

# Reti e Laboratorio III

## Modulo Laboratorio III

### AA. 2022-2023

docente: **Laura Ricci**

**[laura.ricci@unipi.it](mailto:laura.ricci@unipi.it)**

## Lezione I

### **JAVA multithreading: threads e thread pooling**

**15/9/2022**

# INFORMAZIONI GENERALI

- **Docente**
  - *Laura Ricci* ([laura.ricci@unipi.it](mailto:laura.ricci@unipi.it)),
- **Supporto alla Didattica**
  - *Matteo Loporchio*
- **Crediti:** 6 CFU Reti (Prof.Paganelli), 3 CFU Laboratorio III (Prof. Laura Ricci)
- **Orario Modulo Laboratorio**

giovedì 9.00 - 11.00 – AULA E, BYOD (*Bring Your Own Device*)
- **Orario ricevimento**

giovedì 15:00 – 17:00 Dipartimento Informatica e online su Teams
- **Materiale su Moodle:** <https://elearning.di.unipi.it/course/view.php?id=311>
  - slides
  - forum
  - quiz
  - assignments



# INFORMAZIONI GENERALI

- parte teorica: presentazione dei concetti di base
- parte pratica
  - verifica esercizi assegnati nelle lezioni teoriche
  - quiz anonimi a risposta chiusa per l'autoverifica
- consegna **facoltativa** degli assignment entro 15 giorni dalla data di assegnazione.
- lo studente che consegna l'80% degli esercizi, li discuterà all'esame e, se la discussione è positiva, potrà ottenere un bonus di 2 punti sul voto finale



# MODALITA' DI ESAME

- l'esame di Reti e Laboratorio si svolge in due prove:
  - prova di Reti (Teoria)
  - prova di Laboratorio
- non ci sono vincoli di precedenza tra la prova di Reti e quella di Laboratorio.
- il voto di ciascuna prova ha validità per l'AA 2022/23 (entro l'appello straordinario di ottobre/novembre 2023 compreso per chi ha i requisiti per partecipare all'appello).
- **voto finale:**
  - media pesata (2/3 reti, 1/3 laboratorio) dei voti ottenuti nelle due prove e arrotondamento per eccesso.
  - chi non si iscrive entro i termini non può partecipare alla prova di esame.  
**Attenzione alle scadenze!**



# MODALITA' DI ESAME

- Tutte le prove d'esame prevedono obbligatoriamente l'iscrizione sul **SISTEMA DI ISCRIZIONE DI ATENEO**
  - chi non si iscrive entro i termini non può partecipare alla prova di esame
  - attenzione alle scadenze!!!
- **Prova di Laboratorio**
  - lo studente deve consegnare un progetto, da svolgere secondo le specifiche consegnate durante il corso (entro la prima metà di dicembre).
  - le specifiche del progetto sono valide fino all'appello straordinario di novembre 2023 (a questo appello può accedere solo chi ha i requisiti).
  - la prova consiste in un colloquio orale che include la discussione del progetto e verifica dell'apprendimento dei concetti e contenuti presentati a lezione.
  - il progetto deve essere svolto **individualmente**



# MODALITA' DI ESAME STUDENTI ISCRITTI ANNI PRECEDENTI

- reti di calcolatori e laboratorio (12 CFU)
- laboratorio (6 CFU)
  - programma AA 2021/22
  - progetto + discussione su contenuti programma
  - specifiche del Progetto ≠ progetto nuovo regolamento
  - le specifiche del progetto sono valide fino all'appello di settembre 2023 (incluso l'appello straordinario di novembre 2023 per chi ha i requisiti per accedere
- Chi ha già sostenuto una delle due prove parziali (i.e. prova di Reti o prova di Laboratorio) mantiene il voto parziale per l'AA 2022/23 (entro l'appello straordinario di ottobre/novembre 2023 compreso per chi ha i requisiti per partecipare all'appello)



# INFORMAZIONI UTILI: PREREQUISITI

- corso di Programmazione 2, conoscenza del linguaggio JAVA.
- dal modulo teorico di reti
  - conoscenza protocollo TCP/IP
- linguaggio di programmazione di riferimento: anche se l'ultima release è la 16, facciamo riferimento a JAVA 8
  - concorrenza: costrutti base, `JAVA.UTIL.CONCURRENT`, concurrent collections
  - `JAVA.NIO`
  - `JAVA.NET`
- ambiente di sviluppo di riferimento: Eclipse



# INFORMAZIONI UTILI

- Materiale Didattico:
  - lucidi delle lezioni
  - testi consigliati (non obbligatori) per la parte relativa ai threads
    - *Bruce Eckel, Thinking in JAVA - Volume 3 - Concorrenza e Interfacce Grafiche*
    - *B. Goetz, JAVA Concurrency in Practice, 2006*
  - Testi consigliati (non obbligatori) per la parte relativa alla programmazione di rete
    - *Dario Maggiorini, Introduzione alla Programmazione Client Server, Pearson*
    - *Esmond Pitt, Fundamental Networking in JAVA*
- Materiale di Consultazione:
  - *Harold, JAVA Network Programming 3nd edition O'Reilly 2004.*
  - *K.Calvert, M.Donhaoo, TCP/IP Sockets in JAVA, Practical Guide for Programmers*
  - Costrutti di base: Horstmann , *Concetti di Informatica e Fondamenti di Java 2*



# PROGRAMMA PRELIMINARE DEL CORSO

## Threads

- meccanismi di gestione di pools di threads, Callable
- richiami su primitive di sincronizzazione (lock)
- concurrent collections

## Stream-based IO

- richiami su streams: tipi di streams, composizione di streams
- meccanismi di serializzazione: JSON libreria GSON

## NewIO

- Channels, buffers, memory mapped IO

## Programmazione di rete

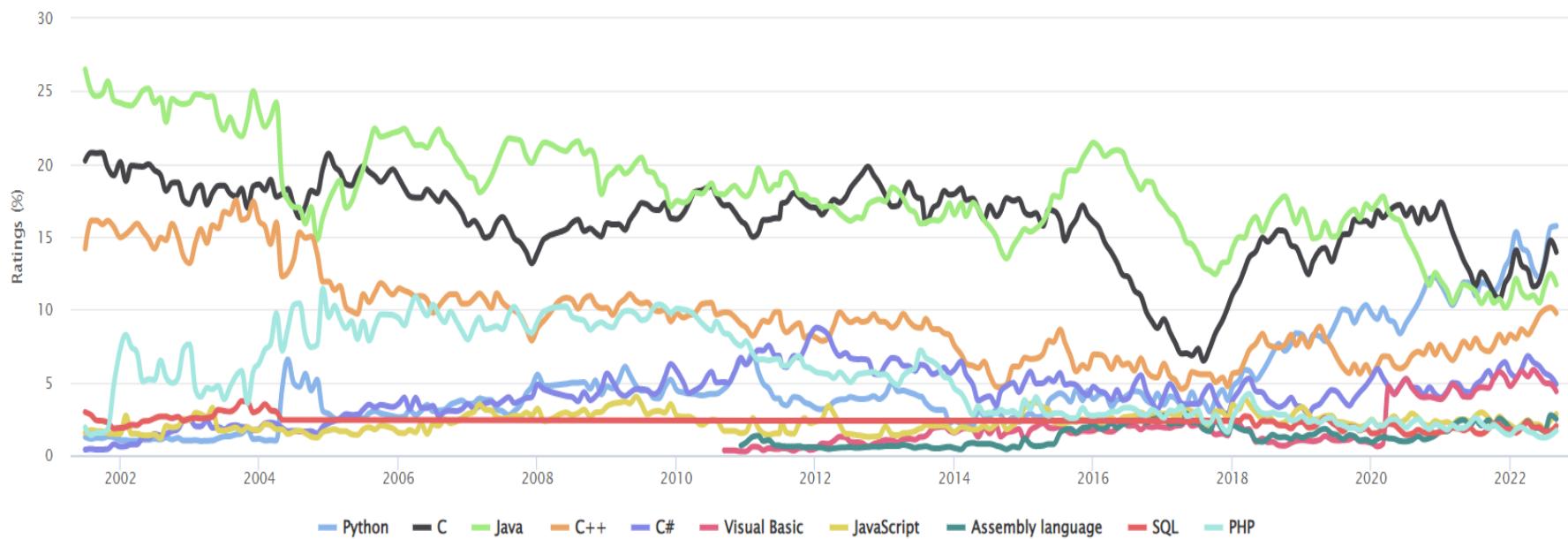
- connection oriented Sockets
- connectionless sockets: UDP, multicast
- NewIO e sockets
- Selector: channel multiplexing



# L'INDICE TIOBE DEI LINGUAGGI DI PROGRAMMAZIONE

TIOBE Programming Community Index

Source: [www.tiobe.com](http://www.tiobe.com)



- misura la popolarità dei linguaggi di programmazione in funzione del numero di ricerche contenenti il nome del linguaggio
- JAVA uno dei top-3 linguaggi: utile studiarlo!

