
Laboratorio di Reti – B
Lezione 2
**BlockingQueues, Thread pools, Callable,
Thread synchronization: introduction**

21/09/2021

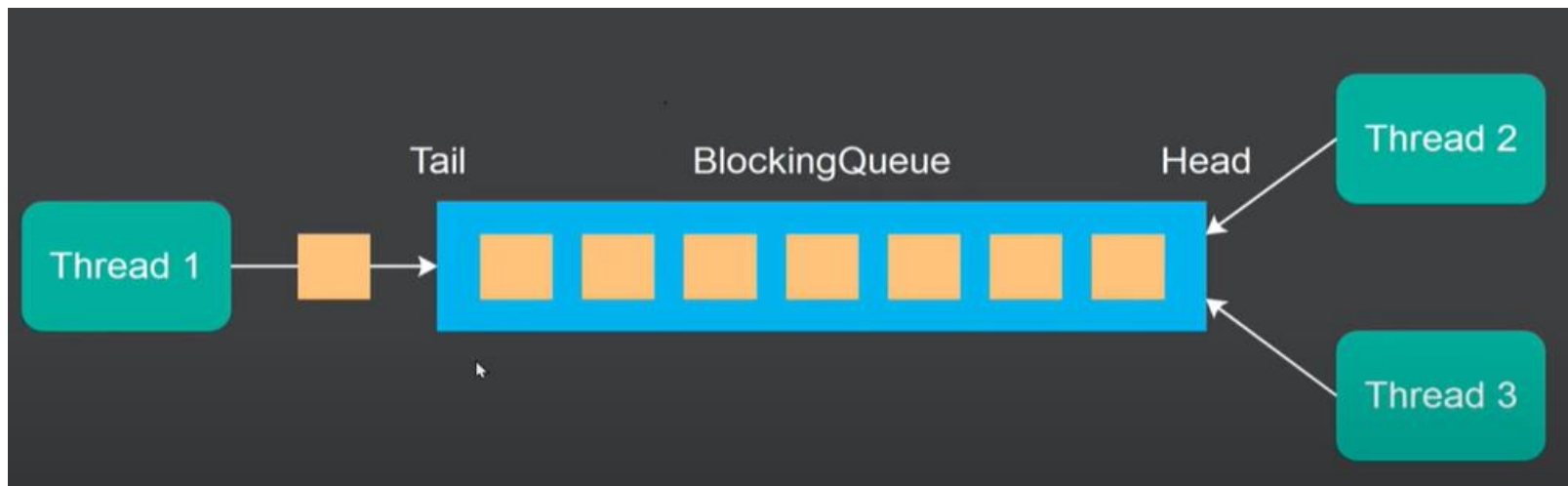
Federica Paganelli

Oggi vedremo

- BlockingQueues
- Threadpools
- Introduzione a: Accesso a risorse condivise e sincronizzazione

BlockingQueues

- Una Queue che supporta inoltre le operazioni che:
 - attendono che la coda diventi non vuota quando si recupera un elemento
 - attendono che lo spazio diventi disponibile nella coda durante la memorizzazione di un elemento
- Le implementazioni di BlockingQueue sono thread-safe. Tutti i metodi di accodamento ottengono i loro effetti in modo atomico utilizzando blocchi interni o altre forme di controllo della concorrenza.



Blockingqueue methods and implementations

4 categorie di metodi differenti, 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	<code>add(o)</code>	<code>offer(o)</code>	<code>put(o)</code>	<code>offer(o, timeout, timeunit)</code>
Remove	<code>remove(o)</code>	<code>poll()</code>	<code>take()</code>	<code>poll(timeout, timeunit)</code>
Examine	<code>element()</code>	<code>peek()</code>		

- *add, remove, element* operations **lanciano un'eccezione** se si tenta di aggiungere un elemento ad una coda piena o rimuovere un elemento dalla coda vuota.


N.B. In un programma multi-threaded la coda può diventare piena o vuota in qualsiasi momento:

- *offer, poll, and peek*: ritornano rispettivamente **false, null, null** se non possono portare a termine l'operazione.
- *put, take*: **comportamento bloccante** (se non si può portare a termine l'operazione)

Uso tipico 1/2

```
class Producer implements Runnable {  
    private final BlockingQueue queue;  
    Producer(BlockingQueue q) { queue = q; }  
    public void run() {  
        try {  
            while (true) { queue.put(produce()); }   
        } catch (InterruptedException ex) { ... handle ... }  
    }  
    Object produce() { ... }  
}
```

