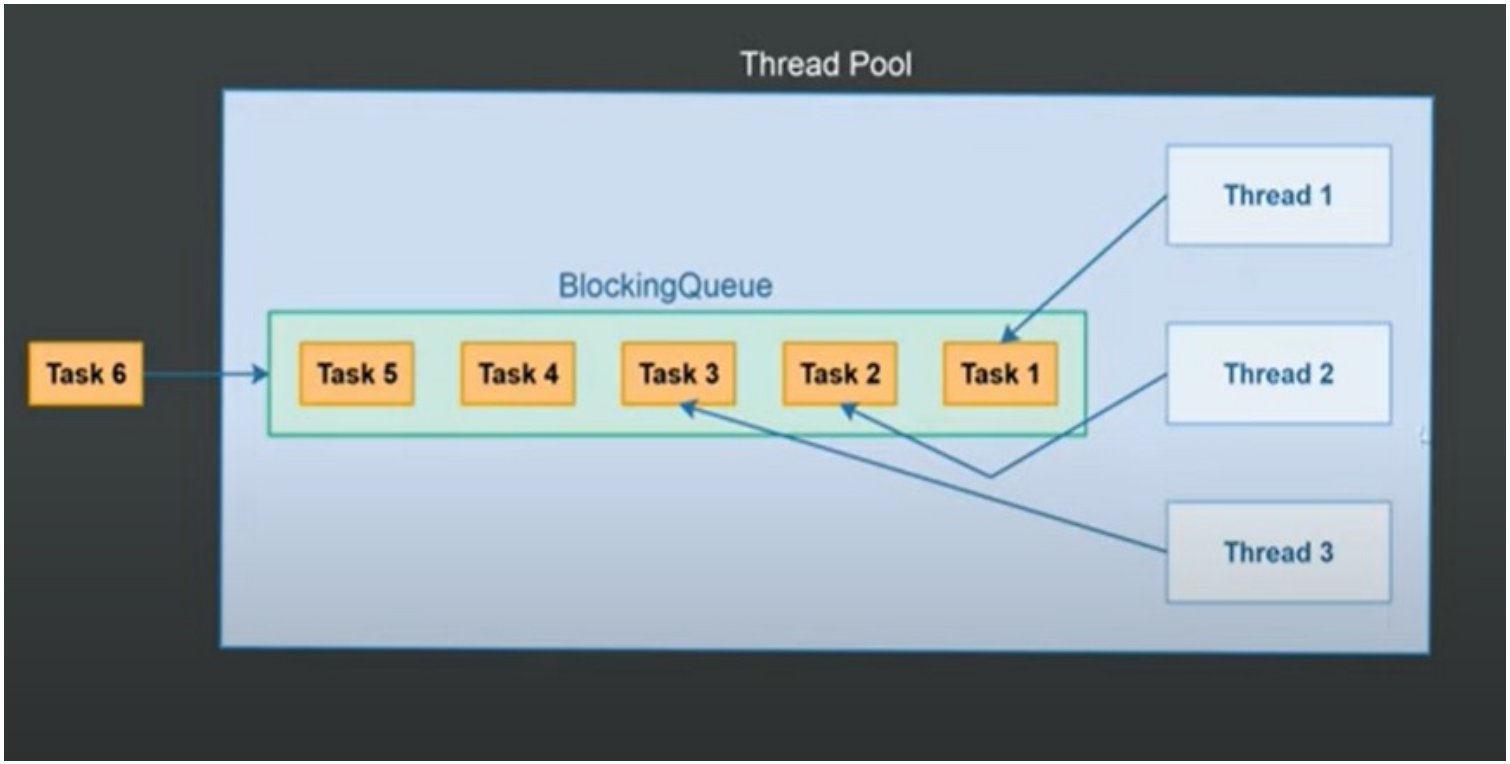


THREAD POOL: MOTIVAZIONI

scenario di riferimento: si deve eseguire un gran numero di task (ad esempio un task per ogni client, nel server):

- un thread per ogni task: può diventare improponibile, specialmente nel caso di lightweight tasks molto frequenti.
- alternativa
  - creare un pool di thread
  - ogni thread può essere usato per più task
- obiettivo:
  - diminuire il costo per l'attivazione/terminazione dei threads
  - riusare lo stesso thread per l'esecuzione di più tasks
  - eventualmente controllare il numero massimo di thread che possono essere eseguiti concorrentemente

THREAD POOLING



- in questo scenario: 3 thread attivi al massimo
- una coda interna (`BlockingQueue`) usata per memorizzare i task
- quando un thread finisce un task, automaticamente tenta di prelevarne un altro dalla coda: distribuzione automatica dei task

UN PO' DI TERMINOLOGIA

- l'utente struttura l'applicazione mediante un insieme di tasks.
- task segmento di codice che può essere eseguito da un esecutore
  - in JAVA corrisponde ad un oggetto di tipo `Runnable`
- Thread è un esecutore di tasks.
- Thread Pool
  - struttura dati la cui dimensione massima può essere prefissata, che contiene riferimenti ad un insieme di threads
  - i thread del pool possono essere riutilizzati per l'esecuzione di più tasks
  - la sottomissione di un task al pool viene disaccoppiata dall'esecuzione del thread.
  - l'esecuzione del task può essere ritardata se non vi sono risorse disponibili

THREAD POOL: CONCETTI GENERALI

- il progettista
  - crea il pool e stabilisce una politica per la gestione dei thread del pool
    - quando i thread vengono attivati: al momento della creazione del pool, on demand, all'arrivo di un nuovo task,....
    - se e quando è opportuno terminare l'esecuzione di un thread
      - se non c'è un numero sufficiente di tasks da eseguire
  - sottomette i tasks per l'esecuzione al thread pool.