# L'EVOLUZIONE DI JAVA



**23 JAN 1996 - JAVA 1**

First public release. The stable version Java 1.0.2 is called Java 1.

**1995 - JDK BETA**

The first beta version of Java. Developed by James Gosling at Sun Microsystems.

**8 DEC 1998 - JAVA 1.2**

Swing, JIT Compiler, Collections

**19 FEB 1997 - JAVA 1.1**

Inner Classes, Java Beans, JDBC, RMI

**6 FEB 2002 - JAVA 1.4**

Assertions, RegEx Improvements, Image IO API, XML Parsers, XSLT Processors, Preferences API

**8 MAY 2000 - JAVA 1.3**

HotSpot JVM, JNDI, JPDA

**30 SEP 2004 - JAVA 5**

Generics API, Varargs, for-each loop, Autoboxing, Enum, Annotations, Static Imports



# L'EVOLUZIONE DI JAVA

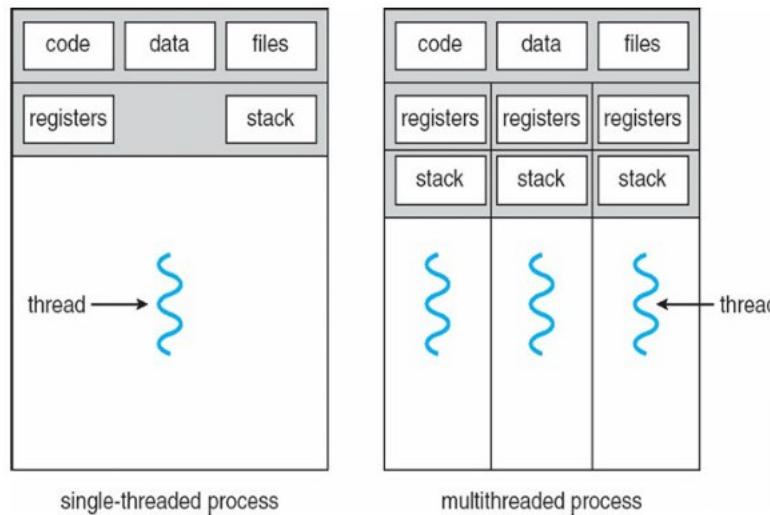


Ultima versione JAVA 18  
In questo corso faremo  
riferimento a JAVA8



# THREAD: RICHIAMI

- processo: programma in esecuzione
  - due diverse applicazioni, ad esempio MS Word, MS Access, sono eseguite da processi diversi.
- thread (light weight process): un flusso di esecuzione all'interno di un processo



- multitasking, si può riferire a thread o processi
  - a livello di processo è controllato esclusivamente dal sistema operativo
  - a livello di thread è controllato, almeno in parte, dal programmatore



# PROCESSI E THREADS: RICHIAMI

- thread multitasking verso process multitasking:
  - i thread condividono lo **stesso spazio degli indirizzi**
  - meno costosi
    - il cambiamento di contesto tra thread
    - la comunicazione tra thread
- esecuzione dei thread:
  - single core: multiplexing, interleaving (meccanismi di time sharing,...)
  - multicore: più flussi in esecuzione eseguiti in parallelo, simultaneità di esecuzione



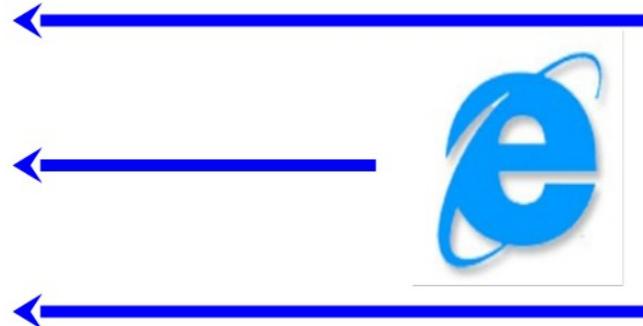
# MULTITHREADING: PERCHE'?

- migliore utilizzazione delle risorse
  - quando un thread è sospeso, altri thread vengono mandati in esecuzione
  - riduzione del tempo complessivo di esecuzione
- migliore performance per applicazioni computationally intensive
  - dividere l'applicazione in task ed eseguirli in parallelo
- tanti vantaggi, ma anche alcuni problemi:
  - più difficile il debugging e la manutenzione del software rispetto ad un programma single threaded
  - race conditions, sincronizzazioni
  - deadlock, livelock, starvation,...



# THREAD E PROGRAMMAZIONE DI RETE

- applicazioni client server
  - più client serviti simultaneamente
  - un client non deve aspettare che il server termini di elaborare la richiesta del client precedente

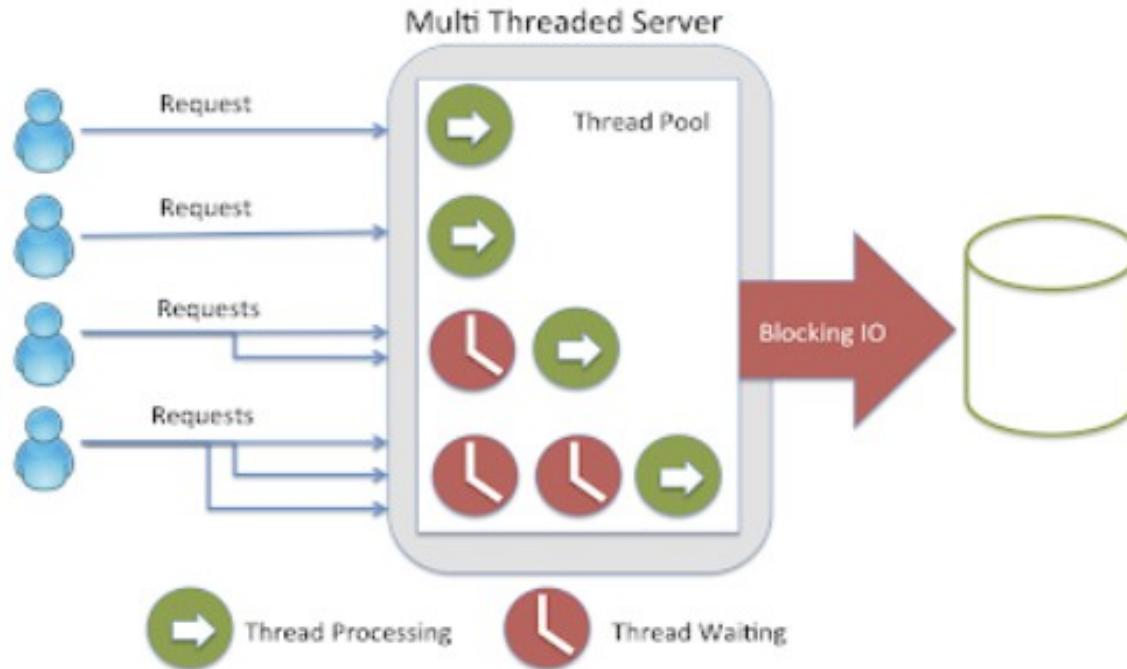


Web Server uses  
threads to handle ...

