Reti e Laboratorio III
Modulo Laboratorio III
AA. 2022-2023

docente: Laura Ricci

laura.ricci@unipi.it

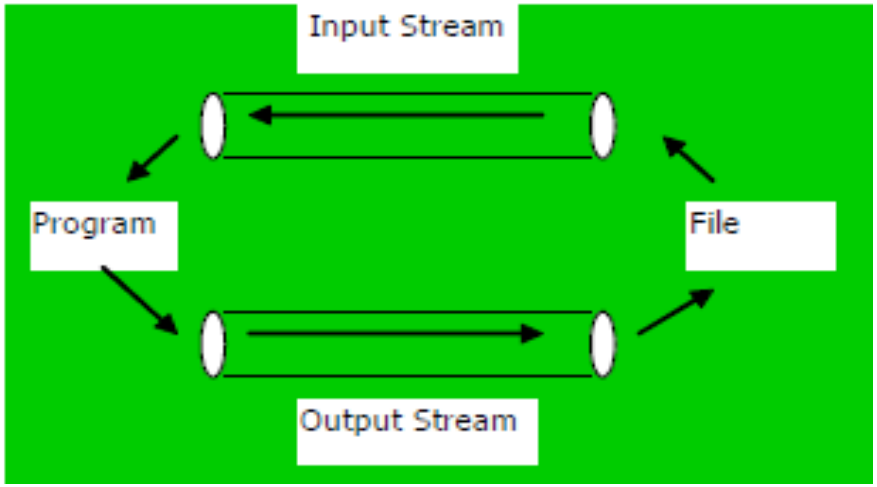
Lezione 2

Stream: richiami, ThreadPoolExecutor

22/9/2022

STREAM: IL PACKAGE JAVA.IO

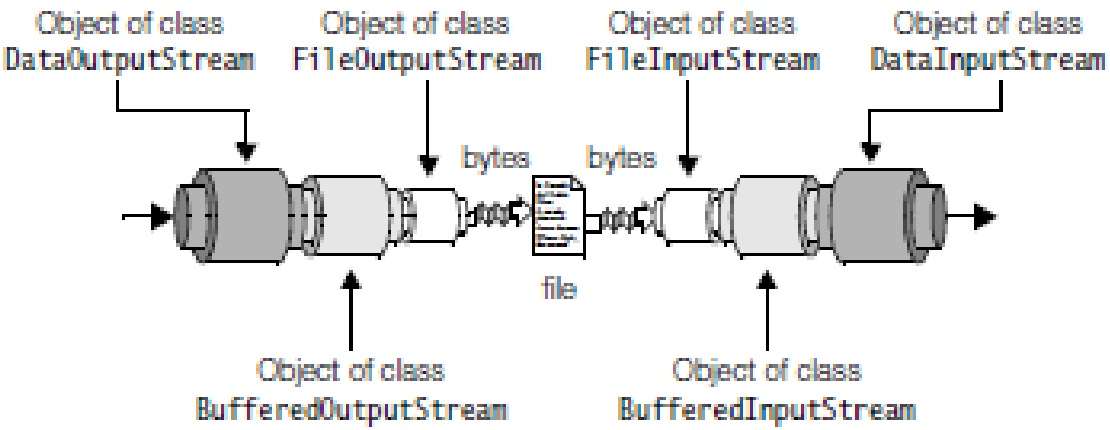
- definisce i concetti base per gestire l'I/O da/verso qualsiasi sorgente/destinazione
- basato sul concetto base di stream che è un canale di comunicazione:
  - monodirezionale
  - ad eccesso sequenziale, mantengono l'ordinamento FIFO
  - di uso generale
  - adatto a trasferire byte o caratteri
  - bloccante



