Reti e Laboratorio
Modulo Laboratorio III
a.a. 2026-2027

docente: Laura Ricci

laura.ricci@unipi.it

Lezione 2

ThreadPoolExecutor, ScheduledPool, BlockingQueues

24/09/2026

JAVA THREADPOOL: IMPLEMENTAZIONE

- fino a `JAVA 4` la programmazione del threadpool è stata 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 specifiche politiche per la gestione del pool
    - tipo di coda
    - elasticità del threadpool
- il meccanismo introdotto permette una migliore strutturazione del codice poichè tutta la gestione dei threads può essere delegata al supporto

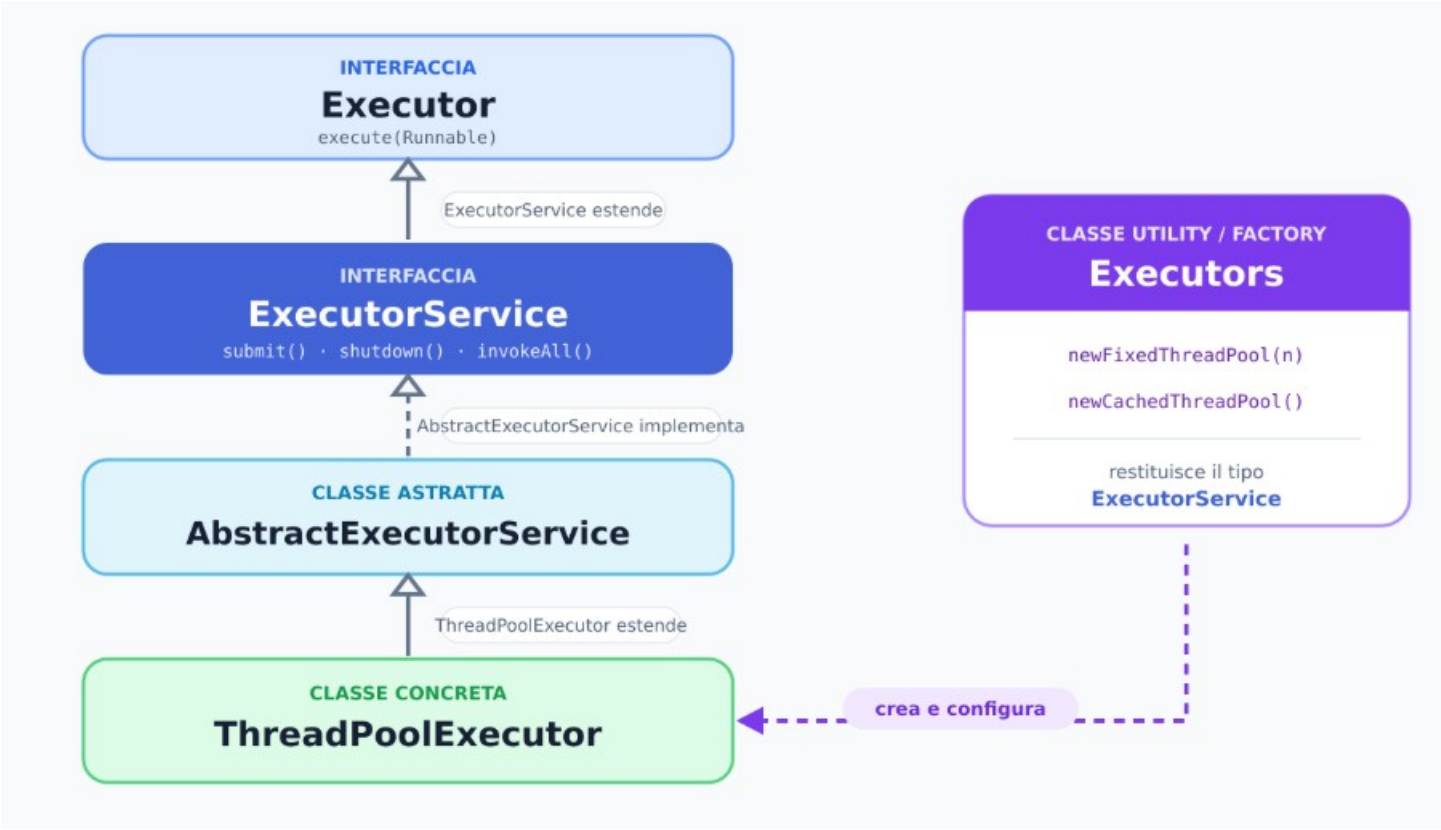
THREADPOOL: GERARCHA INTERFACCE/CLASSI

- 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()`

THREADPOOL: GERARCHA INTERFACCE/CLASSI



THREADPOOL: UN ESEMPIO

- realizzare un programma `Java` che simuli un server per l'esecuzione di attività di intelligenza artificiale, come la generazione di un'immagine ola traduzione di un documento
- ogni attività è caratterizzata da:
  - un identificatore numerico
  - una descrizione
  - un tempo di elaborazione
- il server dispone di due thread di elaborazione e può quindi eseguire contemporaneamente al massimo due attività
- quando un thread inizia un'attività, il programma deve visualizzare il suo nome, l'identificatore dell'attività e la relativa descrizione
- il tempo necessario per l'elaborazione deve essere simulato mediante una sospensione del thread effettuata mediante `sleep`
- al termine dell'attività, il programma deve indicare quale thread l'ha completata

IL TASK CHE IMPLEMENTA IL PROMPT

```
package AICenter;

class AIRequest implements Runnable {

    private String prompt;

    public AIRequest(String prompt) {
        this.prompt = prompt;
    }

    @Override
    public void run() {
        String worker = Thread.currentThread().getName();
        System.out.println(worker + " is processing: " + prompt);
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        System.out.println(worker + " completed: " + prompt);
    }
}
```