Multiple simultaneous  
web browser requests



# THREAD E PROGRAMMAZIONE DI RETE



- il throughput dell'applicazione può essere incrementato se client diversi sono serviti da thread diversi, ma solo fino ad un certo limite
- oltre quel limite, i thread iniziano a competere per la CPU e il costo del cambio di contesto supera il beneficio del multithreading
- limitare questo fenomeno con il meccanismo del **threadpooling**



# JAVA UTIL.CONCURRENT FRAMEWORK

- JAVA < 5 built in for concurrency: lock implicite, wait, notify e poco più.
- **JAVA.util.concurrency**
  - lo stesso scopo del framework `java.util.Collections`
  - un toolkit general purpose per lo sviluppo di applicazioni concorrenti.  
*no more “reinventing the wheel”!*
- definire un insieme di utility che risultino:
  - standardizzate
  - facili da utilizzare e da capire
  - high performance
  - utili in un grande insieme di applicazioni per un vasto insieme di programmatore, da quelli più esperti a quelli meno esperti.



# JAVA UTIL.CONCURRENT FRAMEWORK

- sviluppato in parte da Doug Lea, disponibile, come insieme di librerie JAVA non standard prima della integrazione in JAVA 5.0.
- tra i package principali:
  - `java.util.concurrent`
    - executor, concurrent collections, semaphores,...
  - `java.util.concurrent.atomic`
    - AtomicBoolean, AtomicInteger,...
  - `java.util.concurrent.locks`
    - Condition
    - Lock
    - ReadWriteLock



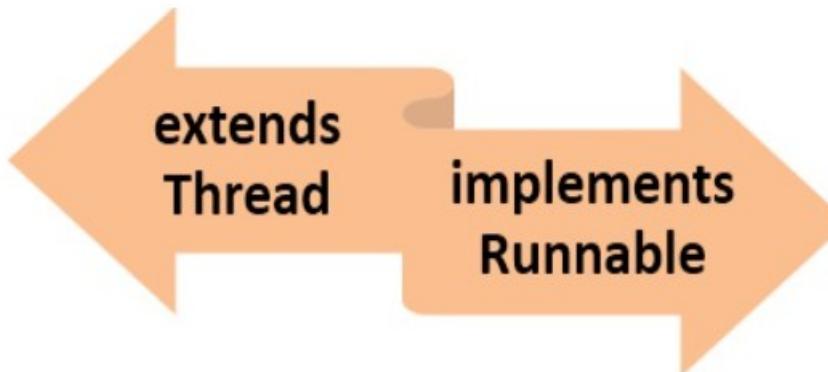
# JAVA 5 CONCURRENCY FRAMEWORK

- Executors
  - Executor
  - ExecutorService
  - ScheduledExecutorService
  - Callable
  - Future
  - ScheduledFuture
  - Delayed
  - CompletionService
  - ThreadPoolExecutor
  - ScheduledThreadPoolExecutor
  - AbstractExecutorService
  - Executors
  - FutureTask
  - ExecutorCompletionService
- Queues
  - BlockingQueue
  - ConcurrentLinkedQueue
  - LinkedBlockingQueue
  - ArrayBlockingQueue
  - SynchronousQueue
  - PriorityBlockingQueue
  - DelayQueue
- Concurrent Collections
  - ConcurrentHashMap
  - CopyOnWriteArrayList
  - CopyOnWriteArraySet
- Synchronizers
  - CountDownLatch
  - Semaphore
  - Exchanger
  - CyclicBarrier
- Locks: `java.util.concurrent.locks`
  - Lock
  - Condition
  - ReadWriteLock
  - AbstractQueuedSynchronizer
  - LockSupport
  - ReentrantLock
  - ReentrantReadWriteLock
- Atomics: `java.util.concurrent.atomic`
  - Atomic[Type]
  - Atomic[Type]Array
  - Atomic[Type]FieldUpdater
  - Atomic{Markable,Stampable}Reference



# JAVA: CREAZIONE ED ATTIVAZIONE DI THREAD

- quando si manda in esecuzione un programma JAVA
  - la JVM crea un thread che invoca il metodo main del programma
  - quindi esiste sempre almeno un thread per ogni programma, il main
- in seguito...
  - altri thread sono attivati automaticamente da JAVA (gestore eventi, interfaccia, garbage collector,...).
  - ogni thread durante la sua esecuzione può creare ed attivare altri threads.
- due modalità per creare ed attivare esplicitamente un thread





- definire un task
  - creare un oggetto thread e passargli il task definito che contiene il codice da eseguire
  - attivare il thread con una start()
- per definire un task
- definire una classe che implementi l'interfaccia Runnable
  - creare un'istanza R di questa classe, Questo è il task da passare al thread

# CREAZIONE-ATTIVAZIONE DI THREAD: SOLUZIONE I

```
public class ThreadRunnable {  
    public static class MyRunnable implements Runnable {  
        public void run() {  
            System.out.println("MyRunnable running");  
            System.out.println("MyRunnable finished");  
        }  
    }  
  
    public static void main(String [] args) {  
        Thread thread = new Thread (new MyRunnable());  
        thread.start();  
    }  
}
```

*Stampa:*  
*MyRunnable running*  
*MyRunnable finished*



# L'INTERFACCIA RUNNABLE

- appartiene al package `java.lang`
- contiene solo la segnatura del metodo `void run()`, che deve essere implementato
- un'istanza della classe che implementa Runnable è un **task**
  - un **fragmento di codice** che può essere eseguito in un thread
  - la creazione del task non implica la creazione di un thread per lo esegua.
  - lo stesso task può essere eseguito da più threads: un solo codice, più esecutori
  - il task viene passato al Thread che deve eseguirlo



# TASK DEFINITO CON CLASSE ANONIMA

```
public class RunnableAnonymous {  
    public static void main (String[] args) {  
        Runnable runnable = new Runnable () {  
            public void run() {  
                System.out.println("Runnable running");  
                System.out.println("Runnable finished");  
            }  
        };  
    };
```

```
    Thread thread = new Thread (runnable);  
    thread.start();  
}  
}
```

*Stampa:*

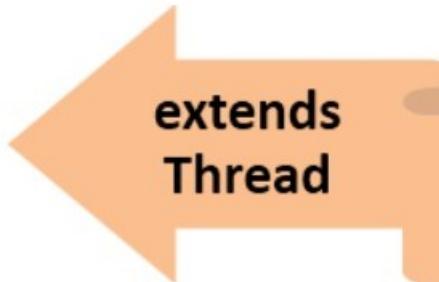
*Runnable running*

*Runnable finished*



# CREAZIONE-ATTIVAZIONE DI THREAD: SOLUZIONE 2

- creare una classe C che estenda Thread
- effettuare l'*overriding* del metodo `run()`
- istanziare un oggetto di quella classe
  - questo oggetto è un thread il cui comportamento è quello definito nel metodo `run` ridefinito
- invocare il metodo `start()` sull'oggetto istanziato.



Overriding:

- metodo in una sottoclasse con lo stesso nome e segnatura del metodo della superclasse
- decisione a run-time su quale metodo viene invocare in base all'istanza su cui si invoca il metodo



# CREAZIONE-ATTIVAZIONE DI THREAD: SOLUZIONE 2

```
public class ExtendingThread {  
  
    public static class MyThread extends Thread {  
        public void run() {  
            System.out.println("MyThread running");  
            System.out.println("MyThread finished");  
        }  
    }  
  
    public static void main (String [] args) {  
        MyThread myThread = new MyThread();  
        myThread.start();  
    }  
}
```