IL PACKAGE JAVA.IO: OBIETTIVI

- fornire un'astrazione che incapsuli tutti i dettagli del dispositivo sorgente/ destinazione dei dati
- fornire un modo semplice e flessibile per aggiungere ulteriori funzionalità quelle fornite dallo "stream base"
- un approccio "a livelli"
  - alcuni stream di base per connettersi a dispositivi "standard": file, connessioni di rete, console,.....
  - altri stream sono pensati per "avvolgere" i precedenti ed aggiungere ulteriori funzionalità
  - così è possibile configurare lo stream con tutte le funzionalità che servono senza doverle re-implementare più volte

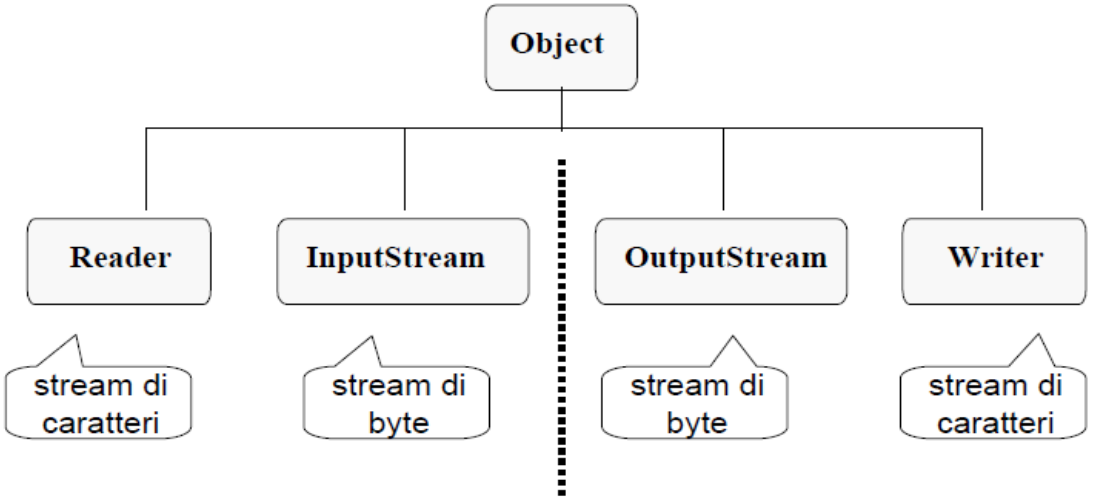
IL PACKAGE JAVA.IO: OBIETTIVI



- nell'esempio
  - stream di base è il `FileOutputStream`
  - viene "avvolto" in un `BufferedOutputStream`: byte raggruppati in blocchi, migliori prestazioni
  - viene "avvolto" in un `DataOutputStream`: trasforma tipi di dato strutturati in byte