IL TASK CHE IMPLEMENTA IL PROMPT

```
package AICenter;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class AICenter {

    public static void main(String[] args) {

        ExecutorService aiWorkers =
                Executors.newFixedThreadPool(2);

        aiWorkers.execute(new AIRequest("Generate an image"));
        aiWorkers.execute(new AIRequest("Translate a document"));
        aiWorkers.execute(new AIRequest("Detect fake news"));
        aiWorkers.execute(new AIRequest("Summarize an article"));
        aiWorkers.execute(new AIRequest("Recognize a face"));

        aiWorkers.shutdown();
    }
}
```

FIXEDTHREADPOOL

- un tipo di threadpool con comportamento predefinito
- viene creato un numero fisso di thread: n thread, n fissato al momento dell'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

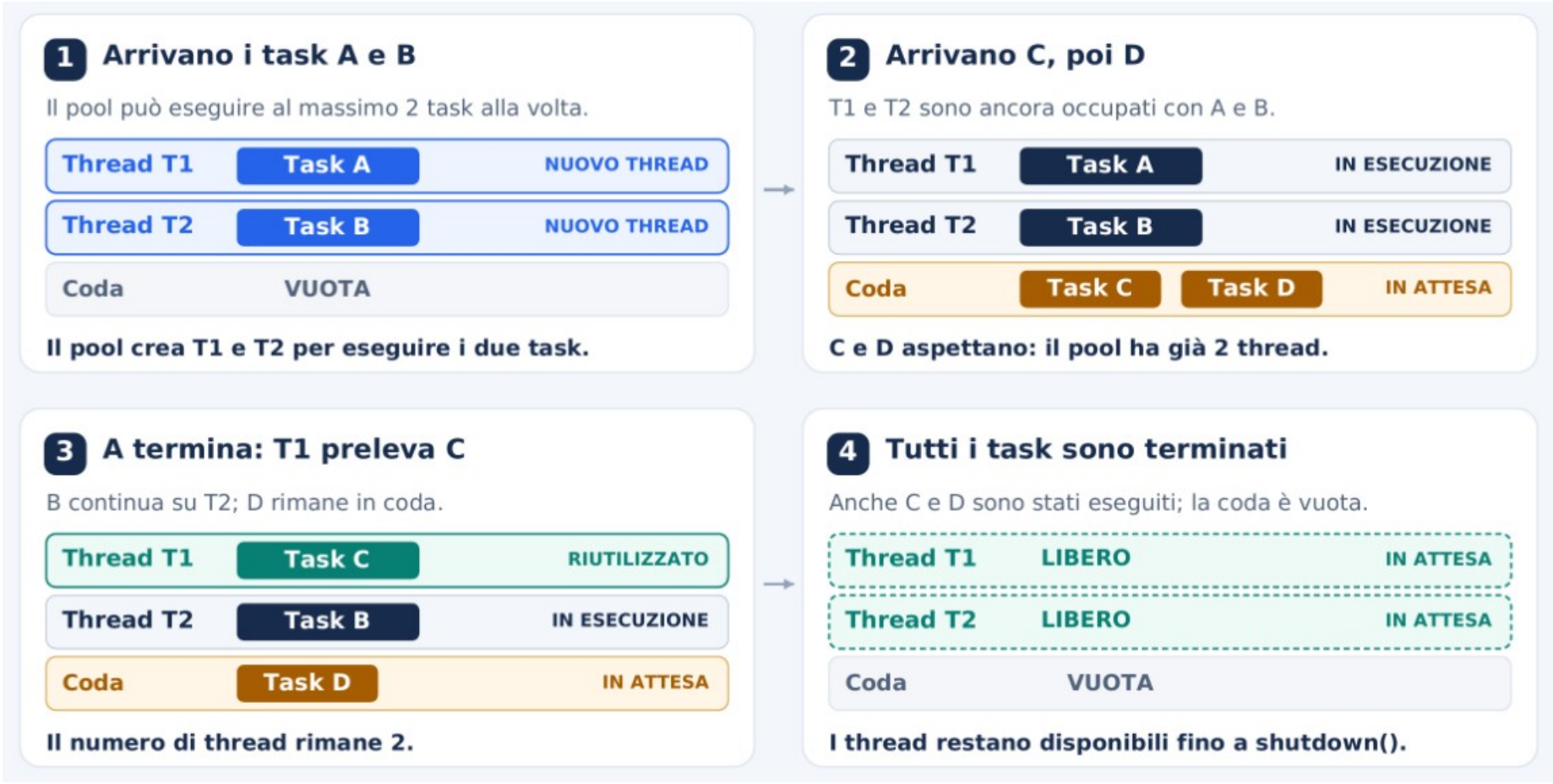
L'OUTPUT GENERATO

```
pool-1-thread-2 is processing: Translate a document
pool-1-thread-1 is processing: Generate an image
pool-1-thread-1 completed: Generate an image
pool-1-thread-1 is processing: Detect fake news
pool-1-thread-2 completed: Translate a document
pool-1-thread-2 is processing: Summarize an article
pool-1-thread-2 completed: Summarize an article
pool-1-thread-1 completed: Detect fake news
pool-1-thread-2 is processing: Recognize a face
pool-1-thread-2 completed: Recognize a face
```

nota:

- lo stesso thread riutilizzato per più tasks

FIXEDTHREADPOOL: RIASSUNTO



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"*

CACHEDTHREADPOOL

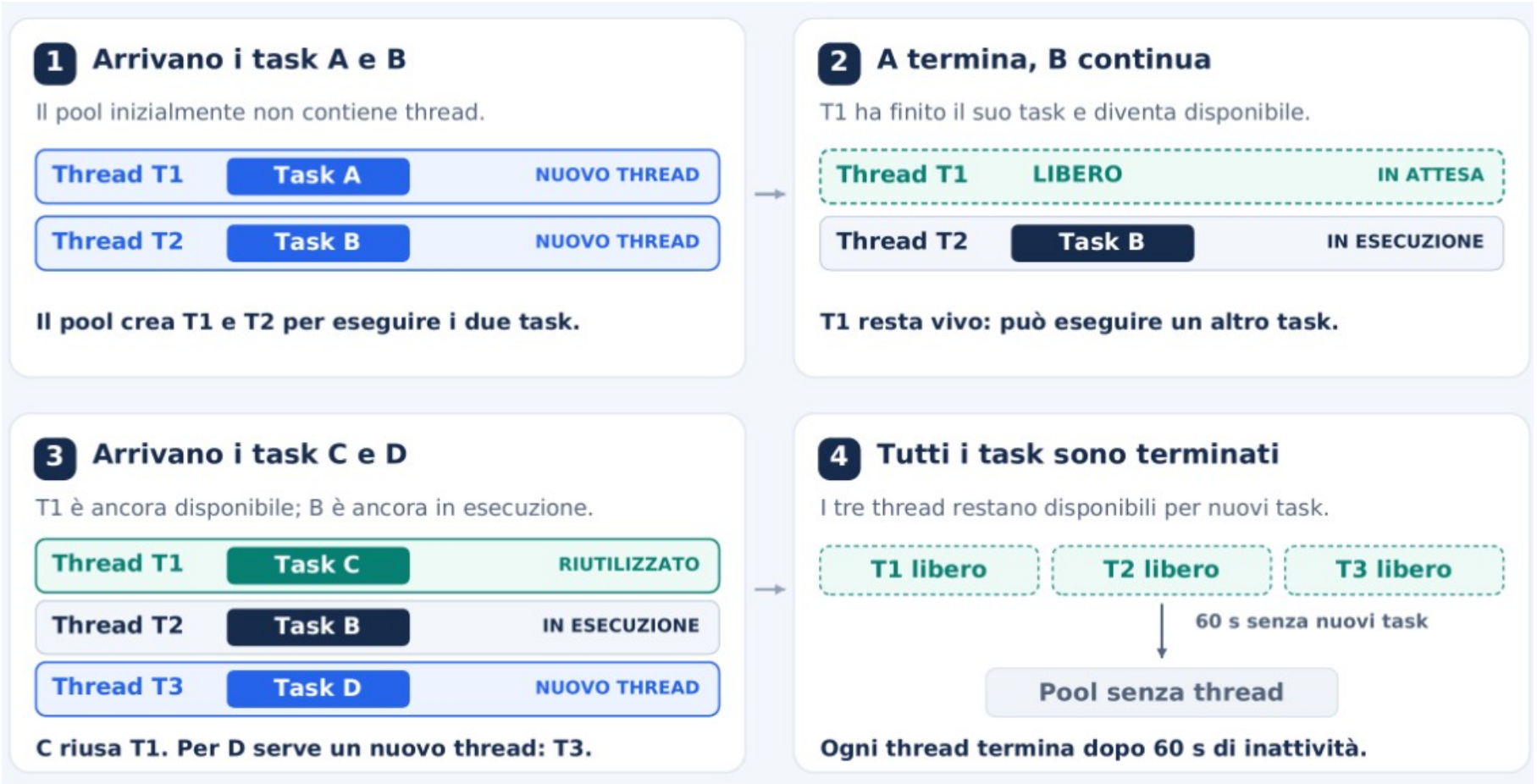
```
package AICenter;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class AICenter {

    public static void main(String[] args) {

        ExecutorService aiWorkers =
                Executors.newCachedThreadPool();

        aiWorkers.execute(new AIRequest("Generate an image"));
        aiWorkers.execute(new AIRequest("Translate a document"));
        aiWorkers.execute(new AIRequest("Detect fake news"));
        aiWorkers.execute(new AIRequest("Summarize an article"));
        aiWorkers.execute(new AIRequest("Recognize a face"));

        aiWorkers.shutdown();
    }
}
```

OUTPUT GENERATO

```
pool-1-thread-4 is processing: Summarize an article
pool-1-thread-3 is processing: Detect fake news
pool-1-thread-1 is processing: Generate an image
pool-1-thread-2 is processing: Translate a document
pool-1-thread-5 is processing: Recognize a face
pool-1-thread-2 completed: Translate a document
pool-1-thread-3 completed: Detect fake news
pool-1-thread-5 completed: Recognize a face
pool-1-thread-4 completed: Summarize an article
pool-1-thread-1 completed: Generate an image
```