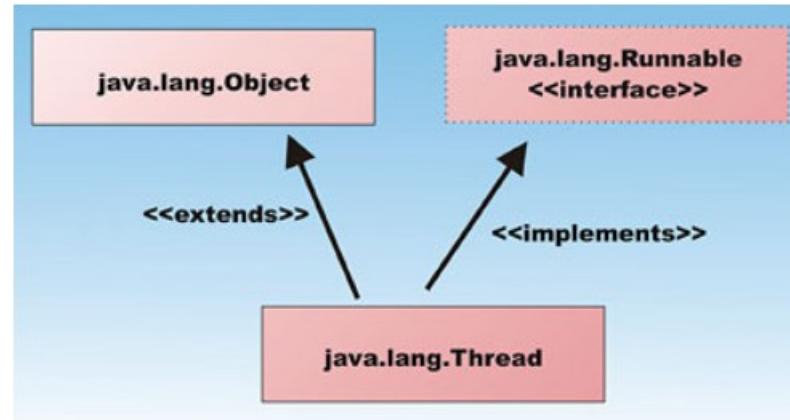
*Stampa*

*MyThread running*

*MyThread finished*



# LA CLASSE THREAD

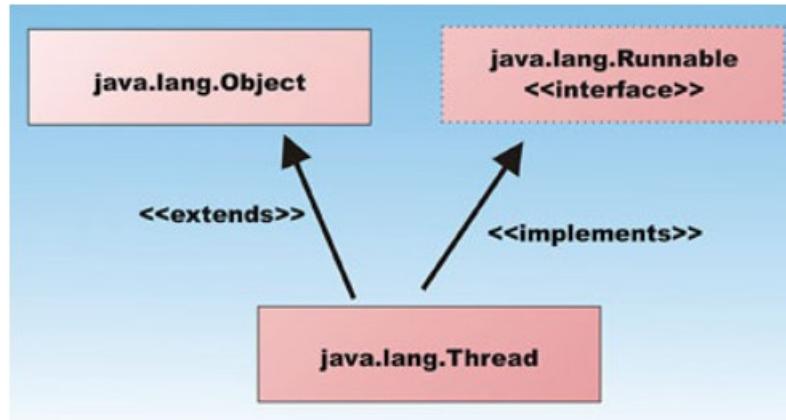


- memorizza un riferimento all'oggetto `Runnable`, eventualmente passato come parametro, nella variabile `runnable`
- definisce il metodo `run( )` come segue

```
public void run( )
{ if (runnable != null)
  runnable.run( ); }
```



# LA CLASSE THREAD



- quando viene invocata la `start()`  
se il metodo `run()` è stato ridefinito mediante overriding (soluzione 2)  
si invoca il metodo `run()` più specifico, che è quello definito dal programmatore
- altrimenti, si esegue il metodo `run()` predefinito nella classe `Thread`, (soluzione 1)
  - se la variable `Runnable` è diversa da nil, questo metodo, a sua volta, invoca il metodo `run()` dell'oggetto `Runnable` passato
  - si esegue il metodo definito dal programmatore

# ATTIVARE UN INSIEME DI THREAD

- scrivere un programma che stampi le tabelline moltiplicative dall' 1 al 10
  - si attivino 10 threads
  - ogni numero  $n$ ,  $1 \leq n \leq 10$ , viene passato ad un thread diverso
  - il task assegnato ad ogni thread consiste nello stampare la tabellina corrispondente al numero che gli è stato passato come parametro



# IL TASK CALCULATOR

```
public class Calculator implements Runnable {  
    private int number;  
    public Calculator(int number) {  
        this.number=number; }  
    public void run() {  
        for (int i=1; i<=10; i++){  
            System.out.printf("%s: %d * %d = %d\n",  
                Thread.currentThread().getName(), number, i, i*number);  
        } } }
```

- NOTA: **public static native** Thread *currentThread* ( ):

- più thread potranno eseguire il codice di Calculator
- qual'è il thread che sta eseguendo attualmente questo codice?

*currentThread( )* restituisce un riferimento al thread che sta eseguendo il  
fragmento di codice



# IL MAIN PROGRAM

```
public class Main {  
    public static void main(String[] args) {  
        for (int i=1; i<=10; i++){  
            Calculator calculator=new Calculator(i);  
            Thread thread=new Thread(calculator);  
            thread.start();}  
            System.out.println("Avviato Calcolo Tabelline"); } }
```

L'output Generato dipende dalla schedulazione effettuata, un esempio è il seguente:

```
Thread-0: 1 * 1 = 1  
Thread-9: 10 * 1 = 10  
Thread-5: 6 * 1 = 6  
Thread-8: 9 * 1 = 9  
Thread-7: 8 * 1 = 8  
Thread-6: 7 * 1 = 7  
Avviato Calcolo Tabelline  
Thread-4: 5 * 1 = 5  
Thread-2: 3 * 1 = 3
```



# ALCUNE OSSERVAZIONI

- Output generato (dipendere comunque dallo scheduler):

Thread-0:  $1 * 1 = 1$

Thread-9:  $10 * 1 = 10$

Thread-5:  $6 * 1 = 6$

Thread-8:  $9 * 1 = 9$

Thread-7:  $8 * 1 = 8$

Thread-6:  $7 * 1 = 7$

**Avviato Calcolo Tabelline**

Thread-4:  $5 * 1 = 5$

Thread-2:  $3 * 1 = 3$

- da notare: il messaggio **Avviato Calcolo Tabelline** è stato visualizzato prima che tutti i threads completino la loro esecuzione. Perchè?
  - il controllo ripassa al programma principale, dopo la attivazione dei threads e prima della loro terminazione.



# START( ) E RUN()

```
public class Main {  
    public static void main(String[] args) {  
        for (int i=1; i<=10; i++){  
            Calculator calculator=new Calculator(i);  
            Thread thread=new Thread(calculator);  
            thread.run(); // questa versione del programma è errata  
            System.out.println("Avviato Calcolo Tabelline") }  
    }  
}
```

Output generato

main: 1 \* 1 = 1

main: 1 \* 2 = 2

main: 1 \* 3 = 3

.....

main: 2 \* 1 = 2

main: 2 \* 2 = 4

.....

Avviato Calcolo Tabelline



# START E RUN

cosa accade se sostituisco l'invocazione del metodo run alla start?

- non viene attivato alcun thread
- ogni metodo run() viene eseguito all'interno del flusso del thread attivato per l'esecuzione del programma principale
- flusso di esecuzione sequenziale
- il messaggio “Avviato Calcolo Tabelline” viene visualizzato dopo l'esecuzione di tutti i metodi metodo run() quando il controllo torna al programma principale
- solo il metodo start() comporta la creazione di un nuovo thread()!



# IL METODO START

- segnala allo schedulatore (tramite la JVM) che il thread può essere attivato (invoca un metodo nativo)
- l'ambiente del thread viene inizializzato.
- restituisce immediatamente il controllo al chiamante, senza attendere che il thread attivato inizi la sua esecuzione.
  - la stampa del messaggio “Avviato Calcolo Tabelline” precede quelle effettuate dai threads.
  - questo significa che il controllo è stato restituito al thread chiamante (il thread associato al main) prima che sia iniziata l'esecuzione dei threads attivati



# TASK CALCULATOR CON METODO 2

```
public class Calculator extends Thread {  
    .....  
    public void run() {  
        for (int i=1; i<=10; i++)  
            {System.out.printf("%s: %d * %d = %d\n",  
                Thread.currentThread().getName(), number, i, i*number);}}}  
  
public class Main {  
    public static void main(String[] args) {  
        for (int i=1; i<=10; i++){  
            Calculator calculator=new Calculator(i);  
            calculator.start();}  
            System.out.println("Avviato Calcolo Tabelline"); } }
```



# QUALE ALTERNATIVA UTILIZZARE?

- in JAVA una classe può estendere una solo altra classe (**eredità singola**)
  - se si estende la classe Thread, la classe i cui oggetti devono essere eseguiti come thread non può estendere altre classi.
- questo può risultare svantaggioso in diverse situazioni, ad esempio:
  - gestione di eventi dell'interfaccia (movimento mouse, tastiera...)
    - la classe che gestisce un evento deve estendere una classe C predefinita di JAVA
    - se il gestore deve essere eseguito in un thread separato, occorrerebbe definire una classe che estenda sia C che Thread, ma questo non è permesso in JAVA, occorrerebbe l'ereditarietà multipla
  - si definisce allora una classe che :
    - estenda C (non può estendere contemporaneamente Thread)
    - implementi la interfaccia Runnable