**Uso tipico: scenario
produttore-consumatore**

```
class Consumer implements Runnable {  
    private final BlockingQueue queue;  
    Consumer(BlockingQueue q) { queue = q; }  
    public void run() {  
        try {  
            while (true) { consume(queue.take()); }   
        } catch (InterruptedException ex) { ... handle ... }  
    }  
    void consume(Object x) { ... }  
}
```

Uso tipico 2/2

```
class Setup {  
    void main() {  
        BlockingQueue q = new SomeQueueImplementation();  
        Producer p = new Producer(q);  
        Consumer c1 = new Consumer(q);  
        Consumer c2 = new Consumer(q);  
        new Thread(p).start();  
        new Thread(c1).start();  
        new Thread(c2).start();  
    }  
}
```

Blockingqueue implementations

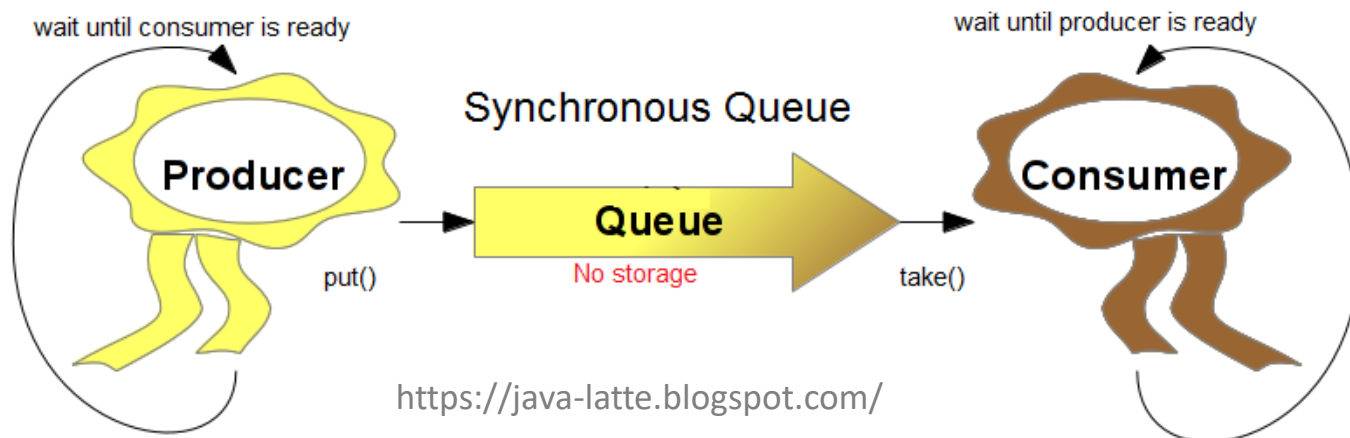
BlockingQueue è un'interfaccia, varie implementazioni disponibili

Noi useremo maggiormente:

- `ArrayBlockingQueue`
- `LinkedBlockingQueue`
- `SynchronousQueue`

Blockingqueue implementations

- **ArrayBlockingQueue** una coda di dimensione limitata, che memorizza gli elementi all'interno di un array. Upper bound definito a tempo di inizializzazione.
- **LinkedBlockingQueue** mantiene gli elementi in una struttura linkata che può avere un upper bound, oppure, se non si specifica un upper bound, l'upper bound è `Integer.MAX_VALUE`.
- **SynchronousQueue** non possiede capacità interna. Operazione di inserzione deve attendere per una corrispondente rimozione e viceversa (un po' fuorviante chiamarla coda).



Code thread safe: BlockingQueue

```
import java.util.concurrent.*;

public class BlockingQueueExample {
    public static void main(String[] args) throws Exception
    {
        BlockingQueue queue = new ArrayBlockingQueue(1024);
        Producer producer = new Producer(queue);
        Consumer consumer = new Consumer(queue);
        producer.start();
        consumer.start();
        Thread.sleep(4000);
    }
}
```

Code thread safe: BlockingQueue

```
import java.util.concurrent.*;

public class Producer extends Thread{
    protected BlockingQueue queue = null;
    public Producer(BlockingQueue queue) {
        this.queue = queue; }
    public void run() {
        try {
            queue.put("1");
            Thread.sleep(1000);
            queue.put("2");
            Thread.sleep(1000);
            queue.put("3");
        } catch (InterruptedException e) {
            e.printStackTrace(); } } }
```