attivato un nuovo thread per ogni nuovo task

CACHEDTHREADPOOL : RIASSUNTO

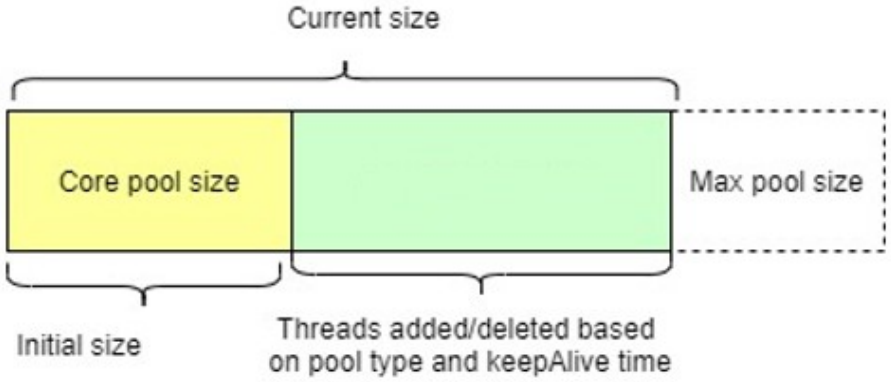


THREAD POOL EXECUTOR

```
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: consente la 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
  - tutti 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

THREAD POOL EXECUTOR: 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 un thread è inattivo e 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`

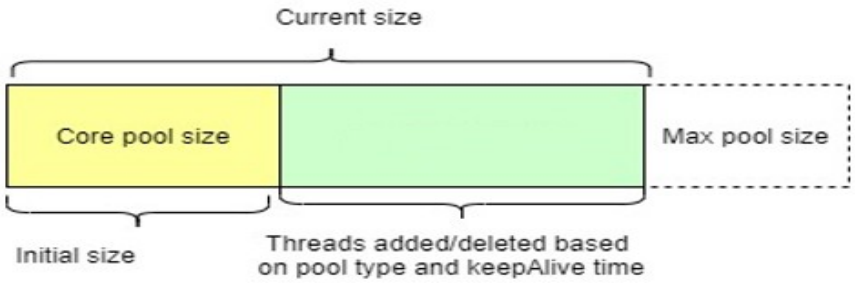
THREAD POOL EXECUTOR: GESTIONE CODA

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

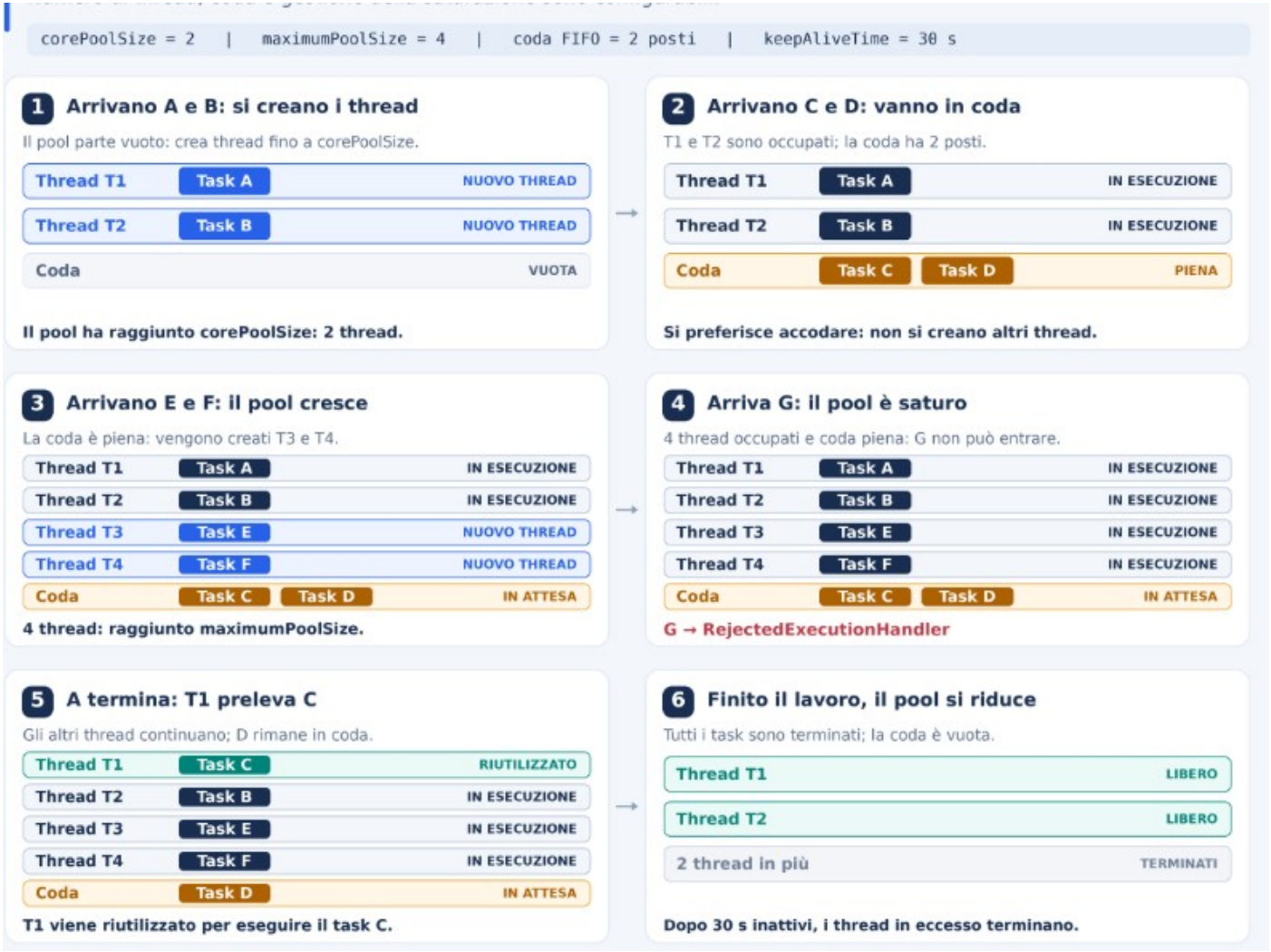
- se un thread del core è inattivo, il task viene assegnato ad esso
  - se tutti i thread del core stanno eseguendo un task e la coda non è piena, il nuovo task viene inserito nella coda: i task verranno quindi poi prelevati dalla coda ed inviati ai thread disponibili
- se tutti i thread del core stanno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

$$\texttt{corePoolSize} \leq k \leq \texttt{MaxPoolSize}$$

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



THREAD POOL EXECUTOR: RIASSUNTO



THREAD POOL EXECUTOR: RIASSUNTO

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

THREAD POOL EXECUTOR: ISTANZE

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

THREAD POOL EXECUTOR: ISTANZE

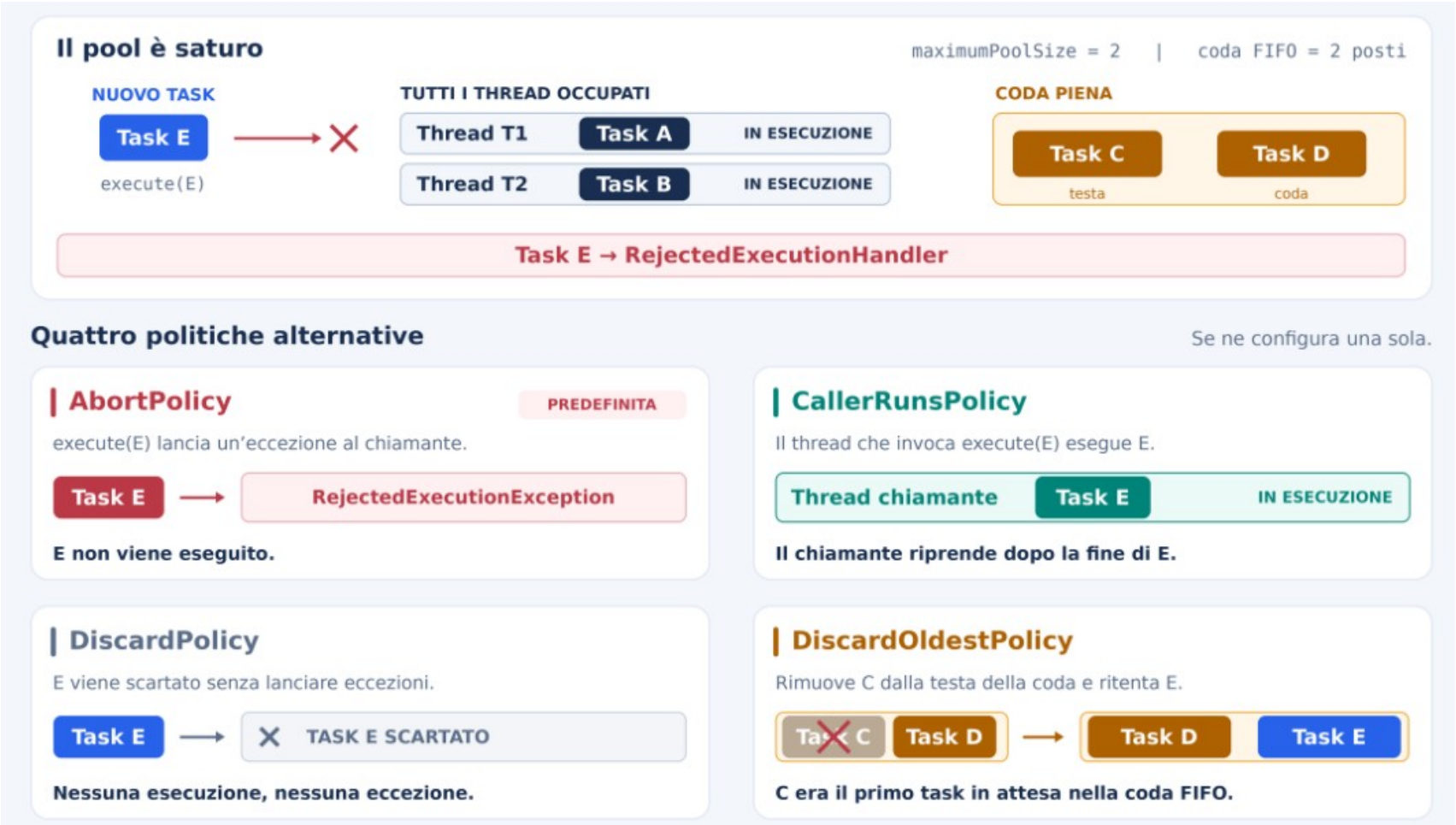
POOL	QUEUE TYPE	WHY?
FixedThreadPool	LinkedBlockingQueue	Threads are limited, thus unbounded queue to store tasks Note: since queue can never become full, new threads are never created
CachedThreadPool	SynchronousQueue	Threads are unbounded, thus no need to store the tasks. Create the new thread and 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,....);
```