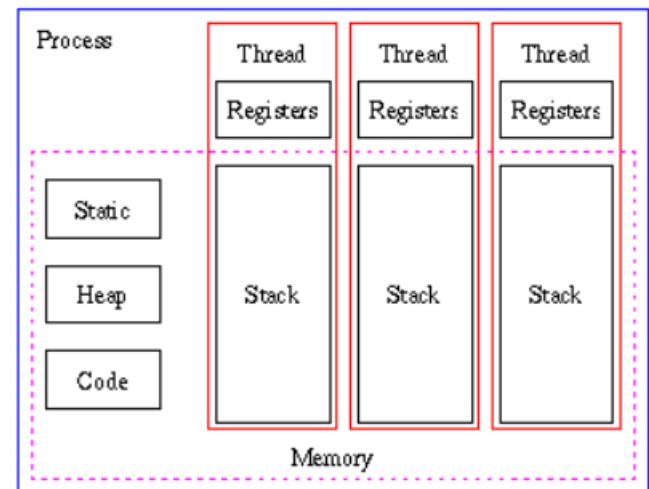
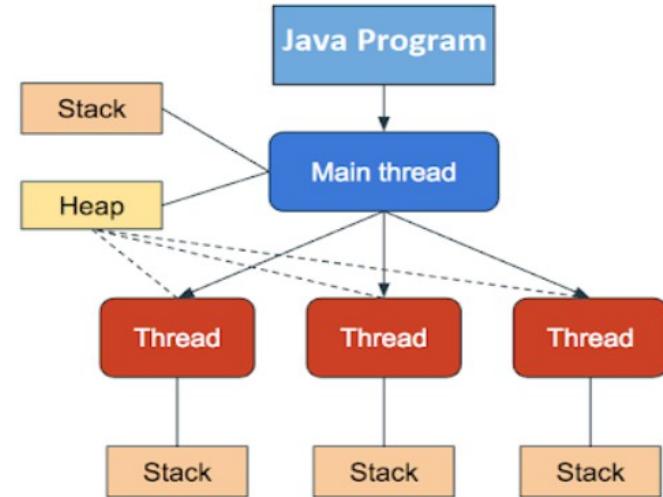
# TERMINAZIONE DI PROGRAMMI CONCORRENTI

- un programma JAVA termina quando terminano tutti i threads **non demoni** che lo compongono
- se il thread iniziale, cioè quello che esegue il metodo `main()` termina, i restanti thread ancora attivi e non demoni continuano la loro esecuzione, il programma termina quando anche questi terminano.
  - il “quadratino” rosso di Eclipse rimane “rosso” anche se il main è terminato
- se uno dei thread usa l'istruzione `System.exit()` per terminare l'esecuzione, allora tutti i threads terminano la loro esecuzione

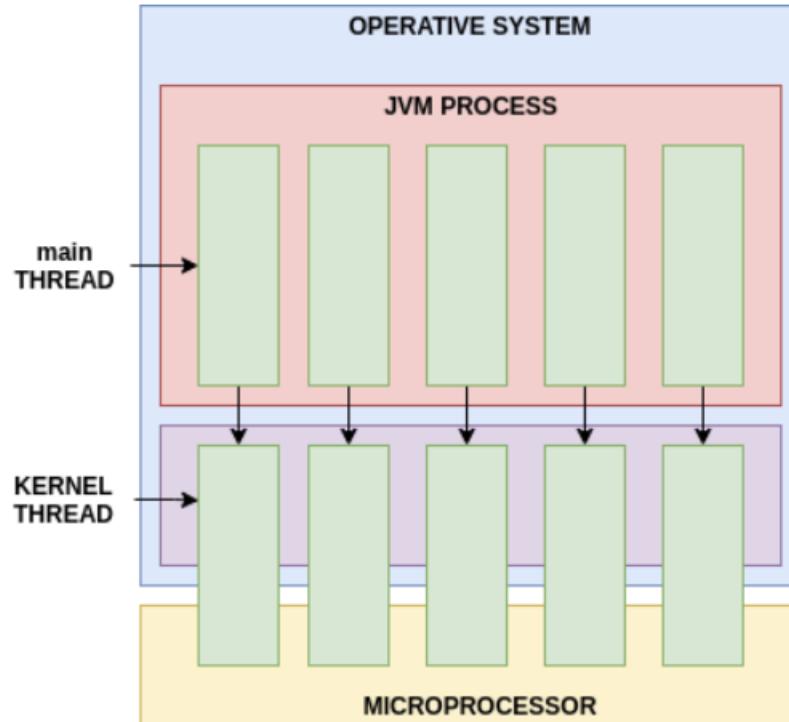


# THREAD OVERHEAD

- 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



# LIMITAZIONI DEL JAVA THREAD MODEL



*Java Thread model*

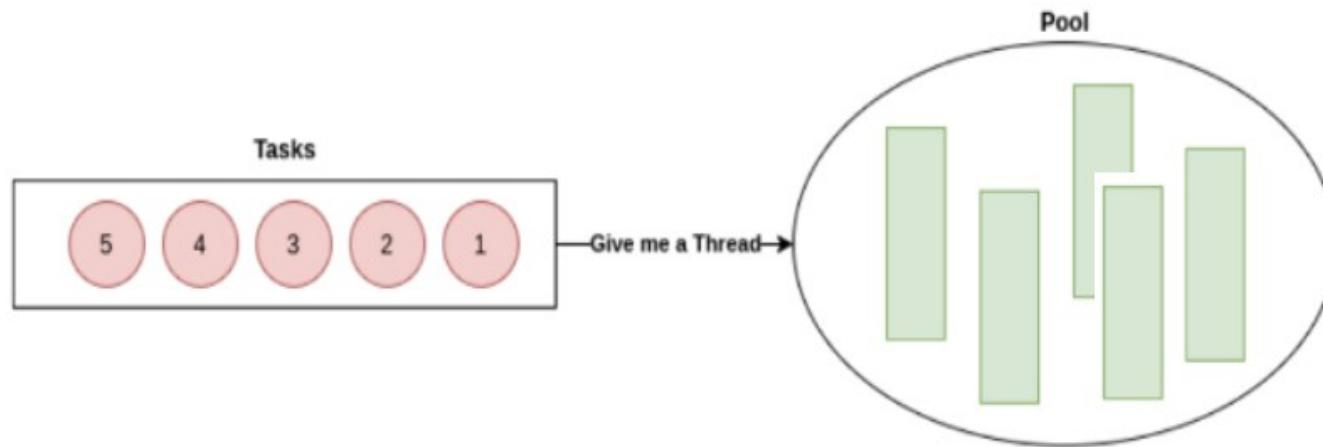
- numero di thread limitato dal livello di capacità di kernel thread
- “JAVA break” se si usano più thread di quelli supportati dal SO

# THREAD POOL: MOTIVAZIONI

- scenario di riferimento: si deve eseguire un gran numero di task
  - esempio: un task per ogni client, nel server
- un thread per ogni task: può diventare non sostenibile, specialmente nel caso di lightweight tasks molto frequenti.
- alternativa
  - creare un **pool di thread**
  - ogni thread può essere usato per l'esecuzione di più task
- obiettivo:
  - **riusare lo stesso thread** per l'esecuzione di più tasks
  - diminuire il costo per l'attivazione/terminazione dei threads
  - controllare il numero massimo di thread che possono essere eseguiti concorrentemente