Code thread safe: BlockingQueue

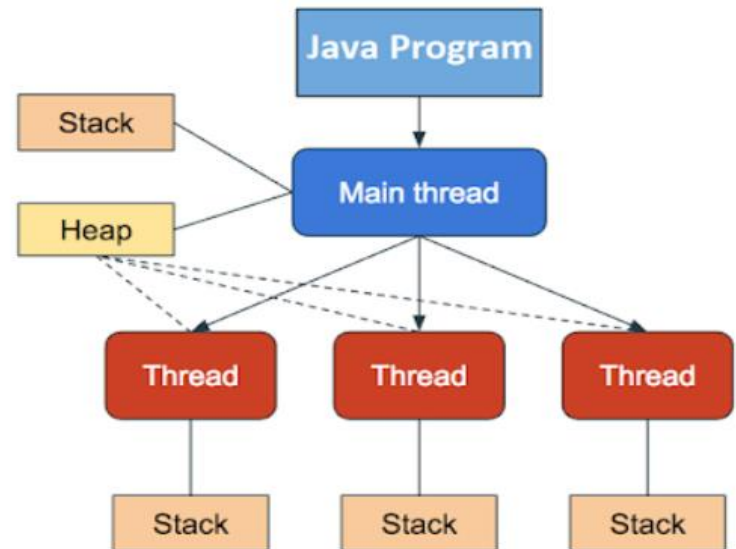
```
import java.util.concurrent.*;

public class Consumer extends Thread {
    protected BlockingQueue queue = null;
    public Consumer(BlockingQueue queue) {
        this.queue = queue; }
    public void run() {
        try {
            System.out.println(queue.take());
            System.out.println(queue.take());
            System.out.println(queue.take());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Threadpools

THREAD POOL: MOTIVAZIONI

- La creazione di un thread è costosa: richiede tempo e richiede attività di elaborazione da parte della JVM e del SO
- Creare un nuovo thread per ogni task risulta una soluzione improponibile, specialmente nel caso di task 'leggeri' molto frequenti.
- esiste un limite oltre il quale non risulta conveniente creare ulteriori threads
- Resource consumption
 - alloca uno stack per ogni thread
 - garbage collector stress



THREAD POOL: MOTIVAZIONI

Scenario: 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 contenente un numero predeterminato di thread (es. limite massimo per il numero di threads)
- obiettivo: controllare il numero massimo di thread che possono essere eseguiti concorrentemente
- evitare di avere un numero troppo alto di threads in competizione per le risorse disponibili
- diminuire il costo per l'attivazione/terminazione dei threads
- riusare lo stesso thread per l'esecuzione di più tasks

THREAD POOL: CONCETTI GENERALI

- 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 = esecutore di tasks.
- Thread Pool
 - **Permette di gestire l'esecuzione di task senza dover gestire esplicitamente il ciclo di vita dei thread**
 - 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 da parte del thread. L'esecuzione del task può essere ritardata se non vi sono risorse disponibili

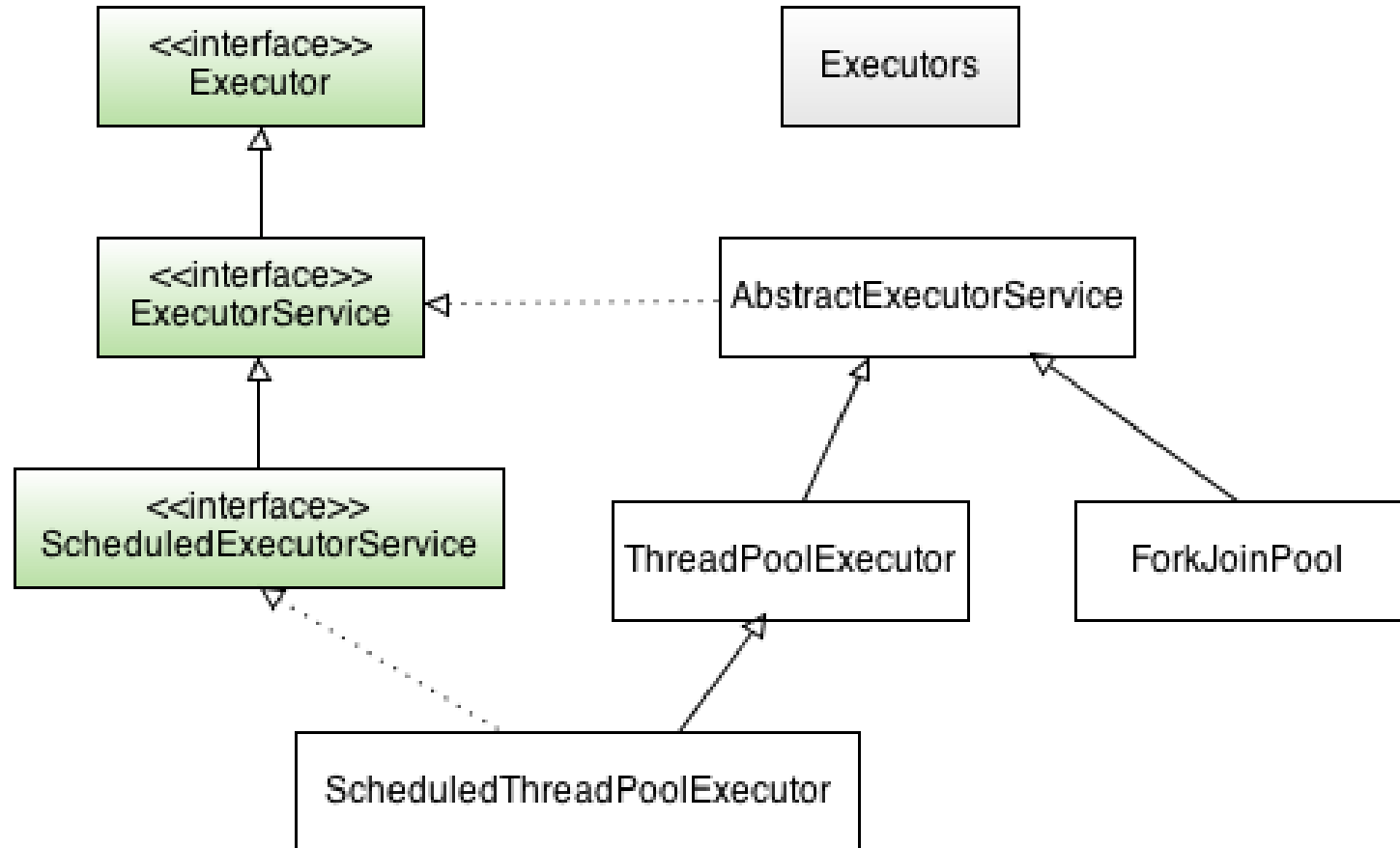
THREAD POOL: CONCETTI GENERALI

- L'utente crea il **pool** e stabilisce una **politica per la gestione dei thread del pool** che stabilisce:
 - quando i thread del pool **vengono attivati**: (al momento della creazione del pool, on demand, all'arrivo di un nuovo task,...)
 - se e quando è opportuno terminare un thread (ad esempio se non c'è un numero sufficiente di tasks da eseguire)
- L'utente invia i tasks al thread pool per la loro esecuzione
- Il supporto, al momento della sottomissione del task, può
 - **utilizzare un thread attivato in precedenza**, inattivo al momento dell'arrivo del nuovo task
 - **creare un nuovo thread**
 - **memorizzare il task** in una **struttura dati (coda)**, in attesa di eseguirlo
 - **respingere** la richiesta di esecuzione del task
- il numero di threads attivi nel pool può **variare dinamicamente**

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



JAVA THREADPOOL

- Interfacce che definiscono servizi generici di esecuzione
- Executor: esegue un task Runnable

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

- ExecutorService estende Executor con metodi che permettono di gestire il ciclo di vita del pool (es. Terminazione)