- il supporto, al momento della sottomissione del task, può
  - utilizzare un thread attivato in precedenza, inattivo in quel momento
  - creare un nuovo thread
  - memorizzare il task in una struttura dati (coda), in attesa dell'esecuzione
  - respingere la richiesta di esecuzione del task

- il numero di threads attivi nel pool può variare dinamicamente

JAVA THREADPOOL: IMPLEMENTAZIONE

- fino a JAVA 4 a carico del programmatore
- JAVA 5.0 definisce la libreria java.util.concurrent che contiene metodi per
  - creare un thread pool ed il gestore associato
  - definire la struttura dati utilizzata per la memorizzazione dei tasks in attesa
  - definire specifiche politiche per la gestione del pool
- il meccanismo introdotto permette una migliore strutturazione del codice poichè tutta la gestione dei threads può essere delegata al supporto

JAVA THREADPOOL: IMPLEMENTAZIONE

- alcune interfacce definiscono servizi generici di esecuzione

```
public interface Executor  {
        public void execute (Runnable task) }
public interface ExecutorService extends Executor
                {.. }
```

- diversi servizi implementano il generico `ExecutorService` (`ThreadPoolExecutor, ScheduledThreadPoolExecutor,..`)
- la classe `Executors` opera come una Factory in grado di generare oggetti di tipo `ExecutorService` con comportamenti predefiniti.
- i tasks devono essere incapsulati in oggetti di tipo `Runnable` e passati a questi esecutori, mediante invocazione del metodo `execute()`

I TASK DA SOTTOMETTERE AL POOL

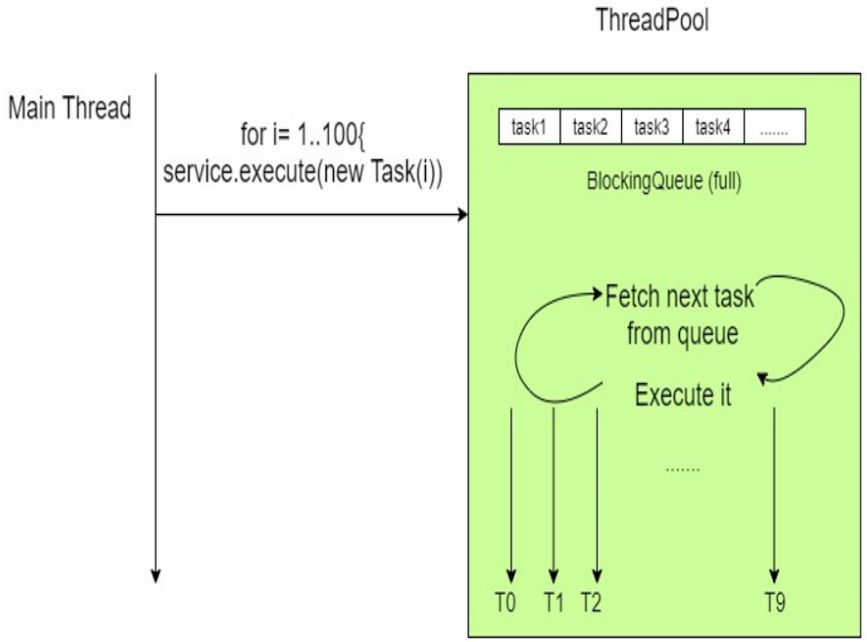
```
import java.util.*;
public class Task implements Runnable {
   Private int  name;
   public Task(int name) {this.name=name;}
   public void run() {
      try{
        Long duration=(long)(Math.random()*10);
        System.out.printf("%s: Task %s: Starting a task during %d seconds\n",
                          Thread.currentThread().getName(),name,duration);
        Thread.sleep(duration);
         }
      catch (InterruptedException e) {e.printStackTrace();}
      System.out.printf("%s: Task Finished %s \n",
                     Thread.currentThread().getName(),name);}}}
```

FIXEDTHREADPOOL

- un tipo di threadpool con comportamento predefinito
- vengono creati n thread, n fissato al momento della inizializzazione del pool, riutilizzati per l'esecuzione di più tasks
- quando viene sottomesso un task T
  - se tutti i threads sono occupati nell'esecuzione di altri tasks, T viene inserito in una coda, gestita automaticamente dall'`ExecutorService`
  - se almeno un thread è inattivo, viene utilizzato quel thread
- utilizza una `LinkedBlockingQueue`
  - coda illimitata

FIXEDTHREADPOOL

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
public class ExampleFixed{
   public static void main(String[] args) {
   // create the pool
   ExecutorService service =
      Executors.newFixedThreadPool(10);
   //submit the task for execution
   for (int i =0; i<100; i++) {
         service.execute(new Task(i)) }
    System.out.println("Thread Name:"+
        Thread.currentThread().getName()
    } }
```



la coda è una LinkedBlockingQueue

L'OUTPUT DEL PROGRAMMA