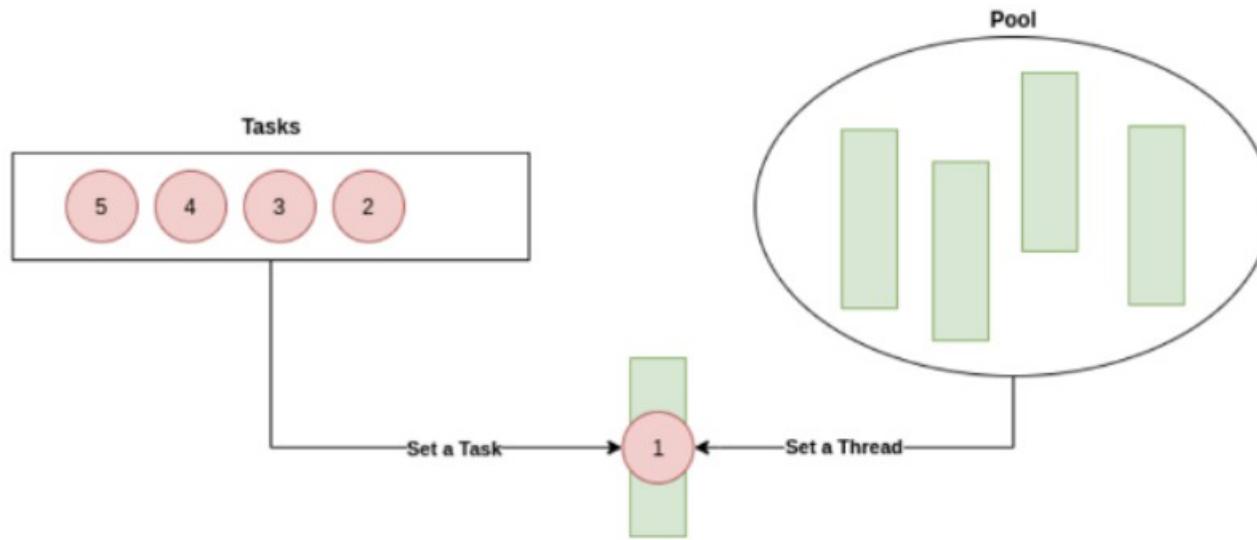


# THREAD POOLING: CONCETTI DI BASE



- una coda di task che aspettano l'esecuzione
  - politica FIFO per l'estrazione dei task dalla coda
- un pool di thread disponibili (rettangoli verdi) per l'esecuzione di un task
- il sistema di gestione del threadpool chiede se esiste un thread libero per l'esecuzione del primo task della coda

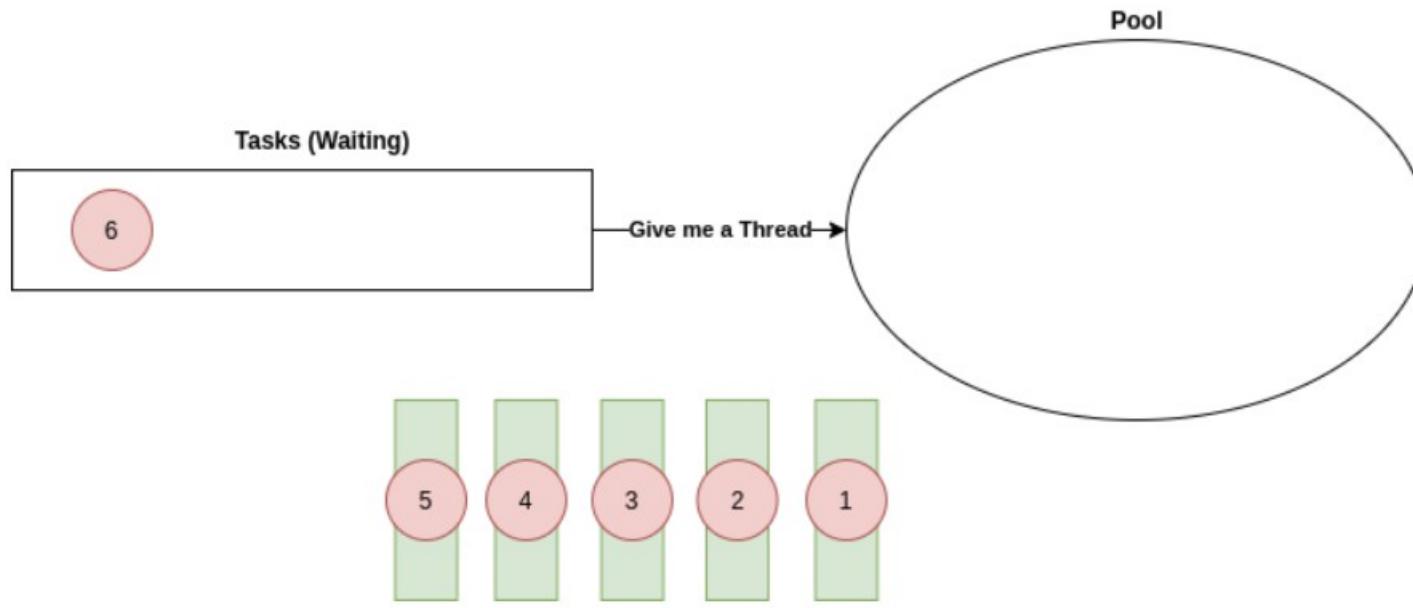
# THREAD POOLING: CONCETTI DI BASE



- il task viene assegnato ad un thread libero
- il thread viene tolto dal pool dei thread disponibili

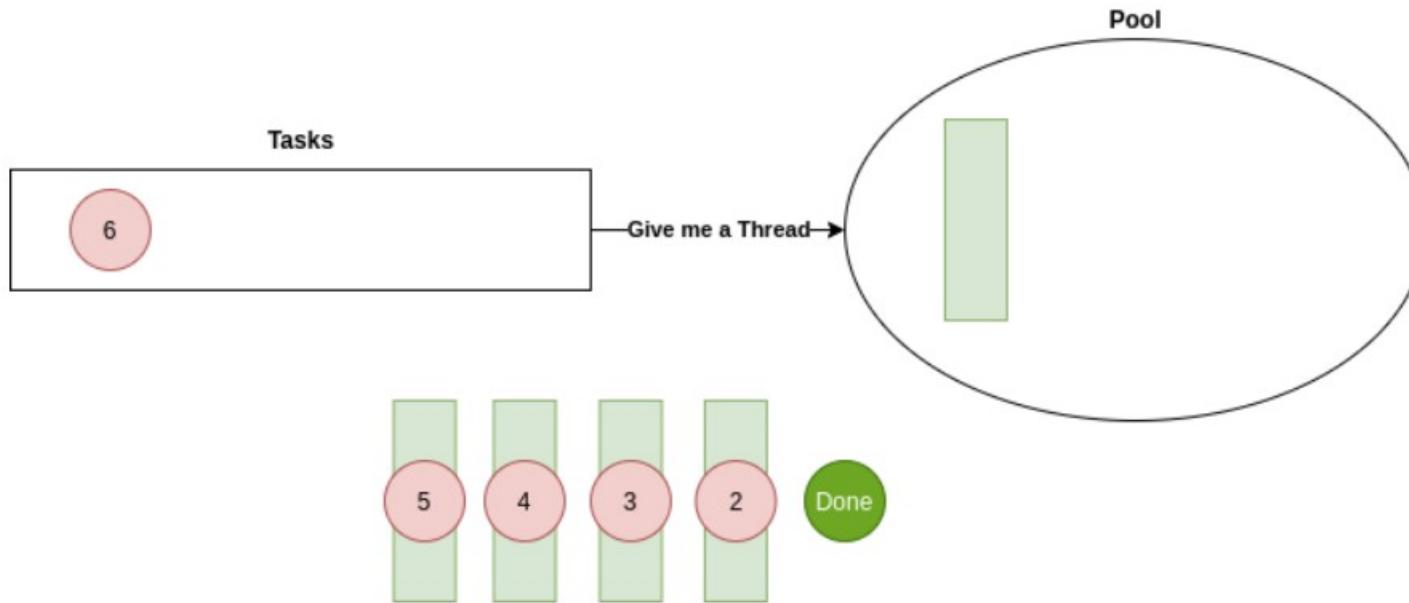


# THREAD POOLING: CONCETTI DI BASE



- tutti i threads sono occupati nella esecuzione di task
- il pool è vuoto
- due alternative
  - il task successivo viene inserito nella coda, in attesa che si renda disponibile un thread
  - si crea un nuovo thread all'interno del threadpool