```
public interface ExecutorService extends  
    Executor { .. }
```

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

INVIO DI TASK AD UN SERVER

```
public class Main {  
    public static void main(String[] args) throws Exception  
    {  
        Server server=new Server();  
        for (int i=0; i<10; i++){  
            Task task=new Task("Task "+i);  
            server.executeTask(task);  
        }  
        server.endServer();}}
```

- creazione di un server
- invio di una sequenza di task al server. Il server eseguirà i task in modo concorrente utilizzando un thread pool.
- terminazione del server

DEFINIZIONE DI UN SERVER CONCORRENTE

```
import java.util.concurrent.*;

public class Server {

    private ThreadPoolExecutor executor;

    public Server( ) {

        executor=(ThreadPoolExecutor)Executors.newCachedThreadPool();

    }

    public void executeTask(Task task){

        System.out.printf("Server: A new task has arrived\n");
        executor.execute(task);

        System.out.printf("Server:Pool Size:%d\n",executor.getPoolSize());
        System.out.printf("Server:Active
                           Count:%d\n",executor.getActiveCount());

        System.out.printf("Server:Completed Tasks:%d\n",
                           executor.getCompletedTaskCount());

    }

    public void endServer() {

        executor.shutdown();

    }

}
```

NewCachedThreadPool

crea un pool con un comportamento predefinito:

- 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 sulla 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”

UN TASK CHE SIMULA UN SERVIZIO...