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



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 `ExecutorService` 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 dell'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

ALTRI TIPI DI THREAD POOL

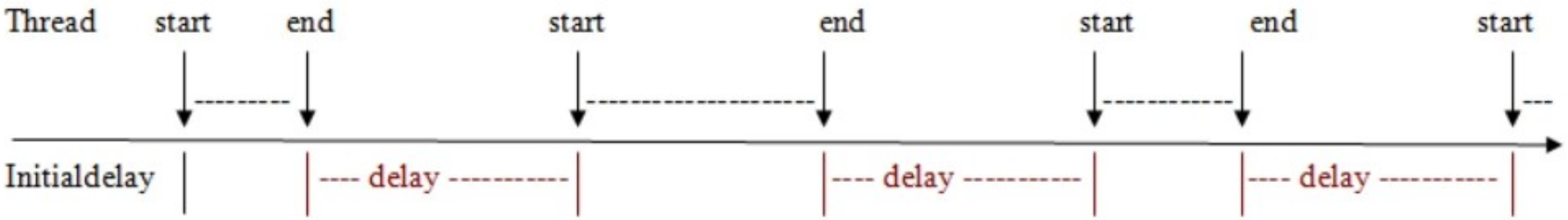
- `Single Threaded Executor`
  - un singolo thread
  - equivalente ad invocare un `FixedThreadPool` di dimensione 1
  - utilizzo: assicurare che i task vengano eseguiti nell'ordine con cui si trovano in coda (sequenzialmente), ma con riutizzo dello stesso thread
- `Scheduled Thread Pool`
  - distanziare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

SCHEDULED EXECUTOR SERVICE

- l'interfaccia `ScheduledExecutorService` da la possibilità di schedulare un task periodicamente
  - single shot: esegue un task una sola volta, dopo un certo periodo di tempo (delay)
  - `schedule(Runnable command, long delay, TimeUnit unit)`
- multiple shots: esegue un task più volte, periodicamente
  - `scheduleAtFixedRate(Runnable command, long initialDelay, long delay, TimeUnit unit)`
  - `scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)`

SCHEDULE_WITH_FIXED_DELAY

- `scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)`
- qualunque sia la durata dell'esecuzione di un task, la "pausa" tra due esecuzioni successive di un task è sempre della stessa lunghezza, ed è detta delay
- quindi esegue un task dopo un intervallo iniziale, poi lo ripete periodicamente con delay tra la terminazione di una esecuzione e l'inizio della successiva



`scheduleWithFixedDelay`

verifichiamo che questo sia l'effettivo comportamento!