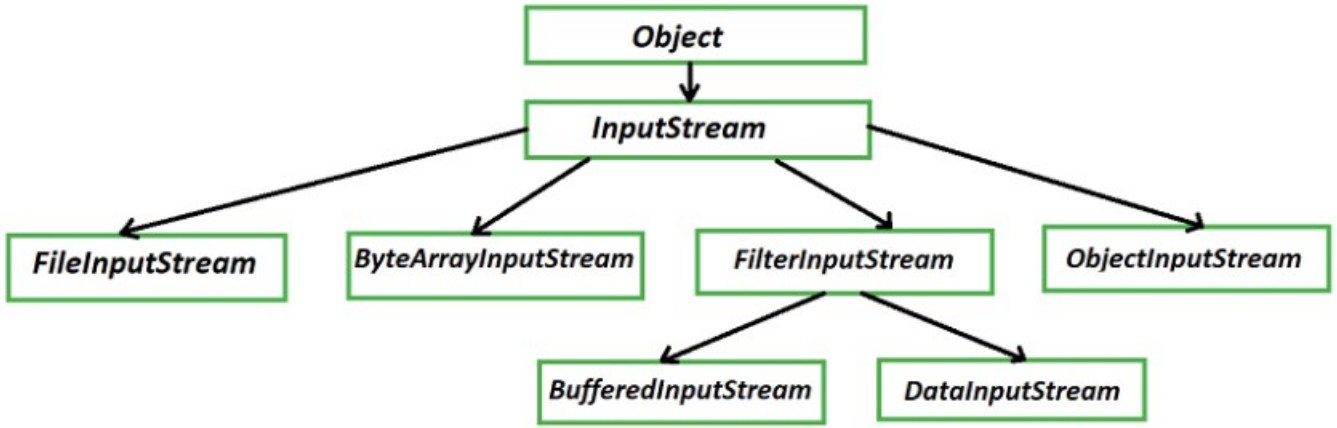
IL PACKAGE JAVA.IO

- `java.io` distingue fra:
  - stream di byte (analoghi ai file binari del C)
  - stream di caratteri (analoghi ai file di testo del C)
- modellate da altrettante classi base astratte:
  - stream di byte: `InputStream` e `OutputStream`
  - stream di caratteri: `Reader` e `Writer`
- i metodi sono simili per le due classi, per cui parleremo di stream di byte



STREAM DI BYTE

- la classe base `InputStream` definisce il concetto generale di "canale di input" che lavora a byte
  - il costruttore apre lo stream
  - `read()` legge uno o più byte
  - `close()` chiude lo stream
- `InputStream` è una classe astratta
  - il metodo `read()` dovrà essere realmente definito dalle classi derivate
  - un metodo specifico per ogni sorgente dati



STREAM DI BYTE: LEGGERE DA FILE

- `FileInputStream` è la classe derivata che rappresenta il concetto di sorgente di byte "agganciata" a un file
- il nome del file da aprire può essere passato come parametro al costruttore di `FileInputStream`

```
import java.io.*;
public class LetturaDaFileBinario {
public static void main(String args[]){
   FileInputStream is = null;
   try { is = new FileInputStream(args[0]); }
   catch(FileNotFoundException e){
        System.out.println("File non trovato");
        System.exit(1);
         }
```

- in alternativa si può passare al costruttore un oggetto `File` (o un `FileDescriptor`) costruito in precedenza

STREAM DI BYTE: LEGGERE DA FILE

- si usa il metodo `read()`
  - permette di leggere uno o più byte dal file
  - restituisce il byte letto come intero fra `0` e `255`
  - se lo stream è finito, restituisce `-1`
  - se non ci sono byte, ma lo stream non è finito, rimane in attesa dell'arrivo di un byte

```
try { int x; int n = 0;
      while ((x = is.read())>=0) {
         System.out.println(" " + x); n++;
        }
      System.out.println("\nTotale byte: " + n);
     }
catch(IOException ex){
       System.out.println("Errore di input");
       System.exit(2);}}}
```

STREAM DI BYTE: SCRIVERE SU FILE

- metodi analoghi per la apertura/scrittura su file
- `FileOutputStream` è la classe derivata che rappresenta il concetto di dispositivo di uscita "agganciato" a un file
- il nome del file da aprire è passato come parametro al costruttore di `FileOutputStream`, o in alternativa si può passare al costruttore un oggetto `File` costruito in precedenza
- per scrivere sul file si usa il metodo `write()` che permette di scrivere uno o più byte
  - scrive l'intero (`0 - 255`) passatogli come parametro
  - non restituisce nulla

JAVA: FILTER STREAMS

- `FilterInputStream` and `FilterOutputStream` con diverse sottoclassi
  - `BufferedInputStream` e `BufferedOutputStream` implementano filtri che bufferizzano l'input da/l'output verso lo stream sottostante
    - i dati vengono scritti e letti in blocchi di bytes, invece che un solo blocco per volta miglioramento significativo della performance
  - `DataInputStream` and `DataOutputStream` implementano filtri che permettono di " formattare" i dati presenti sullo stream