```
public class Task implements Runnable {  
    private String name;  
    public Task(String name){ this.name=name;}  
    public void run() {  
        System.out.printf("%s: Task %s \n",  
                           Thread.currentThread().getName(),name);  
        try{  
            Long duration=(long)(Math.random()*1000);  
            System.out.printf("%s: Task %s: Doing 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);  
    }  
}
```

Per semplicità in questo e nei successivi esempi il
singolo servizio sarà simulato inserendo delle attese
casuali (Thread.sleep())

OSSERVARE L'OUTPUT: IL RIUSO DEI THREAD

Server: A new task has arrived

Server: Pool Size: 1

pool-1-thread-1: Task Task 0

Server: Active Count: 1

Server: Completed Tasks: 0

pool-1-thread-1: Task Task 0: Doing a task during 1 seconds

Server: A new task has arrived

Server: Pool Size: 2

Server: Active Count: 1

pool-1-thread-1: Task Finished Task 0

pool-1-thread-2: Task Task 1

pool-1-thread-2: Task Task 1: Doing a task during 7 seconds

Server: Completed Tasks: 0

Server: A new task has arrived

Server: Pool Size: 2

pool-1-thread-1: Task Task 2

AUMENTARE IL RIUSO

```
import java.util.*;
public class Main {
    public static void main(String[] args) throws Exception{
        Server server=new Server();
        for (int i=0; i<10; i++){
            Task task=new Task("Task "+i);
            server.executeTask(task);
            Thread.sleep(5000);
        }
        server.endServer();
    }
}
```

La sottomissione di tasks al pool viene **distanziata di 5 secondi**. In questo modo se l'esecuzione precedente è terminata il programma **riutilizza sempre lo stesso thread**.

AUMENTARE IL RIUSO

Server: A new task has arrived

Server: Pool Size: 1

pool-1-thread-1: Task Task 0

Server: Active Count: 1

Server: Completed Tasks: 0

pool-1-thread-1: Task Task 0: Doing a task during 6 seconds

pool-1-thread-1: Task Finished Task 0

Server: A new task has arrived

Server: Pool Size: 1

pool-1-thread-1: Task Task 1

Server: Active Count: 1

pool-1-thread-1: Task Task 1: Doing a task during 2 seconds

Server: Completed Tasks: 1

pool-1-thread-1: Task Finished Task 1

AUMENTARE IL RIUSO

Server: A new task has arrived

Server: A new task has arrived

Server: Pool Size: 1

pool-1-thread-1: Task Task 2

Server: Active Count: 1

pool-1-thread-1: Task Task 2: Doing a task during 5 seconds

Server: Completed Tasks: 2

pool-1-thread-1: Task Finished Task 2

Server: A new task has arrived

Server: Pool Size: 1

Server: Active Count: 1

pool-1-thread-1: Task Task 3

newFixedThreadPool()

```
import java.util.concurrent.*;

public class Server {
    private ThreadPoolExecutor executor;

    public Server(){
        executor=(ThreadPoolExecutor)Executors.newFixedThreadPool(2);
    } ...
}
```

newFixedThreadPool(int N) crea un pool in cui:

- vengono creati N thread, 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
 - la coda è una LinkedBlockingQueue
 - se almeno un thread è inattivo, viene utilizzato quel thread

IL COSTRUTTORE THREAD POOL EXECUTOR

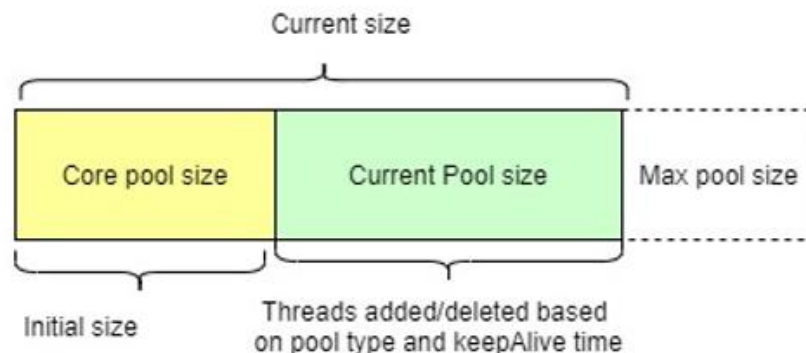
```
import java.util.concurrent.*;

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

- il costruttore più generale: consente di personalizzare la 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