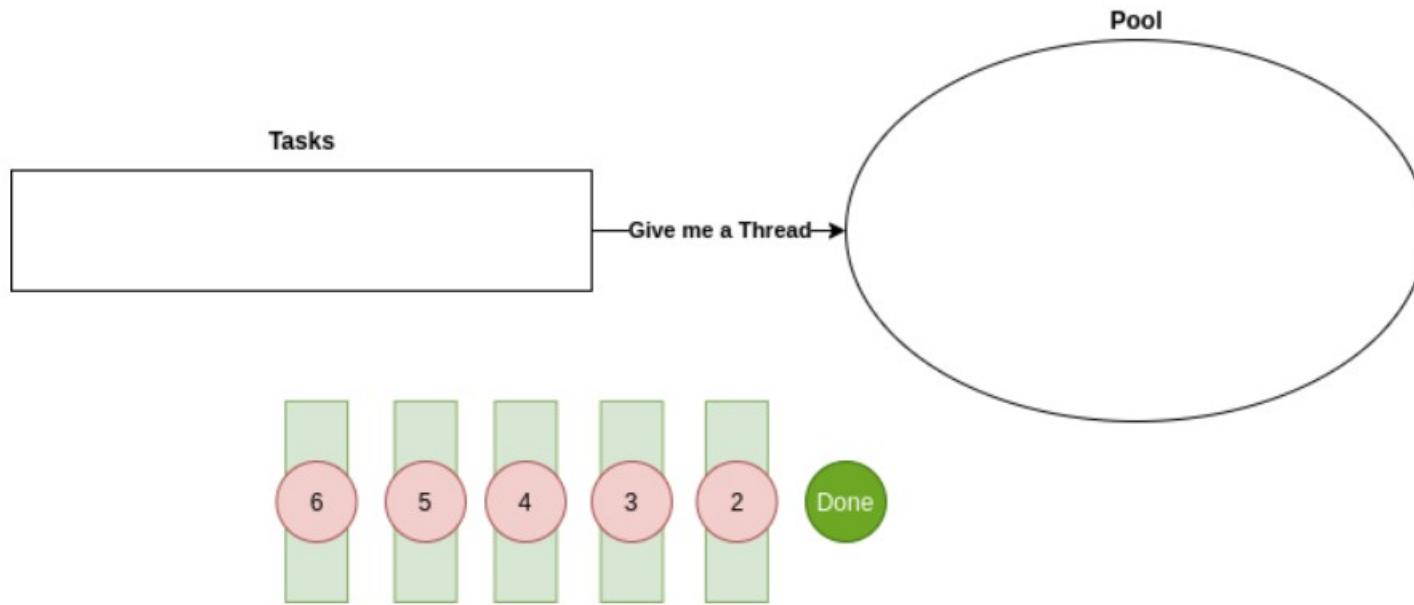
# THREAD POOLING



- supponiamo che il task 6 attenda nella coda
- quando un thread finisce l'esecuzione del task assegnato, il thread ritorna nel pool e si rende disponibile per l'esecuzione di un altro task



# THREAD POOLING: CONCETTI DI BASE



- il task in attesa viene associato al thread che si è reso disponibile
- il pool di thread ritorna ad essere vuoto
- il comportamento descritto è quello del `FixedThreadPool` di JAVA
  - in JAVA disponibili altre politiche di gestione dei threads



# 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
  - 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 la programmazione del threadpool è 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