COPYING A FILE .JPEG

```
import java.io.*;
public class FileCopyNoBuffer{
   public static void main(String[] args) {
      String inFileStr  = "relax.jpg"; String outFileStr = "relax_new.jpg";
      long startTime, elapsedTime;  // for speed benchmarking
      File fileIn = new File(inFileStr);
      System.out.println("File size is " + fileIn.length() + " bytes");
      FileInputStream  in;  FileOutputStream out;
      try
          { in  = new FileInputStream(inFileStr);
            out = new FileOutputStream(outFileStr);
            startTime = System.nanoTime();
            int byteRead;
            while ((byteRead = in.read()) != -1)
                  { out.write(byteRead);}
            elapsedTime = System.nanoTime() - startTime;
            System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + "msec");
       }  catch (IOException ex) { ex.printStackTrace(); }}}
```

```
File size is 16473 bytes
Elapsed Time is 54.2873 msec
```

JAVA: FILTER STREAMS

cosa accade sostituendo

```
FileInputStream  in  = new FileInputStream(inFileStr);
FileOutputStream out = new FileOutputStream(outFileStr)
```

con

```
BufferedInputStream in = new BufferedInputStream(new
                                 FileInputStream(inFileStr));
BufferedOutputStream out = new BufferedOutputStream(new
                                 FileOutputStream(outFileStr)))
```

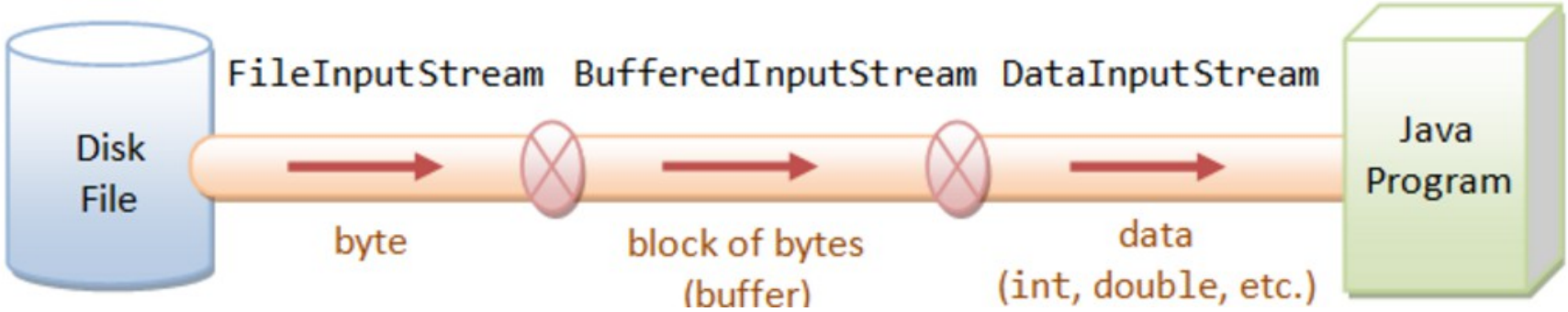
?

i tempi di esecuzione del programma si abbassano notevolmente

```
File size is 16473 bytes
Elapsed Time is 1.2581 msec
```