- **CorePoolSize**: dimensione **minima** del pool, definisce il **core** del pool.
- I thread del core possono venire creati secondo le seguente modalità:
 - **On-demand** construction: per default, all'inizio i thread vengono creati via via che i task vengono sottomessi (anche i core thread), anche se qualche thread già creato del core è inattivo.
 - Obiettivo: riempire il pool prima possibile
 - **prestartAllCoreThreads()**: al momento della **creazione** del pool crea tutti i thread, anche se non ci sono task e questo comporta un'attesa dei thread
 - quando sono stati creati tutti i threads del core, la politica varia (vedi pagina successiva)
- **MaxPoolSize**: dimensione **massima** del pool.
 - non più di MaxpoolSize threads nel pool, anche se vi sono task da eseguire e tutti i threads sono occupati nell'elaborazione di altri tasks.



THREAD POOL EXECUTOR – policy di riferimento

- Alla sottomissione di un nuovo task, se tutti i thread del core sono stati creati
 - se un thread del core è inattivo, il task viene assegnato ad esso
 - altrimenti, se la coda, passata come ultimo parametro del costruttore, non è piena, il task viene inserito nella coda
 - i task vengono poi prelevati dalla coda ed inviati ai thread disponibili
 - altrimenti (coda piena e tutti i thread del core stanno eseguendo un task) si crea un nuovo thread attivando così k thread finché vale
$$\text{corePoolSize} \leq k \leq \text{MaxPoolSize}$$
 - altrimenti (coda piena e sono attivi MaxPoolSize threads), il task viene respinto
- E' possibile scegliere diversi tipi di coda (tipi derivati da BlockingQueue). Il tipo di coda scelto influisce sullo scheduling.

ELIMINAZIONE DI THREAD INUTILI

Supponiamo che un thread termini l'esecuzione di un task, e che il pool contenga k threads:

- Se $k \leq \text{core}$: il thread **si mette in attesa** di nuovi tasks da eseguire. L'attesa è indefinita.
- Se $k > \text{core}$, ed il thread non appartiene al core si considera il **timeout T** definito al momento della costruzione del thread pool
 - se nessun task viene sottomesso **entro T** , il thread termina la sua esecuzione, riducendo così il numero di threads del pool
 - Per definire il timeout: occorre specificare
 - un valore (es: 50000) e
 - l'unità di misura utilizzata (es: TimeUnit.MILLISECONDS)

THREAD POOL EXECUTOR: CODE

- **SynchronousQueue**: dimensione uguale a 0. Ogni nuovo task T
 - viene eseguito immediatamente oppure respinto.
 - eseguito immediatamente se esiste un thread inattivo oppure se è possibile creare un nuovo thread (numero di threads \leq MaxPoolSize)
 - NB una put() in una SynchronousQueue si blocca finchè non c'è una corrispondente take()
- **LinkedBlockingQueue**:
 - E' sempre possibile accodare un nuovo task, nel caso in cui tutti i threads siano attivi nell'esecuzione di altri tasks
 - la dimensione del pool **non può superare core**
- **ArrayBlockingQueue**: dimensione limitata, stabilita dal programmatore