```
Thread Name:main
pool-1-thread-7: Task 6: Starting during 6 seconds
pool-1-thread-9: Task 8: Starting during 9 seconds
pool-1-thread-8: Task 7: Starting during 7 seconds
pool-1-thread-10: Task 9: Starting during 9 seconds
pool-1-thread-2: Task 1: Starting during 9 seconds
pool-1-thread-4: Task 3: Starting during 9 seconds
pool-1-thread-1: Task 0: Starting during 1 seconds
pool-1-thread-6: Task 5: Starting during 0 seconds
pool-1-thread-3: Task 2: Starting during 9 seconds
pool-1-thread-5: Task 4: Starting during 3 seconds
pool-1-thread-6: Task Finished 5
pool-1-thread-6: Task 10: Starting during 9 seconds
pool-1-thread-1: Task Finished 0
pool-1-thread-1: Task 11: Starting during 3 seconds
pool-1-thread-5: Task Finished 4
pool-1-thread-5: Task 12: Starting during 5 seconds
pool-1-thread-1: Task Finished 11
pool-1-thread-7: Task Finished 6
pool-1-thread-1: Task 13: Starting during 1 seconds
pool-1-thread-7: Task 14: Starting during 2 seconds
….....................
```

Importante:

- lo stesso thread riutilizzato per più tasks

L'OUTPUT DEL PROGRAMMA

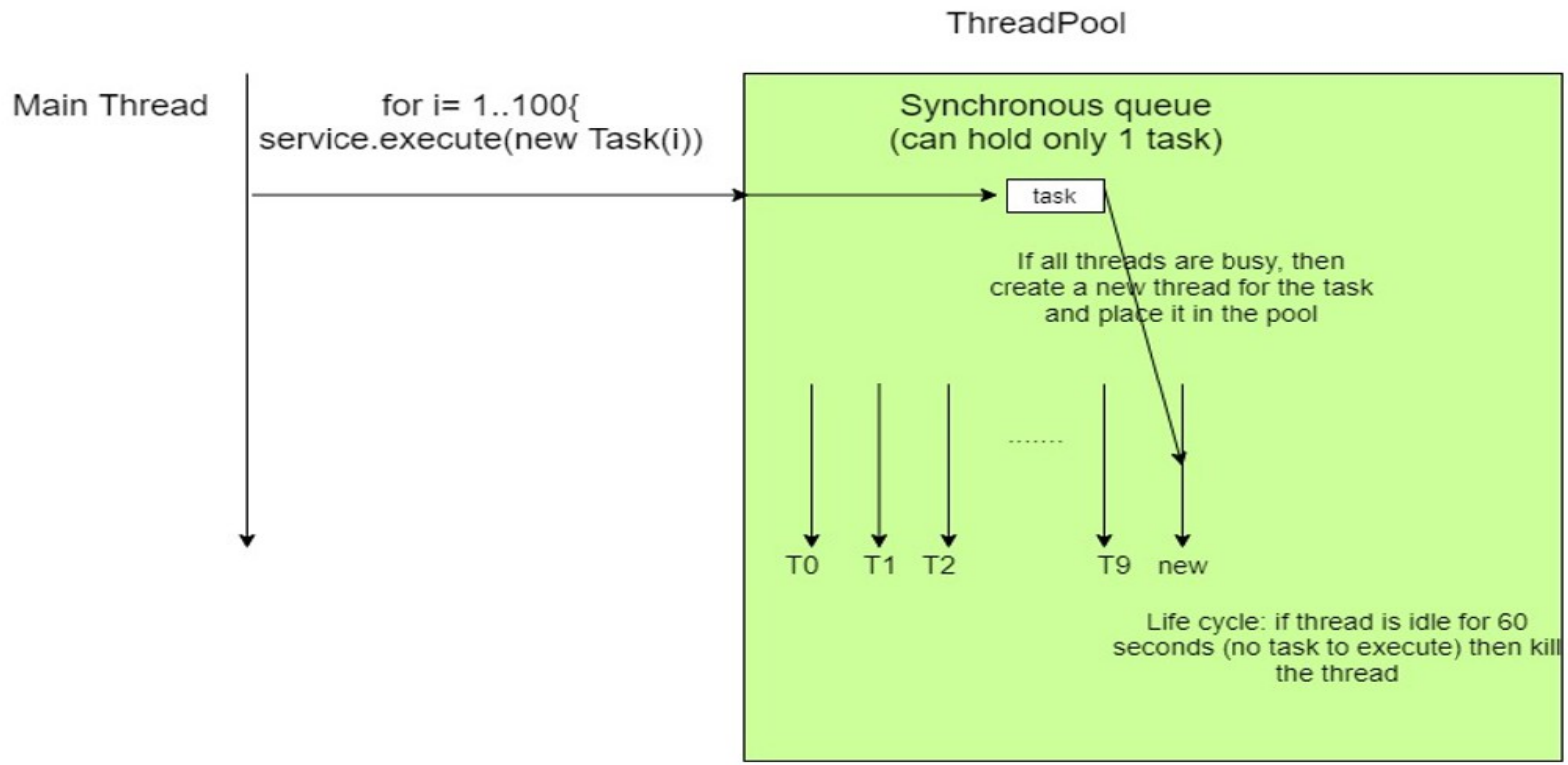
```
pool-1-thread-1: Task 0: Starting during 4 seconds
pool-1-thread-1: Task Finished 0
pool-1-thread-2: Task 1: Starting during 7 seconds
pool-1-thread-2: Task Finished 1
pool-1-thread-3: Task 2: Starting during 0 seconds
pool-1-thread-3: Task Finished 2
pool-1-thread-4: Task 3: Starting during 0 seconds
pool-1-thread-4: Task Finished 3
pool-1-thread-5: Task 4: Starting during 4 seconds
pool-1-thread-5: Task Finished 4
pool-1-thread-6: Task 5: Starting during 2 seconds
pool-1-thread-6: Task Finished 5
pool-1-thread-7: Task 6: Starting during 9 seconds
pool-1-thread-7: Task Finished 6
pool-1-thread-8: Task 7: Starting during 5 seconds
pool-1-thread-8: Task Finished 7
pool-1-thread-9: Task 8: Starting during 5 seconds
pool-1-thread-9: Task Finished 8
pool-1-thread-10: Task 9: Starting during 6 seconds
pool-1-thread-10: Task Finished 9
pool-1-thread-1: Task 10: Starting during 3 seconds
pool-1-thread-1: Task Finished 10
```

- cosa accade se si distanzia la sottomissione dei task ai thread, ad esempio inserendo una sleep, nel for dopo la execute?
- i thread sono tutti attivi e vengono utilizzati in modalità round-robin

CACHEDTHREADPOOL

- un altro tipo di threadpool con comportamento predefinito
- attivato con

```
ExecutorService service = Executors.newCachedThreadPool();
```



CACHEDTHREADPOOL

- se tutti i thread del pool sono occupati nell'esecuzione di altri task e c'è un nuovo task da eseguire, viene creato un nuovo thread.

  nessun limite alla dimensione del pool

- se disponibile, viene riutilizzato un thread che ha terminato l'esecuzione di un task precedente.
- se un thread rimane inutilizzato per 60 secondi, la sua esecuzione termina

- elasticità: *"un pool che può espandersi all'infinito, ma si contrae quando la domanda di esecuzione di task diminuisce"*

DIMINUIRE LA FREQUENZA DEI TASK

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
public class ExampleCached{
  public static void main(String[] args) {
    ExecutorService service = Executors.newCachedThreadPool();
    for (int i =0; i<100; i++) {
        service.execute(new Task(i));  sleep(1000); }
    System.out.println("ThreadName:"+Thread.currentThread().getName());
   }
  private static void sleep(long timeMillis) {
    try {
        Thread.sleep(timeMillis);
      } catch(InterruptedException e) {}}}
```