JAVA: FORMATTED DATA STERAM

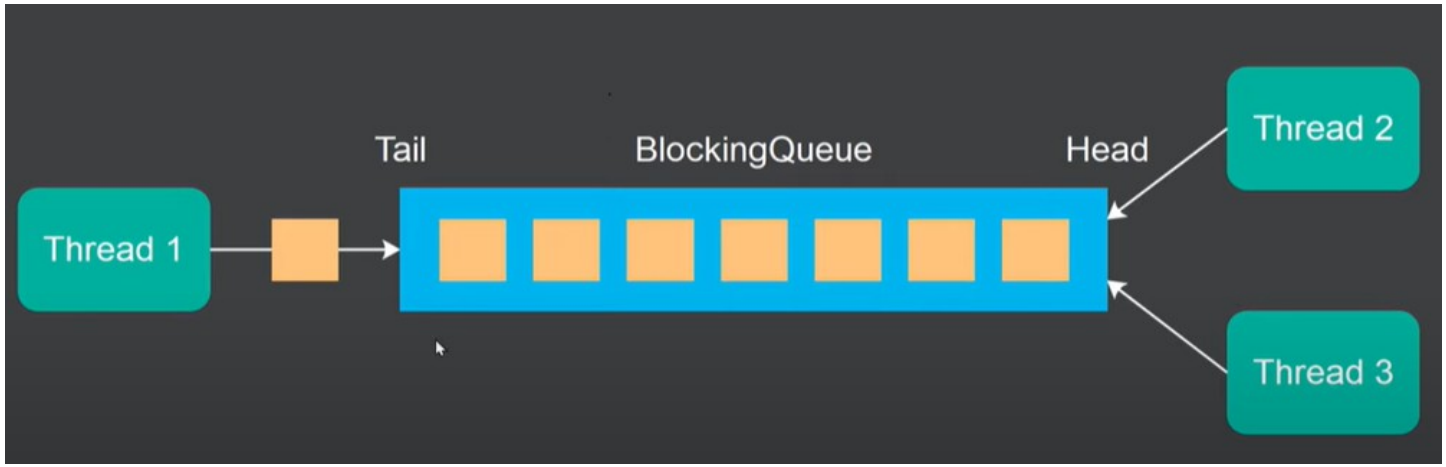
```
mport java.io.*;
public class TestDataIOStream {
   public static void main(String[] args) {
      String filename = "data-out.dat";
      // Write primitives to an output file
      try (DataInputStream in =
              new DataInputStream(
                 new BufferedInputStream(
                    new FileInputStream(filename)))) {
```



JAVA: FORMATTED DATA STERAM

```
mport java.io.*;
public class TestDataIOStream {
   public static void main(String[] args) {
      String filename = "data-out.dat";
      // Write primitives to an output file
      try (DataInputStream in =
              new DataInputStream(
                 new BufferedInputStream(
                    new FileInputStream(filename)))) {
            System.out.println("byte:    " + in.readByte());
            System.out.println("short:   " + in.readShort());
            System.out.println("int:     " + in.readInt());
            System.out.println("long:    " + in.readLong());
            System.out.println("float:   " + in.readFloat());
            System.out.println("double:  " + in.readDouble());
            System.out.println("boolean: " + in.readBoolean());...}
```