THREAD POOL EXECUTOR: ISTANZE

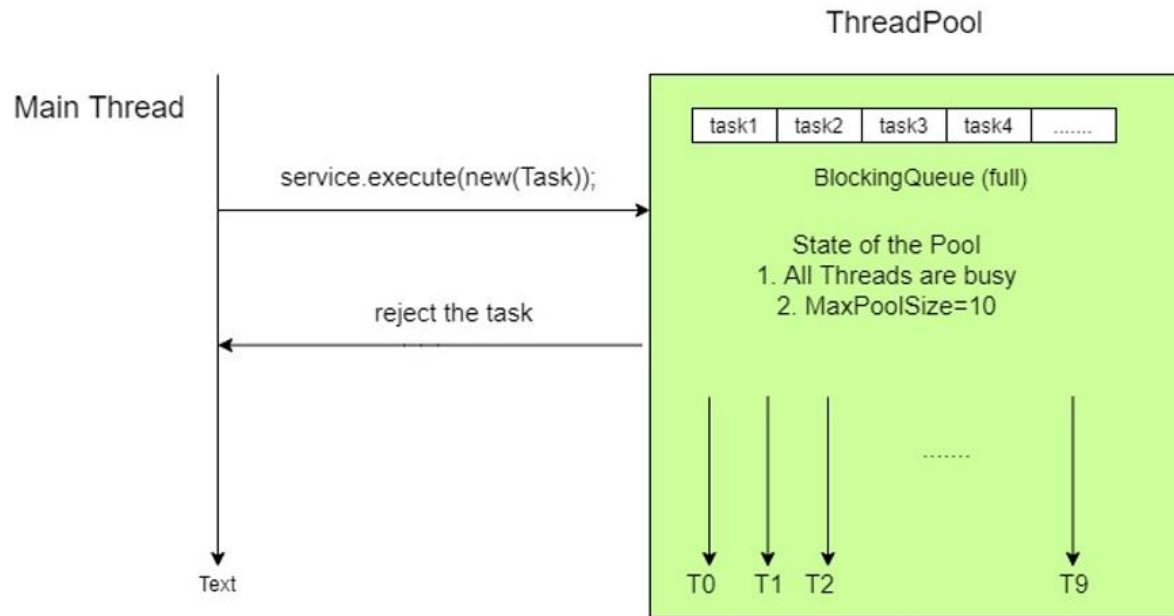
newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int nThreads) {  
    return new ThreadPoolExecutor(nThreads, nThreads, 0L,  
                                   TimeUnit.MILLISECONDS, new  
                                   LinkedBlockingQueue<Runnable>());  
}
```

newCachedThreadPool

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

Task rejection



- *New tasks submitted in method `execute(Runnable)` will be rejected when the Executor has been shut down, and also when the Executor uses finite bounds for both maximum threads and work queue capacity, and is saturated.*
- *In either case, the execute method invokes the `RejectedExecutionHandler.rejectedExecution(Runnable, ThreadPoolExecutor)` method of its `RejectedExecutionHandler` (from Java Documentation)*
 - Per default può sollevare una eccezione
 - può semplicemente scartarli
 - può adottare politiche più complesse

Rejection

```
import java.util.concurrent.*;
public class RejectedTaskMain {

    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 si utilizzi un ExecutorService poiché
 - i tasks vengono eseguito in modo **asincrono** rispetto al loro invio al threadpool.
 - in un **certo istante**, alcuni task inviati precedentemente possono essere **completati**, alcuni in **esecuzione**, alcuni **in coda**.
 - un thread del pool può rimanere attivo anche quando ha terminato l'esecuzione di un task
- poiché alcuni threads possono essere sempre attivi, JAVA mette a disposizione dell'utente alcuni metodi che permettono di terminare l'esecuzione del pool

EXECUTORS: TERMINAZIONE

La terminazione del pool può avvenire

- in **modo graduale**: “finisci ciò che hai iniziato, ma non iniziare nuovi tasks”.
- in **modo istantaneo**. “stacca la spina immediatamente”
- **shutdown()** graceful termination.
 - nessun task viene accettato dopo che la shutdown() è stata invocata.
 - tutti i tasks sottomessi in precedenza e non ancora terminati vengono eseguiti, compresi quelli la cui esecuzione non è ancora iniziata (quelli accodati).
 - successivamente tutti i threads del pool terminano la loro esecuzione
- **shutdownNow()** immediate termination
 - non accetta ulteriori tasks, ed elimina i tasks non ancora iniziati
 - restituisce una lista dei tasks che sono stati eliminati dalla coda
 - **tenta di terminare** l'esecuzione dei thread che stanno eseguendo i tasks (come?)

EXECUTORS: TERMINAZIONE

Life-cycle di un pool (execution service)

- `running`
- `shutting down`
- `terminated`
- Un pool viene creato nello stato `running`, quando viene invocata una `Shutdown()` o una `ShutdownNow()` passa allo stato `shutting down`, quando tutti i thread sono terminati passa nello stato `terminated`
- I task sottomessi per l'esecuzione ad un pool in stato `Shutting Down` o `Terminated` possono essere gestiti da un `rejected execution handler`

EXECUTORS: TERMINAZIONE

Alcuni metodi definiti dalla interfaccia `ExecutorService` per gestire la terminazione del pool

- **void** `shutdown()`
- `List<Runnable> shutdownNow()` restituisce la lista di threads eliminati dalla coda
- **boolean** `isShutdown()`
- **boolean** `isTerminated()`
- **boolean** `awaitTermination(long timeout, TimeUnit unit)`
attende che il pool passi in stato `Terminated`

Per capire se l'esecuzione del pool è terminata:

- **attesa passiva**: invoco la **`awaitTermination()`**, si blocca finché i task hanno completato l'esecuzione o scade il timeout
- **attesa attiva**: invoco ripetutamente la **`isTerminated()`**

EXECUTORS: TERMINAZIONE

ShutdownNow()