OUTPUT DEL PROGRAMMA

```
pool-1-thread-11: Task 10: Starting during 5 seconds
pool-1-thread-100: Task 99: Starting during 5 seconds
Thread Name:main
pool-1-thread-99: Task 98: Starting during 7 seconds
pool-1-thread-98: Task 97: Starting during 7 seconds
pool-1-thread-97: Task 96: Starting during 6 seconds
pool-1-thread-96: Task 95: Starting during 6 seconds
pool-1-thread-95: Task 94: Starting during 9 seconds
pool-1-thread-94: Task 93: Starting during 2 seconds
pool-1-thread-93: Task 92: Starting during 3 seconds
pool-1-thread-92: Task 91: Starting during 0 seconds
pool-1-thread-92: Task Finished 91
pool-1-thread-91: Task 90: Starting during 8 seconds
pool-1-thread-90: Task 89: Starting during 6 seconds
pool-1-thread-89: Task 88: Starting during 6 seconds
pool-1-thread-88: Task 87: Starting during 1 seconds
pool-1-thread-87: Task 86: Starting during 3 seconds
pool-1-thread-86: Task 85: Starting during 7 seconds
pool-1-thread-85: Task 84: Starting during 7 seconds
pool-1-thread-84: Task 83: Starting during 7 seconds
pool-1-thread-83: Task 82: Starting during 4 seconds
pool-1-thread-82: Task 81: Starting during 8 seconds
```

attivato un nuovo therad per ogni nuovo task

OUTPUT DEL PROGRAMMA

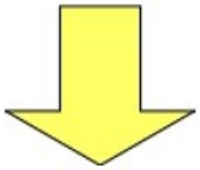
```
pool-1-thread-1: Task 0: Starting during 3 seconds
pool-1-thread-1: Task Finished 0
pool-1-thread-1: Task 1: Starting during 7 seconds
pool-1-thread-1: Task Finished 1
pool-1-thread-1: Task 2: Starting during 0 seconds
pool-1-thread-1: Task Finished 2
pool-1-thread-1: Task 3: Starting during 3 seconds
pool-1-thread-1: Task Finished 3
pool-1-thread-1: Task 4: Starting during 5 seconds
pool-1-thread-1: Task Finished 4
pool-1-thread-1: Task 5: Starting during 5 seconds
pool-1-thread-1: Task Finished 5
pool-1-thread-1: Task 6: Starting during 9 seconds
pool-1-thread-1: Task Finished 6
pool-1-thread-1: Task 7: Starting during 6 seconds
pool-1-thread-1: Task Finished 7
pool-1-thread-1: Task 8: Starting during 1 seconds
pool-1-thread-1: Task Finished 8
pool-1-thread-1: Task 9: Starting during 1 seconds
pool-1-thread-1: Task Finished 9
pool-1-thread-1: Task 10: Starting during 0 seconds
…….....
```

- cosa accade se distanzio la sottomissione dei task ai thread, ad esempio inserendo una sleep, nel for, dopo la execute?

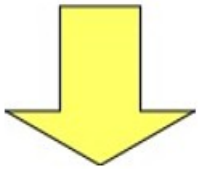
  - ora viene utilizzato sempre il thread-1 per tutti i task

LA CLASSE THREAD POOL EXECUTOR

```
ExecutorService service = Executors.newFixedThreadPool(10)
```



```
public static ExecutorService newFixedThreadPool(int nThreads) {

        return new ThreadPoolExecutor(nThreads, nThreads, 0L,

            TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());}
```



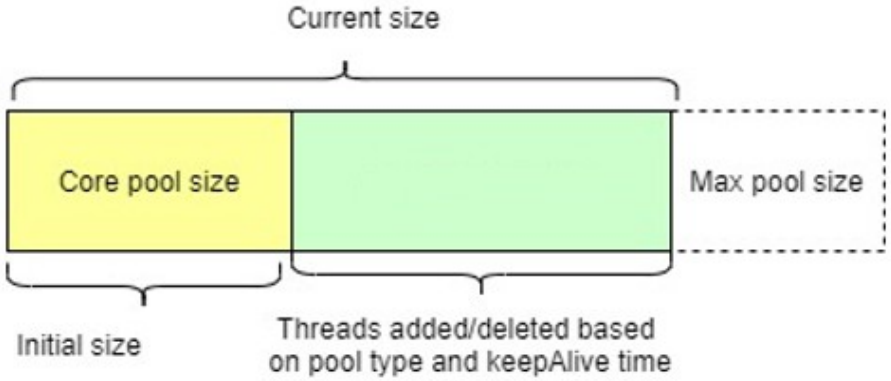
```
public ThreadPoolExecutor (int CorePoolSize,
                           int MaximumPoolSize,
                           long keepAliveTime,
                           TimeUnit unit,
                           BlockingQueue <Runnable> workqueue)......}
```

LA CLASSE THREAD POOL EXECUTOR

```
import java.util.concurrent.*;
public class ThreadPoolExecutor implements ExecutorService
        {public ThreadPoolExecutor
                  (int CorePoolSize,
                   int MaximumPoolSize,
                   long keepAliveTime,
                   TimeUnit unit,
                   BlockingQueue <Runnable> workqueue
                   RejectedExecutionHandler handler)}
```

- il costruttore più generale: personalizzazione della politica di gestione del pool
- CorePoolSize, MaximumPoolSize, keepAliveTime controllano la gestione dei thread del pool
- workqueue è una struttura dati necessaria per memorizzare gli eventuali tasks in attesa di esecuzione

THREAD POOL EXECUTOR



- core: nucleo minimo di thread attivi nel pool
- i thread del core possono essere attivati
  - al momento della creazione del pool: `PrestartAllCoreThreads( )`
  - "on demand" , al momento della sottomissione di un nuovo task, anche se qualche thread già creato del core è inattivo.

    obiettivo: riempire il pool prima possibile.
- quando tutti i threads sono stati creati, la politica cambia

THREADPOOL: ELASTICITA'

Keep Alive Time: per i thread non appartenenti al core

- si considera il timeout T specificato al momento della costruzione del ThreadPool mediante la definizione di
  - un valore (es: 50000)
  - l'unità di misura utilizzata (es: `TimeUnit. MILLISECONDS`)
- se nessun task viene sottomesso entro T, il thread termina la sua esecuzione, riducendo così il numero di threads del pool
- la dimensione del ThreadPool non scende mai sotto Core pool size
  - unica eccezione: *allowCoreThreadTimeOut(boolean value)* invocato con il parametro settato a *true*

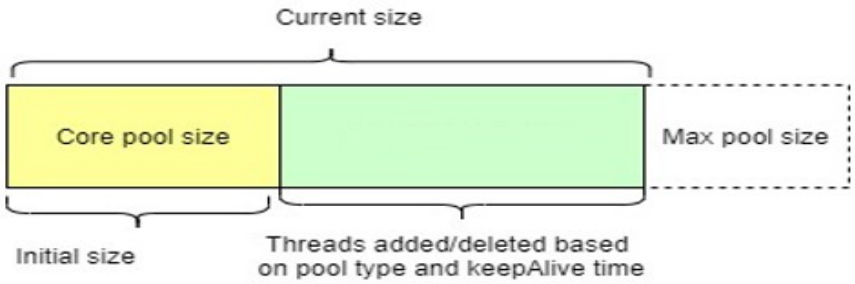
THREAD POOL EXECUTOR: RIASSUNTO

se tutti i thread del core sono già stati creati e viene sottomesso un nuovo task:

- se un thread del core è inattivo, il task viene assegnato ad esso
  - se tutti i thread del core stanno eseguendo un task e la coda non è piena, e , il nuovo task viene inserito nella coda: i task verranno quindi poi prelevati dalla coda ed inviati ai thread disponibili
- se tutti i thread del core stanno eseguendo un task e la coda è piena
  - si crea un nuovo thread attivando così k thread,

$$corePoolSize \leq k \leq MaxPoolSize$$

- se coda piena e sono attivi MaxPoolSize threads
  - il task viene respinto



THREAD POOL EXECUTOR: PARAMETRI

Parameter	Type	Meaning
corePoolSize	int	Minimum/Base size of the pool
maxPoolSize	int	Maximum size of the pool
keepAliveTime + unit	long	Time to keep an idle thread alive (after which it is killed)
workQueue	BlockingQueue	Queue to store the tasks from which threads fetch them
handler	RejectedExecutionHandler	Callback to use when tasks submitted are rejected

ISTANZE DI THREADPOOLEXECUTOR

PARAMETER	FIXEDTHREADPOOL	CACHEDTHREADPOOL
CorePoolSize	Valore passato nel costruttore	0
MaxPoolSize	stesso valore di CorePoolSize	Integer.MAXVALUE
KeepAlive	0 Secondi	60 secondi

```
public static ExecutorService newFixedThreadPool(int nThreads) {
   return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS,....)
public static ExecutorService newCachedThreadPool() {
      return new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L,TimeUnit.SECONDS,....);
```

- KeepAlive= 0 secondi corrisponde a "KeepAlive non significativo", il thread non viene mai disattivato

ISTANZE DI THREADPOOLEXECUTOR

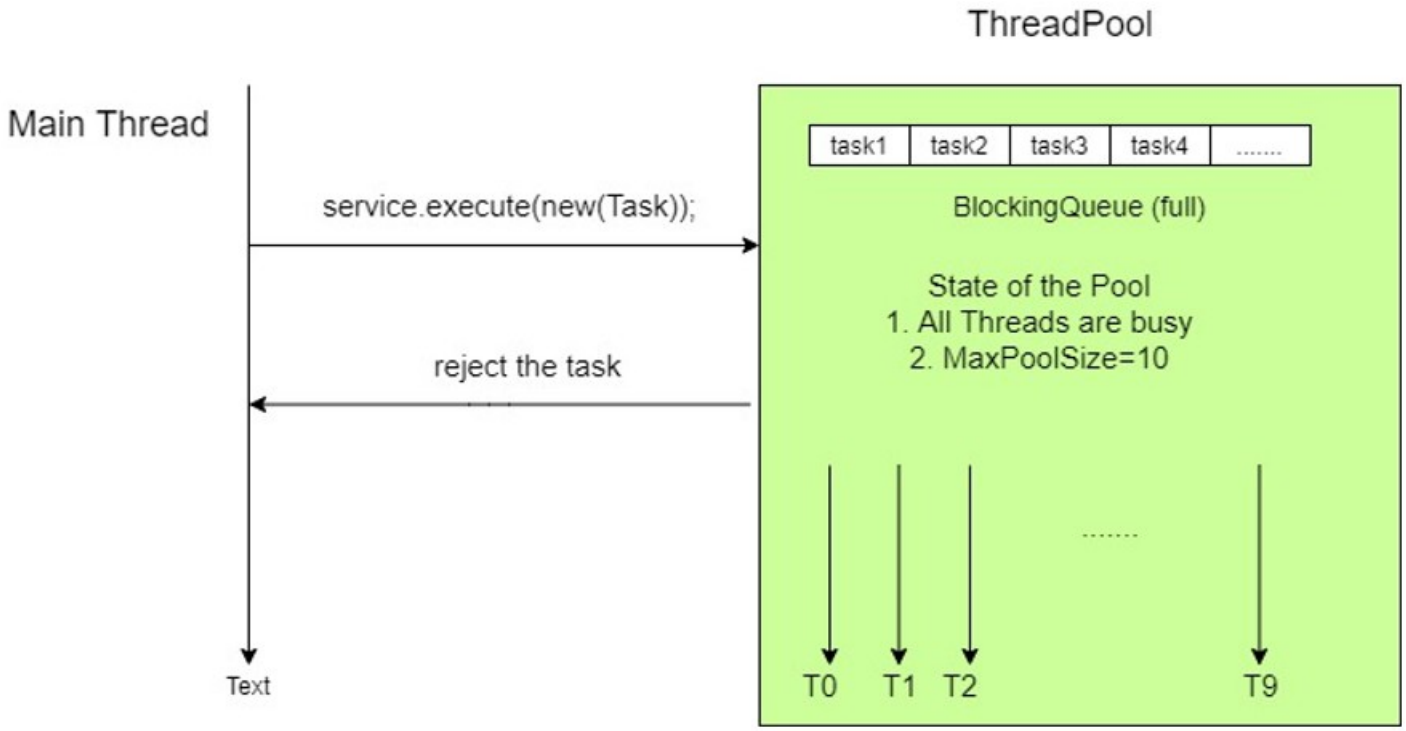
POOL	QUEUE TYPE	WHY?
FixedThreadPool	LinkedBlockingQueue	Threads are limited, thus unbounded queue to store tasks Note: since queue can never become full, new threads are never created
CachedThreadPool	SynchronousQueue	Threads are unbounded, thus no need to store the tasks. Create the new thread and give it directly the task
Custom (ThreadPoolExecutor)	ArrayBlockingQueue	Bounded queue to store the tasks. If queue gets full, a new task is created (as long as count is less than MaxPoolSize)

```
public static ExecutorService newFixedThreadPool(int nThreads) {
   return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS,....)

public static ExecutorService newCachedThreadPool() {
       return new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L,TimeUnit.SECONDS,....);
```