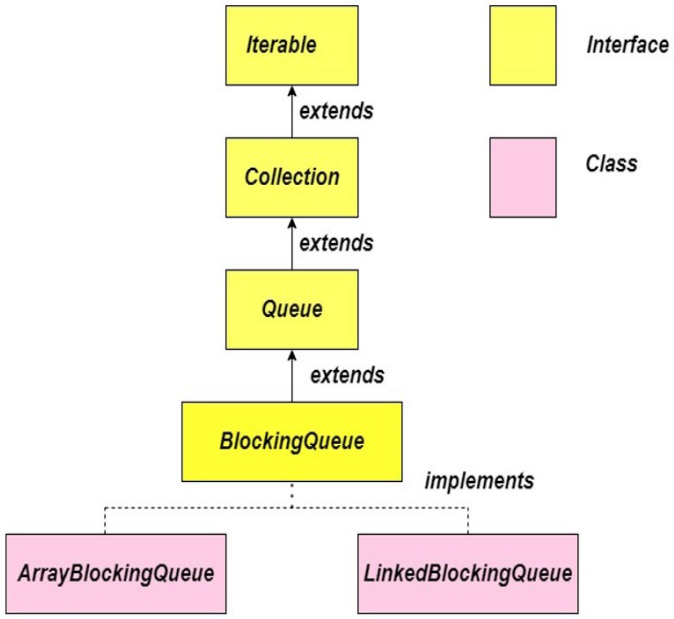
JAVA BLOCKING QUEUE



- `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

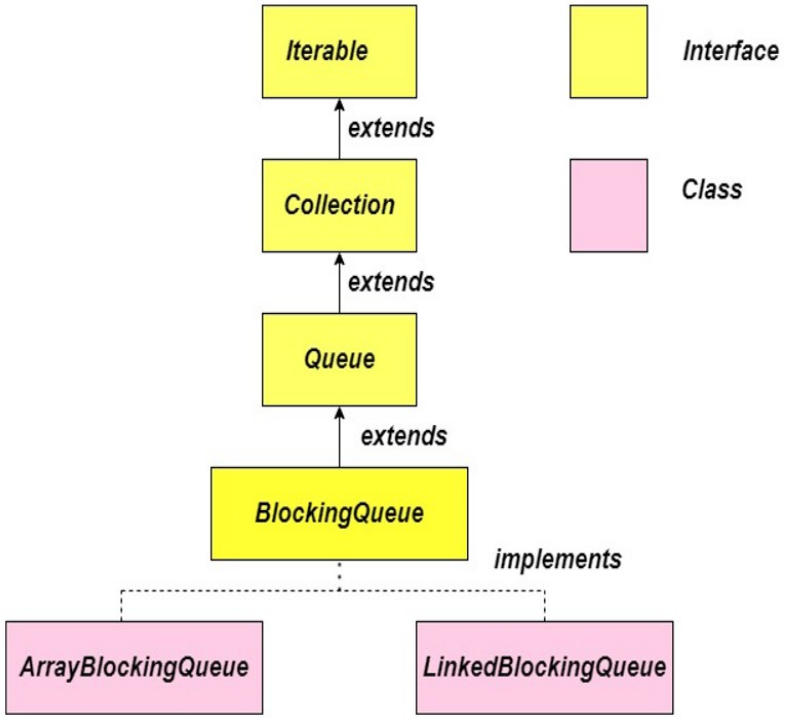
```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

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
             }}
```

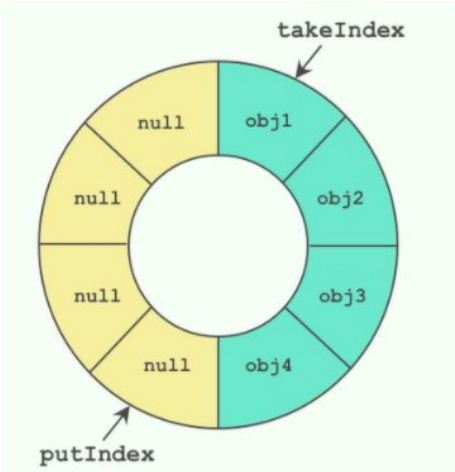


QUALI CODE UTILIZZEREMO MAGGIORMENTE?

`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)