SCHEDULED EXECUTOR SERVICE

- realizzare un programma `Java` che simuli un servizio di backup periodico sul cloud.
- ogni backup richiede un tempo di esecuzione casuale compreso tra 2 e 6 secondi
- al termine di ciascun backup, il sistema deve attendere 3 secondi prima di avviare quello successivo
- il programma deve utilizzare un `ScheduledExecutorService` e il metodo `scheduleWithFixedDelay()`
- Per ogni backup deve visualizzare il numero progressivo, la durata prevista e un messaggio di completamento
- dopo `30` secondi, il servizio di backup deve essere arrestato

SCHEDULE_WITH_FIXED_DELAY

```
package CloudBackupPackage;

import java.util.concurrent.ThreadLocalRandom;
import java.time.*;

class CloudBackup implements Runnable {

    private int backupNumber = 1;
    @Override
    public void run() {
        // Durata casuale compresa tra 2 e 6 secondi
        int backupTime =
                ThreadLocalRandom.current().nextInt(2, 7);

        System.out.println(
            "Uploading backup #" + backupNumber +
            " (" + backupTime + " seconds)..."
        );

        try {
            Thread.sleep(backupTime * 1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return;
        }

        System.out.println(
            "Backup #" + backupNumber + " completed. " + Instant.now() +"\n");

        backupNumber++;
    }
}
```

SCHEDULE_WITH_FIXED_DELAY

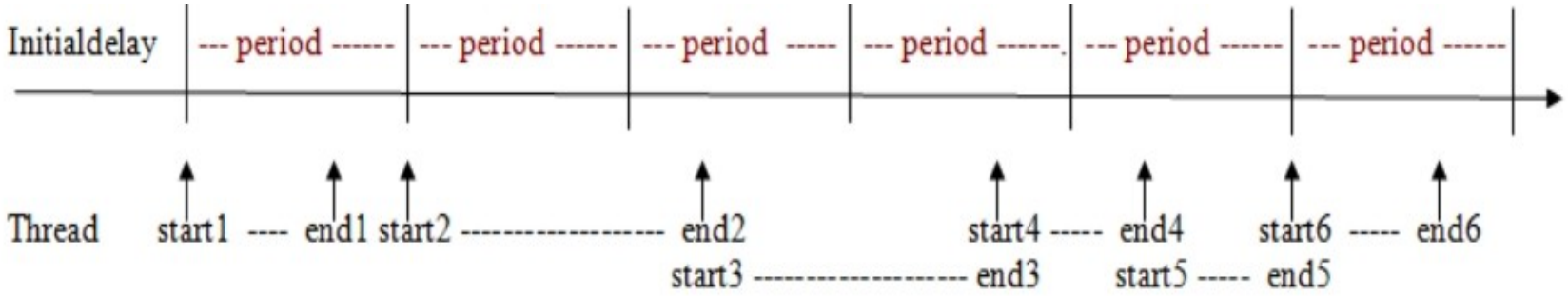
```
package CloudBackupPackage;

import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class BackupService {

    public static void main(String[] args)
            throws InterruptedException {

        ScheduledExecutorService scheduler =
                Executors.newScheduledThreadPool(1);

        scheduler.scheduleWithFixedDelay(
                new CloudBackup(),
                0,                  // primo backup immediato
                3,                  // attesa dopo ogni backup
                TimeUnit.SECONDS
        );

        Thread.sleep(20000);

        scheduler.shutdownNow();
    }
}
```

SCHEDULE_WITH_FIXED_DELAY

```
Uploading backup #1 (3 seconds)...
Backup #1 completed. 2026-09-23T22:23:45.653952300Z

Uploading backup #2 (2 seconds)...
Backup #2 completed. 2026-09-23T22:23:50.700286400Z

Uploading backup #3 (3 seconds)...
Backup #3 completed. 2026-09-23T22:23:56.714969700Z

Uploading backup #4 (4 seconds)...
Backup #4 completed. 2026-09-23T22:24:03.727335600Z

Uploading backup #5 (3 seconds)...
Backup #5 completed. 2026-09-23T22:24:09.730241300Z
```

SCHEDULE_AT_FIXED_RATE

- `scheduleAtFixedRate(Runnable command, long initialDelay, long delay, TimeUnit unit)`
- esegue un task dopo un intervallo iniziale, poi lo ripete periodicamente, ad intervalli costanti.
- se la durata del task è minore della durata dell'intervallo non ci sono problemi
- se il tempo di esecuzione del task è maggiore del periodo specificato, le sue seguenti esecuzioni possono essere ritardate, ma la JVM cerca di "recuperare il ritardo", facendo eseguire immediatamente i task in ritardo



`scheduleAtFixedRate`

verifichiamo che questo sia l'effettivo comportamento!