ALTRI TIPI DI THREAD POOL

- `Single Threaded Executor`
  - un singolo thread
  - equivalente ad invocare un `FixedThreadPool` di dimensione 1
  - utilizzo: assicurare che i thread del pool vengano eseguiti nell'ordine con cui si trovano in coda (sequenzialmente)
  - `SingleThreadExecutor`
- `Scheduled Thread Pool`
  - distanziare esecuzione dei task con un certo delay
  - task periodici

THREADPOOL: REJECTION



THREADPOOL: REJECTION HANDLER

come viene gestito il rifiuto di un task? E' possibile

- scegliere esplicitamente una "rejection policy" al momento della creazione del task
  - `AbortPolicy` : politica di default, consiste nel sollevare `RejectedExecutionException`
  - `DiscardPolicy`,`DiscardOldestPolicy`, `CallerRunsPolicy`: altre politiche predefinite (vedere API):
- definire un custom rejection handler implementando l'interfaccia `RejectExecutionHandler` ed il metodo `rejectedExecution`

THREADPOOL: REJECTION HANDLER

```
import java.util.concurrent.*;
public class RejectedException {
public static void main (String[] args )
    {ExecutorService service
          = new ThreadPoolExecutor(10, 12, 120, TimeUnit.SECONDS,
                                   new ArrayBlockingQueue<Runnable>(3));
    for (int i=0; i<20; i++)
         try {
              service.execute(new Task(i));
         } catch (RejectedExecutionException e)
            {System.out.println("task rejected"+e.getMessage());}
         }}
```

EXECUTOR LIFECYCLE

- la JVM termina la sua esecuzione quando tutti i thread (non demoni) terminano la loro esecuzione
- è necessario analizzare il concetto di terminazione, nel caso di

  `Executor Service` poichè

  - i tasks vengono eseguito in modo asincrono rispetto alla loro sottomissione.
  - in un certo istante, alcuni task sottomessi precedentemente possono essere completati, alcuni in esecuzione, alcuni in coda.
- poichè alcuni threads possono essere sempre attivi, JAVA mette a disposizione dell'utente alcuni metodi che permettono di terminare l'esecuzione del pool

EXECUTORS: TERMINAZIONE GRADUALE

- la terminazione può avvenire
  - in modo graduale: "finisci ciò che hai iniziato, ma non iniziare nuovi tasks"
  - in modo istantaneo. "stacca la spina immediatamente"

- `shutdown()` "terminazione graduale": inizia la terminazione
  - nessun task viene accettato dopo che è stata invocata.
  - tutti i tasks sottomessi in precedenza e non ancora terminati vengono eseguiti, compresi quelli accodati, la cui esecuzione non è ancora iniziata

```
service.shutdown();
// throw RejectionExecutionException on a new task submission
service.isShutdown();
// return true is shutdown has begun
service.isTerminated();
// return true if all tasks are completed, including queued ones
service. awaitTermination(long timeout, TimeUnit unit)
// block until all tasks are completed or if timeout occurs
```

EXECUTORS: TERMINAZIONE IMMEDIATA

```
List <Runnable> runnables = service.shutdownNow();
```

- non accetta ulteriori tasks ed elimina i tasks non ancora iniziati
  - restituisce una lista dei tasks che sono stati eliminati dalla coda
- implementazione best effort: tenta di terminare l'esecuzione dei thread che stanno eseguendo i tasks, inviando una interruzione ai thread in esecuzione nel pool
  - non garantisce la terminazione immediata dei threads del pool
  - se un thread non risponde all'interruzione non termina
- se sottometto il seguente task al pool

```
public class ThreadLoop implements Runnable {
        public ThreadLoop(){};
        public void run( ){while (true)  { }  }  }
```

e poi invoco la shutdownNow( ), osservate che il programma non termina

DETERMINARE LA DIMENSIONE DEL THREADPOOL

- la dimensione ideale per il numero di threads in un ThreadPool non è facile da determinare
- dato il numero di core della macchina, dipende dal tipo di task da eseguire
- CPU bound tasks
  - task che devono eseguire calcoli complessi. Un esempio: inversione parziale di un hash, come le PoW di Bitcoin ed Ethereum
  - in questo scenario, idealmente, la dimensione ottimale del pool = numero di CPU cores
- IO bound tasks
  - accesso a database, accesso alla rete
  - spesso bloccati in attesa del completamento di operazioni del SO
  - un numero di thread maggiore del numero di CPU cores può aumentare le performance della applicazione

DETERMINARE LA DIMENSIONE DEL THREADPOOL

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class CPUIntensiveTask implements Runnable {
                    public void run() {
                      // eseguo la PoW }
                    }
public class ThreadDimensioning {
    public static void main (String [] args) {
    // get count of available cores
    int coreCount = Runtime.getRuntime().availableProcessors();
    System.out.println(coreCount);
    ExecutorService service = Executors.newFixedThreadPool(coreCount);
    // submit the tasks for execution
    for (int i=0; i< 100; i++) {
       service.execute(new CPUIntensiveTask());
      } }
```

DETERMINARE LA DIMENSIONE DEL THREADPOOL

TIPO DI TASK	DIMENSIONE IDEALE POOL	CONSIDERAZIONI
CPU Intensive	CPU Core Count	Quante altre applicazioni sono in esecuzione sulla stessa CPU
IO Intensive	High	Numero esatto dipende anche dalla frequenza con cui i task vengono sottomessi e dal tempo medio di attesa. Troppi thread possono aumentare la memory pressure

THREAD CHE RESTITUISCONO RISULTATI

- un oggetto di tipo `Runnable`
  - incapsula un'attività che viene eseguita in modo asincrono
  - la `Runnable` è un metodo asincrono, senza parametri e che non restituisce un valore di ritorno
- per definire un task che restituisca un valore di ritorno
  - `Interface Callable`: per definire un task che può restituire un risultato e sollevare eccezioni
  - `Future`: per rappresentare il risultato di una computazione asincrona. e definisce metodi
    - per controllare se la computazione è terminata
    - per attendere la terminazione di una computazione (eventualmente per un tempo limitato)
    - per cancellare una computazione, ...
- la classe `FutureTask` fornisce una implementazione della interfaccia Future.

L'INTERFACCIA CALLABLE