- implementazione **best effort**
- non garantisce la terminazione immediata dei threads del pool
- implementazione generalmente utilizzata: invio di **una interrupt()** ai thread in esecuzione nel pool
- se un thread non risponde all'interruzione non termina
- infatti, 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

CALLABLE E FUTURE

- un oggetto di tipo Runnable incapsula un'attività che viene eseguita in modo asincrono
- una Runnable si può considerare un metodo asincrono, senza parametri e che non restituisce un valore di ritorno
- per definire un task che restituisca un valore di ritorno occorre utilizzare le seguenti interfacce:
 - **Callable**: Interfaccia per definire un task che può restituire un risultato e sollevare eccezioni (al posto di Runnable)
 - **Future**: interfaccia per rappresentare il risultato di una computazione asincrona. Definisce metodi
 - per controllare se la computazione è terminata
 - per attendere la terminazione di una elaborazione (eventualmente per un tempo limitato)
 - per cancellare una elaborazione,

Interfaccia CALLABLE

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

- contiene il solo metodo call, analogo al metodo run() dell'interfaccia Runnable
- per definire il codice del task, occorre implementare il metodo call a differenza del metodo run(), il metodo call() puo restituire un valore e sollevare eccezioni
- il parametro di tipo <V> indica il tipo del valore restituito
 - ad esempio: Callable <Integer> rappresenta una elaborazione asincrona che restituisce un valore di tipo Integer

CALLABLE: esempio

- 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 = <approssimazione di  $\pi$ >
        ...
        return result;
    }
}
```

INTERFACCIA FUTURE

- Il valore restituito dalla Callable, acceduto mediante un oggetto di tipo `<Future>`, che rappresenta il risultato della computazione
- Se si usano i thread pools, si sottomette direttamente l'oggetto di tipo Callable al pool mediante il metodo `submit` e si ottiene il riferimento a un oggetto di tipo `<Future>`
- E' possibile invocare sull'oggetto Future restituito diversi metodi che consentono di individuare se il thread ha terminato la computazione del valore richiesto

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 finché il thread non ha prodotto il valore richiesto e restituisce il valore calcolato
- E' possibile definire un tempo massimo di attesa della terminazione del task, dopo cui viene sollevata una **TimeoutException**
- E' possibile cancellare il task e verificare se la computazione è terminata oppure è stata cancellata

Thread Pooling e Callable

```
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(.....){ }
        }
    }
}
```

Sincronizzazione

Condivisione di risorse

Lock espliciti e conditions

CONDIVIDERE RISORSE

- 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**: una race condition si verifica quando più thread o processi leggono o scrivono su un dato condiviso, e **il risultato finale dipende dall'ordine con cui i threads sono stati schedulati**
 - **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
 - **synchronized** keyword e concetto di **monitor**

UN ESEMPIO DI RACE CONDITION

un esempio in cui si verifica una **race condition**:

- si considera un conto bancario e due threads che vi accedono in modo concorrente
 - il thread **Company** versa denaro sul conto corrente
 - il thread **Bancomat** preleva denaro dal conto corrente
- 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** o **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 descheduled prima di avere completato l'esecuzione del metodo
- la risorsa viene lasciata in uno stato inconsistente
- un esempio:
 - primo thread esegue subtractAccount: tmp=1000, poi descheduled prima di completare il metodo
 - secondo thread: completa il metodo addAccount, balance=2000
 - ritorna in esecuzione primo thread: balance=0
- **Classe Thread Safe**: l'esecuzione concorrente dei metodi definiti in una classe thread safe 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**

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();  
    }  
}
```

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, invece, ottengo i seguenti valori per 3 esecuzioni distinte del programma

12349

12639

12170

- **read-modify-write pattern**: JAVA bytecode generato per il comando ++count

getField #2

iconst_1

iadd

putfield #2

Varie istruzioni, il thread che le esegue può essere interrotto in un punto qualsiasi

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

MECCANISMI DI SINCRONIZZAZIONE

- JAVA offre diversi meccanismi per la sincronizzazione di threads
- meccanismi a basso livello
 - **lock()**
 - **variabili di condizione** associate a **lock()**
- meccanismi ad alto livello
 - parola chiave **synchronized()**
 - **wait(), notify(), notifyAll()**
 - **monitors**
- il nostro approccio:
 - iniziamo con i meccanismi a basso livello, con l'obiettivo di capire meglio quelli ad alto livello
 - introduciamo poi quelli ad alto livello motivando la ragione per cui sono stati introdotti.
 - In entrambi i casi prima vediamo le interazioni di tipo competitivo e poi di tipo cooperativo