UN TASK "COUNTDOWN"

- nel programma precedente, sostituire

```
scheduler.scheduleAtFixedRate(
    new CloudBackup(),
    0,                  // ritardo iniziale
    5,                  // periodo tra gli istanti di avvio
    TimeUnit.SECONDS
);
```

UN BEEP PERIODICO...

```
import java.util.concurrent.*;
import java.awt.*;
public class BeepClockS implements Runnable {
    public void run() {
         Toolkit.getDefaultToolkit().beep();
        }
    public static void main(String[] args) {
        ScheduledExecutorService scheduler
                                = Executors.newSingleThreadScheduledExecutor();
        Runnable task = new BeepClockS();
        int initialDelay = 4;
        int periodicDelay = 2;
        scheduler.scheduleAtFixedRate(task, initialDelay, periodicDelay,
                                                    TimeUnit.SECONDS);}}
```

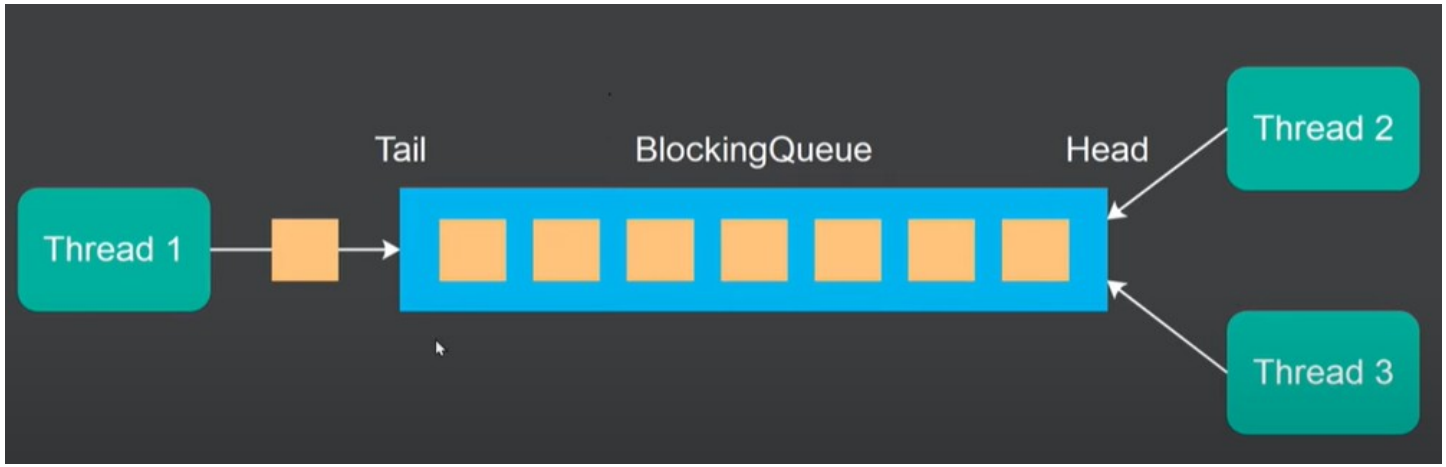
CONDIVIDERE RISORSE TRA THREADS

- un insieme di thread vogliono condividere 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 a cui si effettua l'accesso ad una risorsa condivisa e che deve essere eseguito da un thread per volta
- necessario implementare classi thread safe
  - il codice dei metodi della classe può essere utilizzato/condiviso in un ambiente concorrente senza provocare inconsistenze/comportamenti inaspettati

ALTERNATIVE PER DEFINIRE CLASSI THREAD SAFE

- alternative per definire classi thread safe: usare
  - classi thread safe predefinite
    - concurrent-aware interfaces
      - `Interfaces: Blocking Queue, TransferQueue, Blocking Dequeue, ConcurrentMap, ConcurrentNavigable Map`
    - concurrent-aware classes
      - `LinkedBlockingQueue`
      - `ArrayBlockingQueue`
      - `PriorityBlockingQueue`
      - `DelayQueue`
      - `SynhronousQueue`
      - `CopyOnWriteArrayList`
      - `CopyOnWriteArraySet`
      - `ConcurrentHahsMap`
  - i monitor
  - le lock a basso livello (non le vedremo)

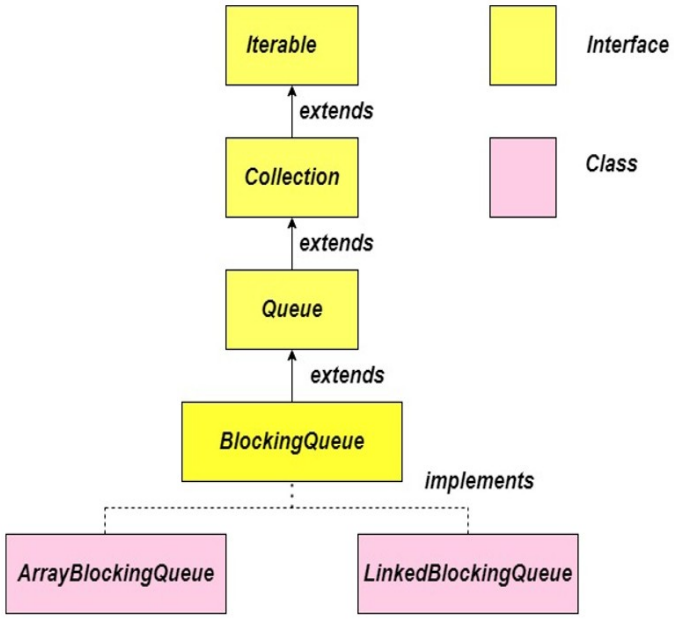
JAVA BLOCKING QUEUE



- `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`)?
  - pensata per essere utilizzata in un ambiente multithreaded
  - permettere una corretta sincronizzazione tra i thread che inseriscono e quelli che eliminano elementi dalla coda
    - `Thread1` si blocca se la coda è piena, `Thread2` e `Thread3` se è vuota
  - implementa una corretta sincronizzazione tra thread