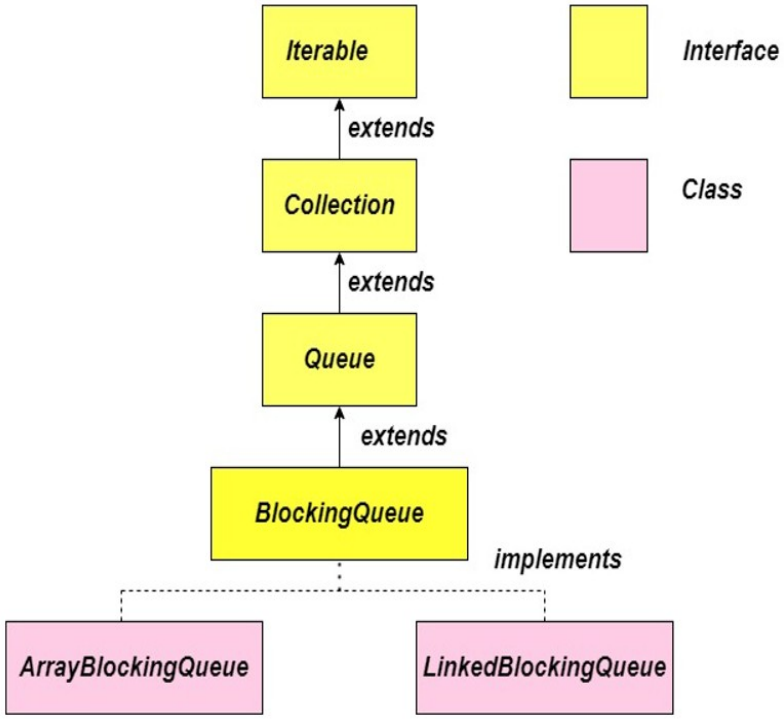


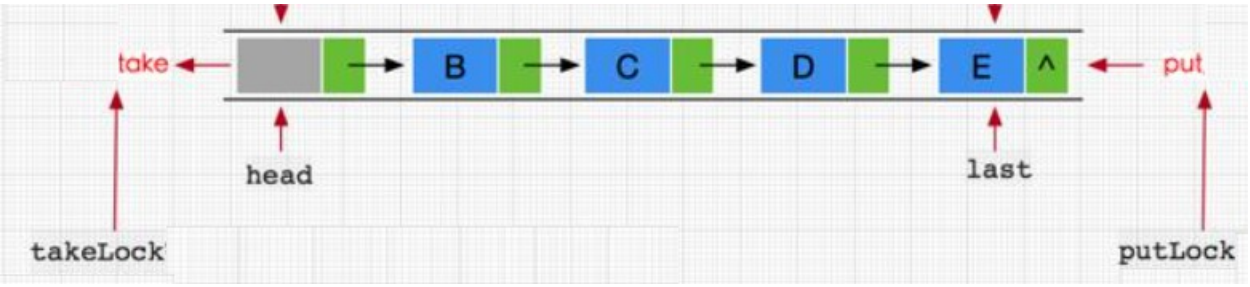
E QUALI CODE UTILIZZEREMO MAGGIORMENTE?

`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

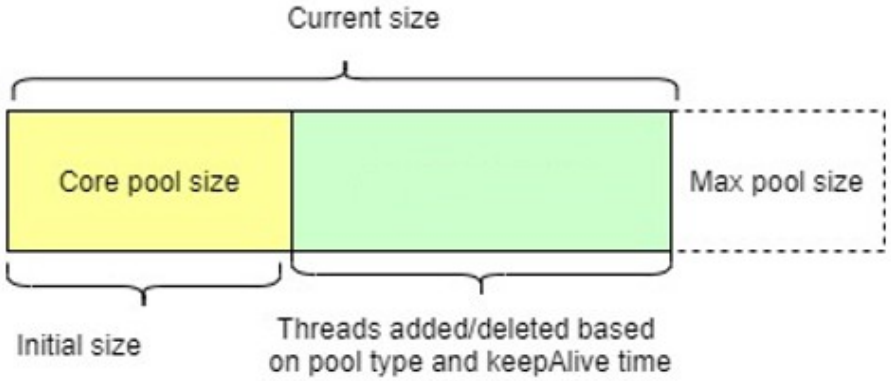
	Throws Exception	Special Value	Blocks	Times Out
Insert	`add(o)`	`offer(o)`	`put(o)`	`offer(o, timeout, timeunit)`
Remove	`remove(o)`	`poll()`	`take()`	`poll(timeout, timeunit)`
Examine	`element()`	`peek()`		

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
  - 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

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*

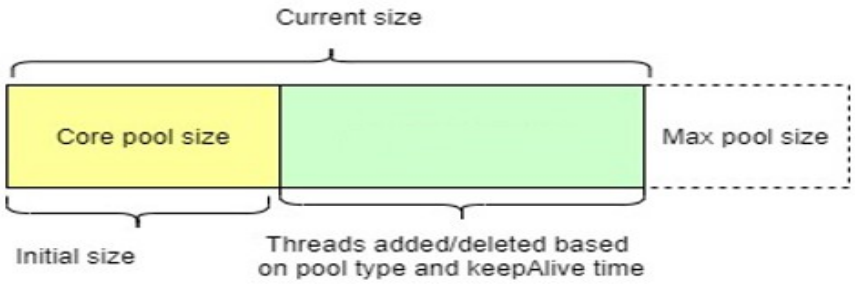
THREAD POOL EXECUTOR: RIASSUNTO

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 , 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 è 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