```
public interface Callable <V>
    { V call() throws Exception;}
```

- contiene il solo metodo `call()`, analogo al metodo `run()` dell'interfaccia `Runnable`
- il codice del task è implementato nel metodo `call()`
- a differenza del metodo `run()`, il metodo `call()` può
  - restituire un valore
  - sollevare eccezioni
- il parametro di tipo <V> indica il tipo del valore restituito
  - `Callable <Integer>` rappresenta una elaborazione asincrona che restituisce un valore di tipo Integer

THREAD CHE RESTITUISCONO RISULTATI

Definire un task T che calcoli una approssimazione di π, mediante la serie di Gregory-Leibniz (vedi lezione precedente). T restituisce il valore calcolato quando la differenza tra l'approssimazione ottenuta ed il valore di Math.PI risulta inferiore ad una soglia precision. T deve essere eseguito in un thread.

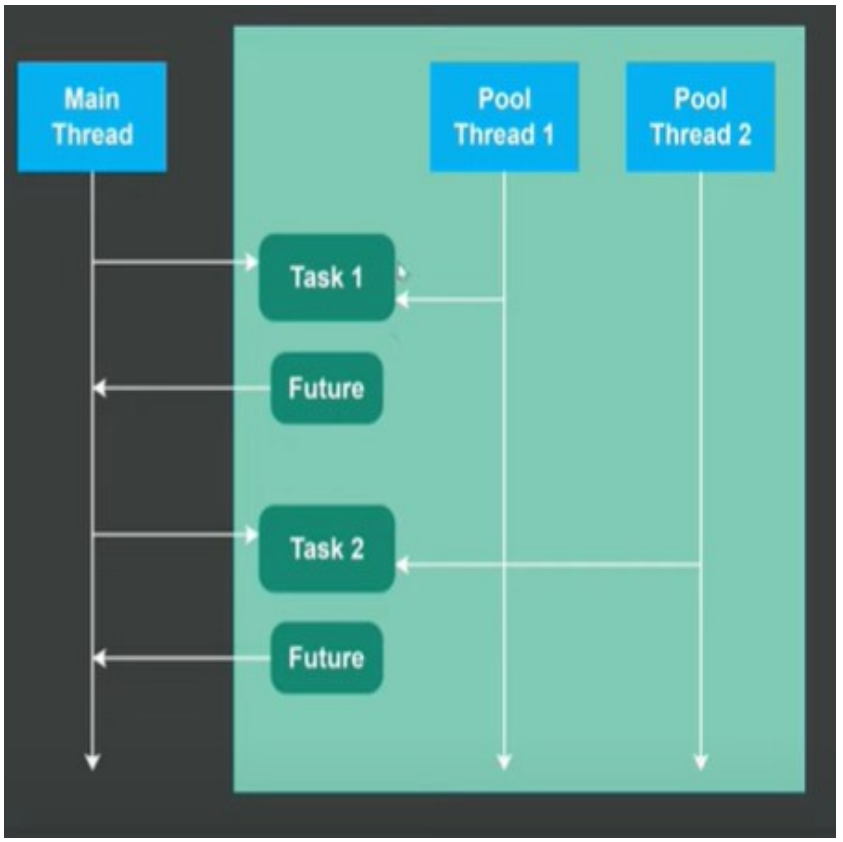
```
import java.util.concurrent.*;
public class pigreco implements Callable <Double>
     {private Double precision;
      public pigreco (Double precision)
           {this.precision=precision;};
      public Double call( )
                {Double result = <calcolo approssimazione di π>
                 return result }  }
```

THREAD CHE RESTITUISCONO RISULTATI

```
import java.util.*;
import java.util.concurrent.*;
public class futurepools {
public static void main(String args[])
    ExecutorService pool = Executors.newCachedThreadPool();
    double precision = ...;
    pigreco pg = new pigreco(precision);
    Future <Double> result = pool.submit(pg);
    try{double ris = result.get(1000L,TimeUnit.MILLISECONDS);
       System.out.println(ris+"valore di pigreco");}
    catch(...){       }
     ...}}
```

L'INTERFACCIA FUTURE

- sottomette direttamente l'oggetto di tipo `Callable` al pool mediante il metodo submit
- la sottomissione restituisce un oggetto di tipo `Future`
- ogni oggetto `Future` è associato ad uno dei task sottomessi al ThreadPool
- è possibile applicare diversi metodi all'oggetto `Future` (vedi slide successiva)



L'INTERFACCIA FUTURE

```
public interface Future <V>
    { V get( ) throws...;
      V get (long timeout, TimeUnit) throws...;
      void cancel (boolean mayInterrupt);
      boolean isCancelled(  );
      boolean isDone(  ); }
```

- metodo get()
  - si blocca fino a che il thread non ha prodotto il valore richiesto e restituisce il valore calcolato
- metodo `get (long timeout, TimeUnit)`
  - definisce un tempo massimo di attesa della terminazione del task, dopo cui viene sollevata una `TimeoutException`
- è possibile cancellare il task e verificare se la computazione è terminata oppure è stata cancellata

CONDIVIDERE RISORSE TRA THREADS

- scenario tipico di un programma concorrente: un insieme di thread condividono una risorsa.

  più thread accedono concorrentemente allo stesso file, alla stessa parte di un database o di una struttura di memoria

- l'accesso non controllato a risorse condivise può provocare situazioni di errore ed inconsistenze.

  race conditions

- sezione critica: blocco di codice in cui si effettua l'accesso ad una risorsa condivisa e che deve essere eseguito da un thread per volta

- meccanismi di sincronizzazione per l'implementazione di sezioni critiche
  - interfaccia `Lock` e le sue diverse implementazioni
  - concetto di `monitor`

UN ESEMPIO DI RACE CONDITION

- si consideri un conto bancario e due thread che vi accedono in modo concorrente
  - il thread Company versa denaro sul conto corrente
  - il thread BancoMat preleva denaro dal conto corrente
- mostreremo come si possa verificare una race condition, nel caso in cui l'accesso al conto sia incontrollato
- invariante: lo stesso numero di versamenti e prelievi dello stesso valore dovrebbe lasciare invariato l'ammontare inizialmente presente sul conto corrente

UN ESEMPIO DI RACE CONDITION

```
public class Account {
    private double balance;
    public  double getBalance() { return balance; }
    public  void   setBalance(double balance)
        { this.balance = balance;}
    public void addAmount(double amount) {
        double tmp=balance;
        try
                { Thread.sleep(10);}
        catch (InterruptedException e) { e.printStackTrace();}
        tmp=tmp+amount;
        balance=tmp;
     }
```

UN ESEMPIO DI RACE CONDITION