BLOCKING QUEUE: IMPLEMENTAZIONI

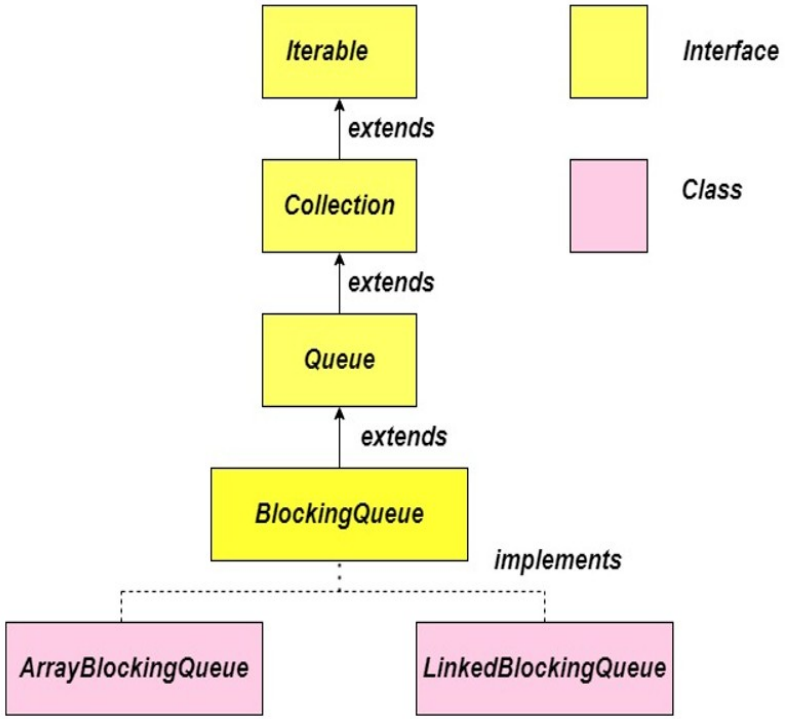
```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

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
           }}
```

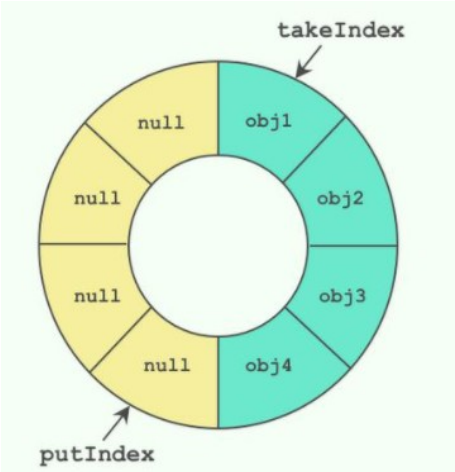


QUALI CODE UTILIZZEREMO MAGGIORMENTE?

`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)



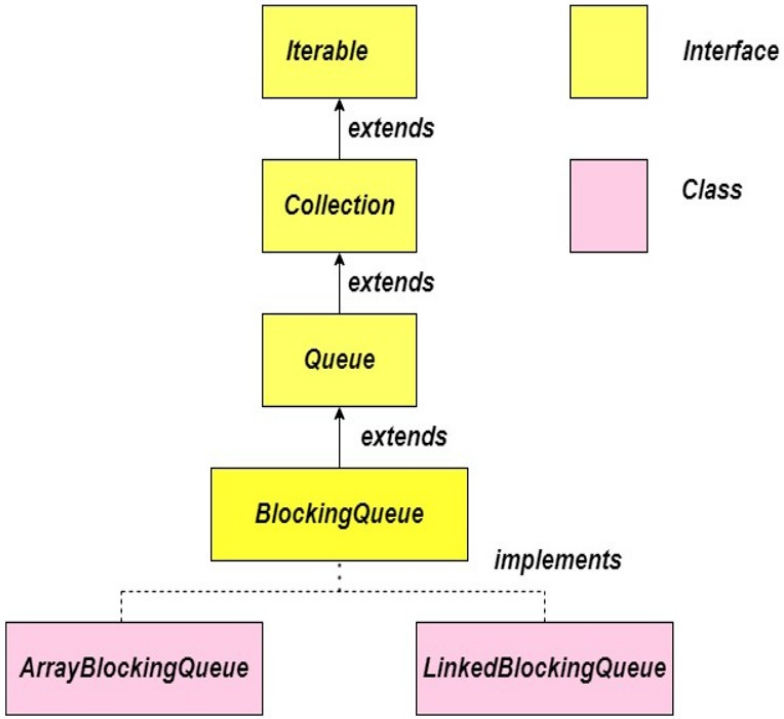


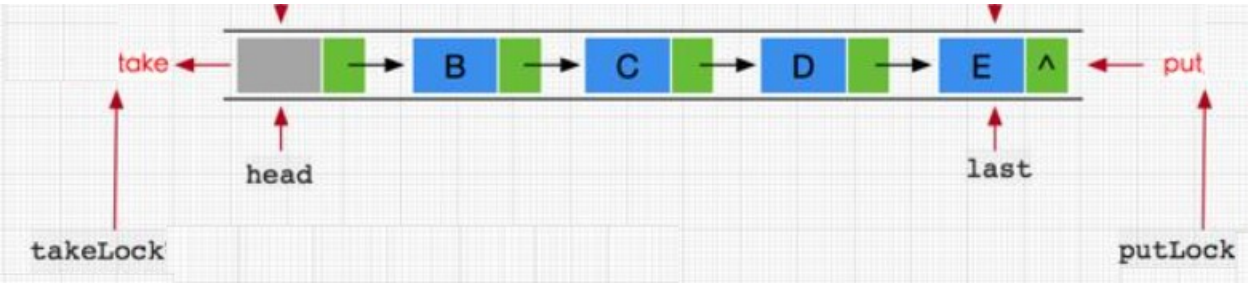
E QUALI CODE UTILIZZEREMO MAGGIORMENTE?

`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





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()`		

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

- 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 4: 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.