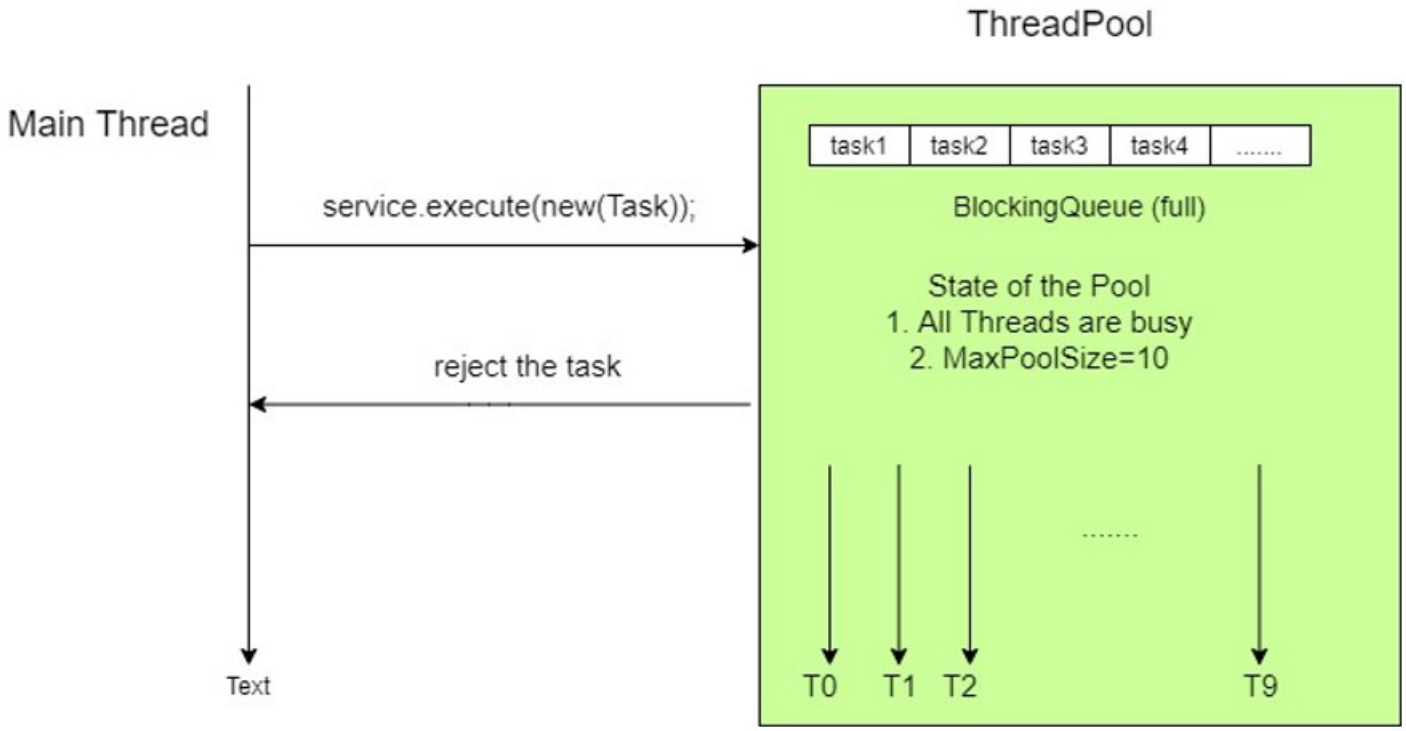
POOL	QUEUE TYPE	WHY?
FixedThreadPool	LinkedBlockingQueue	Threads are limited, thus unbounded queue to store tasks Note: since queue can never become full, new threads are never created
CachedThreadPool	SynchronousQueue	Threads are unbounded, thus no need to store the tasks. Create the new thread and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

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.