```
public void subtractAmount(double amount) {
      double tmp=balance;
      try {
              Thread.sleep(10);
          } catch (InterruptedException e){e.printStackTrace(); }
     tmp=tmp-amount;
     balance=tmp;   } }
```

- un oggetto istanza della classe `Account` rappresenta un oggetto condiviso tra thread che effettuano versamenti e altri che effettuano prelievi
- l'accesso non sincronizzato alla risorsa condivisa può generare situazioni di inconsistenza.

UN ESEMPIO DI RACE CONDITIONS

```
public class Bancomat implements Runnable {
  private Account account;
  public Bancomat(Account account)
      {
      this.account=account;
      }
  public void run() {
        for (int i=0; i<100; i++)
              {
              account.subtractAmount(1000);
              }
                    }
                                        }
```

UN ESEMPIO DI RACE CONDITION

```
public class Company implements Runnable {
    private Account account;
    public Company(Account account) {
           this.account=account;
                                    }
    public void run() {
          for (int i=0; i<100; i++){
                 account.addAmount(1000);
                                   }}}
```

- un riferimento all'oggetto condiviso Account viene passato esplicitamente ai thread Company e Bancomat
- tutti i thread mantengono un riferimento alla struttura dati condivisa

UN ESEMPIO DI RACE CONDITION

```
public class Main {
public static void main(String[] args) {
   Account account=new Account();
   account.setBalance(1000);
   Company company=new Company(account);
   Thread companyThread=new Thread(company);
   Bancomat bank=new Bancomat(account);
   Thread bankThread=new Thread(bank);
   System.out.printf("Initial Balance:%f\n",account.getBalance());
   companyThread.start();
   bankThread.start();
   try { companyThread.join();
         bankThread.join();
         System.out.printf("Final Balance:%f\n",account.getBalance());
       } catch (InterruptedException e) {e.printStackTrace();}}}
```

UN ESEMPIO DI RACE CONDITION

- output di alcune esecuzioni del programma:

```
Account : Initial Balance: 1000,000000
Account : Final Balance:   17000,000000
Account : Initial Balance: 1000,000000
Account : Final Balance:   89000,000000
                  …...
```

- se avviene una commutazione di contesto prima che l'esecuzione di uno dei metodi di Account termini, lo stato della risorsa può risultare inconsistente

  race condition, codice non rientrante

- non necessariamente l' inconsistenza si presenta ad ogni esecuzione e, se si presenta, non vengono prodotti sempre i medesimi risultati
  - non determinismo
  - comportamento dipendente dal tempo

UN ESEMPIO DI RACE CONDITION

- un thread invoca i metodi `addAmount` o `subtractAmount` e viene deschedulato prima di avere completato l'esecuzione del metodo
  - la risorsa viene lasciata in uno stato inconsistente
  - un esempio:
    - Thread Bancomat esegue subtractAccount: tmp=1000, poi viene deschedulato prima di completare il metodo
    - Thread Company completa il metodo addAccount, balance=2000
    - ritorna in esecuzione primo thread: balance=0
- Classe Thread Safe: l'esecuzione concorrente dei metodi definiti nella classe non provoca comportamenti scorretti, ad esempio race conditions
  - `Account` non è una classe thread safe !
  - per renderla thread safe: garantire che le istruzioni contenute all'interno dei metodi addAmount e subtractAmount vengano eseguite in modo atomico / indivisibile / in mutua esclusione

RACE CONDITION IN OPERAZIONI "PSEUDO ATOMICHE'

```
public class Counter {
    private int count = 0;
    public void increment()
            {++count; }
    public int getCount()
            {return count; }
}
public class CountingThread extends Thread {
  Counter c;
  public CountingThread (Counter c)
       {this.c=c;}
  public void run() {
     for(int x = 0; x < 10000; ++x)
         c.increment();
         } }
```

RACE CONDITION IN OPERAZIONI "PSEUDO ATOMICHE

```
public class Main  {
    public static void main (String args[])
    {
      final Counter counter = new Counter();
      CountingThread t1 = new CountingThread(counter);
      CountingThread t2 = new CountingThread(counter);
      t1.start(); t2.start();
      try
         { t1.join(); t2.join();
          } catch (InterruptedException e){};
      System.out.println(counter.getCount());
    }
  }
```

OPERAZIONI "PSEUDO ATOMICHE"

- 2 threads, ognuno invoca 10,000 volte il metodo `increment()`: valore finale di counter dovrebbe essere 20,000. Notate, invece, il risultato di 3 esecuzioni distinte del programma

  12349

  12639

  12170

- read-modify-write pattern: JAVA bytecodes generati per l'istruzione ++count

  getfield #2

  iconst_1

  iadd

  putfield #2

- valore di count= 42, entrambi i threads lo leggono, quindi entrambi memorizzano il valore modificato: un aggiornamento viene perduto

ASSIGNMENT 2: SIMULAZIONE UFFICIO POSTALE

- Simulare il flusso di clienti in un ufficio postale che ha 4 sportelli. Nell'ufficio esiste:
  - un'ampia sala d'attesa in cui ogni persona può entrare liberamente. Quando entra, ogni persona prende il numero dalla numeratrice e aspetta il proprio turno in questa sala.
  - una seconda sala, meno ampia, posta davanti agli sportelli, in cui si può entrare solo a gruppi di k persone

- Una persona si mette quindi prima in coda nella prima sala, poi passa nella seconda sala.

- Ogni persona impiega un tempo differente per la propria operazione allo sportello. Una volta terminata l'operazione, la persona esce dall'ufficio

ASSIGNMENT 2: SIMULAZIONE UFFICIO POSTALE

- Scrivere un programma in cui:
  - l'ufficio viene modellato come una classe JAVA, in cui viene attivato un `ThreadPool` di dimensione uguale al numero degli sportelli
  - la coda delle persone presenti nella sala d'attesa è gestita esplicitamente dal programma
  - la seconda coda (davanti agli sportelli) è quella gestita implicitamente dal `ThreadPool`
  - ogni persona viene modellata come un task, un task che deve essere assegnato ad uno dei thread associati agli sportelli
  - si preveda di far entrare tutte le persone nell'ufficio postale, all'inizio del programma

- Facoltativo: prevedere il caso di un flusso continuo di clienti e la possibilità che l'operatore chiuda lo sportello stesso dopo che in un certo intervallo di tempo non si presentano clienti al suo sportello.