# 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
- 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



# 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
```

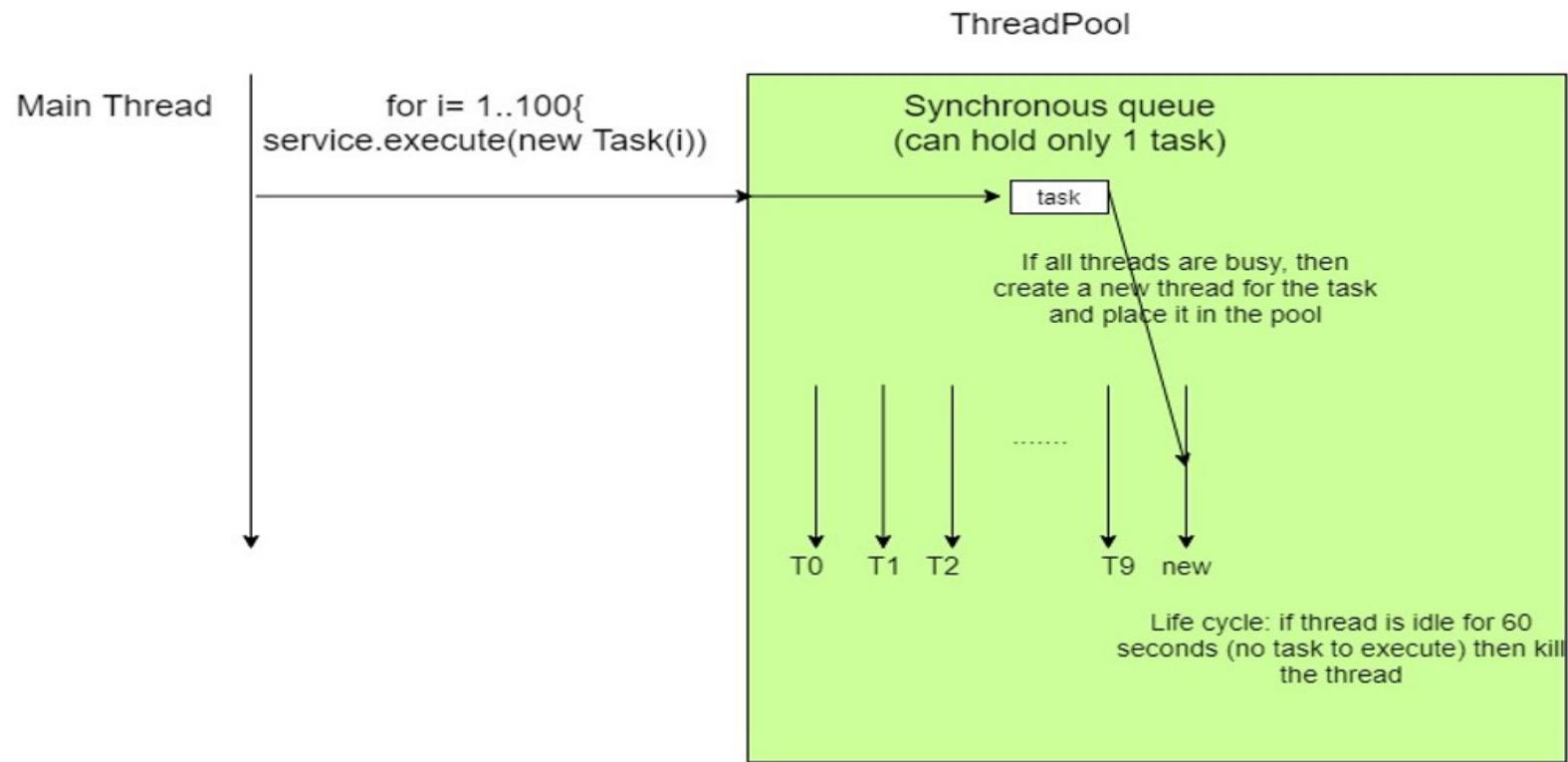
- cosa accade se si distanzia la sottomissione dei task ai thread, ad esempio inserendo una sleep, nel for dopo la execute?
- i thread sono tutti attivi e vengono utilizzati in modalità round-robin



# 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 thread 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



# INPUT/OUTPUT IN JAVA: RICHIAMI

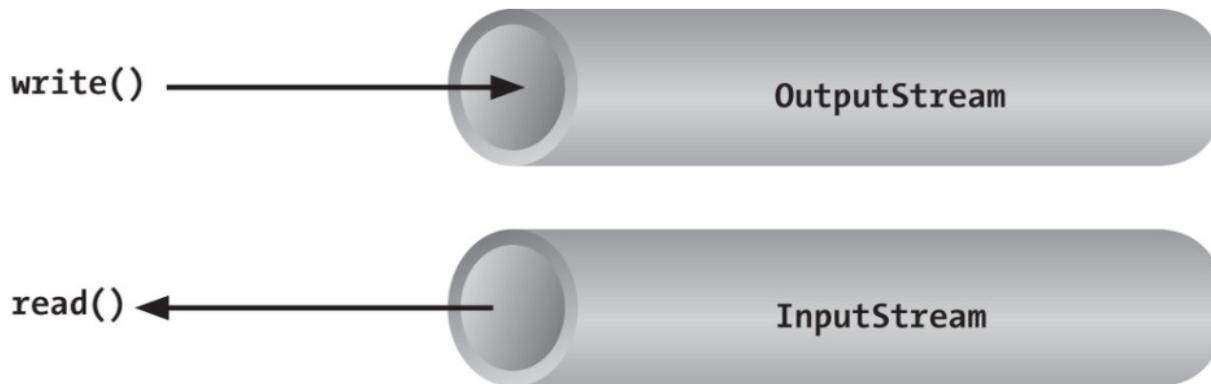
- I/O: reperire informazioni da una sorgente esterna o inviare ad una sorgente esterna
  - *file system*: files e directories
  - *connessioni di rete*
  - *keyboard*: System.in, System.out, System.error
  - *in-memory buffers (array)*
    - “vista” di un buffer di memoria come una sorgente o destinazione esterna.
      - un programma legge da un file csv.
      - per ottimizzare l’accesso ai dati si legge tutto il file in un buffer, in memoria centrale.
      - l’interfaccia verso il modulo che gestisce i dati deve rimanere la solita
  - utilizzo degli stream per le applicazioni di rete di rete:
    - in-memory buffers per la generazione di pacchetti UDP.



- definizione di un insieme di **astrazioni per la gestione dell'I/O**: una delle parti più complesse di un linguaggio
- diversi tipi di device di input/output: se il linguaggio dovesse gestire ogni tipo di device come caso speciale, la complessità sarebbe enorme
  - necessità di **astrazioni opportune** per rappresentare una device di I/O
- in JAVA, la prima astrazione definita è basata sul concetto di **stream** (o flusso)
- altre astrazioni per l'I/O
  - File: per manipolare descrittori di files
  - Channels (NIO)
  - ...

# L'ASTRAZIONE DEGLI STREAM

- uno stream rappresenta una connessione tra un programma JAVA ed un dispositivo esterno (file, buffer di memoria, connessione di rete,...)
- un flusso di informazione di lunghezza illimitata



- un “tubo” tra una sorgente ed una destinazione (dal programma ad un dispositivo e viceversa)
- l'applicazione inserisce dati o li legge ad/da un capo dello stream
- i dati fluiscono da/verso la destinazione

# JAVA STREAMS: CARATTERISTICHE GENERALI

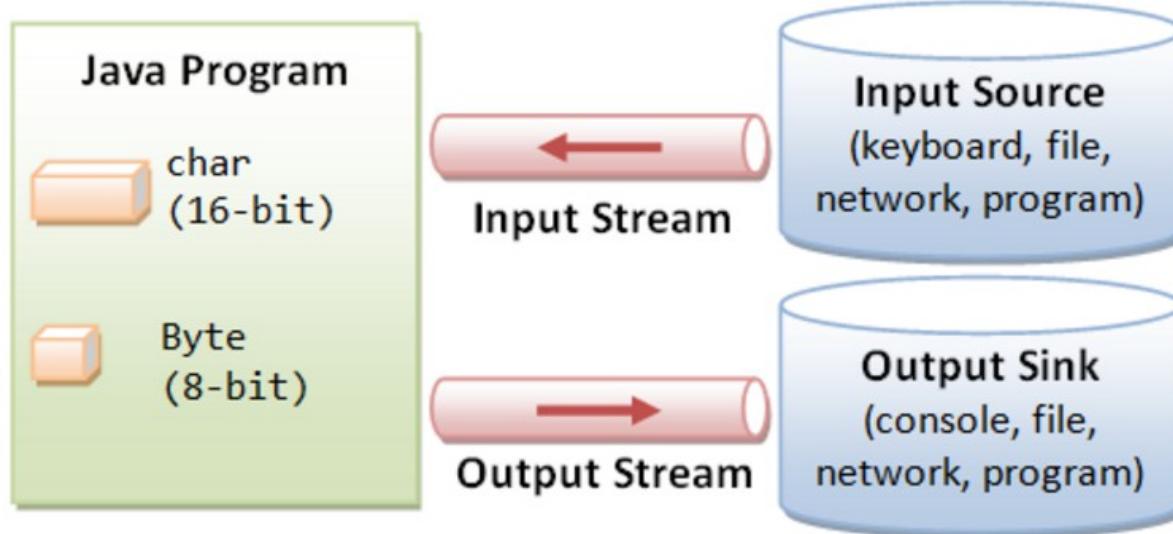
- accesso **sequenziale**
- mantengono l'ordinamento **FIFO**
- **one way**: read only oppure write only (a parte i file ad accesso random)
  - se un programma ha bisogno di dati in input ed output, è necessario aprire due stream, in input ed in output
- **bloccanti**: quando un'applicazione legge un dato dallo stream (o lo scrive) si blocca **finchè l'operazione non è completata**
- non è richiesta una corrispondenza stretta tra letture/scritture
  - una unica scrittura inietta 100 bytes sullo stream
  - i byte vengono letti con due write successive 'all'altro capo dello stream', la prima legge 20 bytes, la seconda 80 bytes)



# LE CLASSI PRINCIPALI: CARATTERI E BYTE

“Character” Streams  
(Reader/Writer)

“Byte” Streams  
(InputStream/  
OutputStream)



Internal Data Formats:

- Text (char): UCS-2
- int, float, double, etc.

External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)



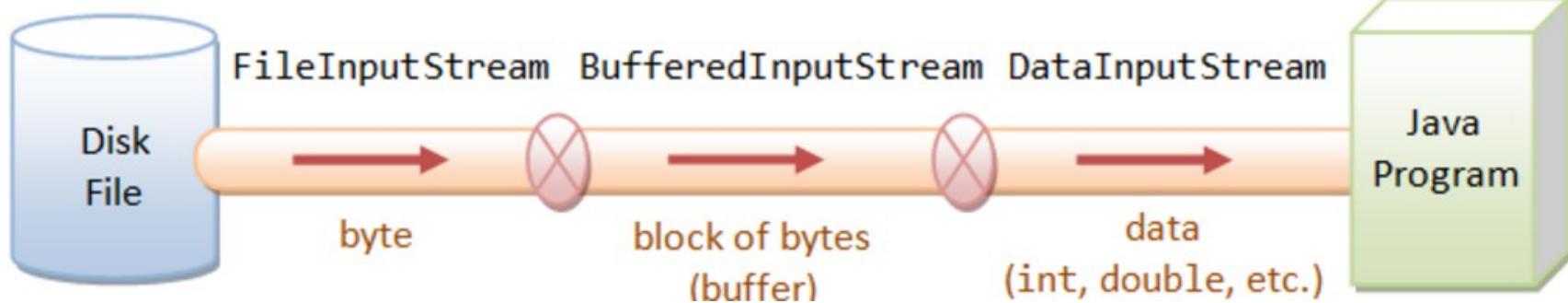
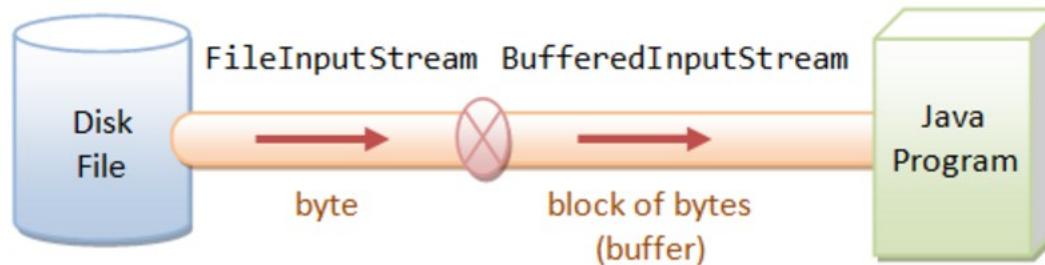
# JAVA: FILTER STREAMS

- InputStream and OutputStream operano su “row bytes”
- classi filtro compiono trasformazioni sui dati a basso livello. Tipi di filtri:
- filter Stream: trasformazioni effettuate
  - crittografia
  - compressione
  - Buffering
  - traduzione dei dati in un formato a più alto livello
- Readers Writes
  - orientati al testo e permettono di decodificare bytes in caratteri
- I filtri possono essere organizzati in catena. Ogni elemento della catena
  - riceve dati dallo stream o dal filtro precedente
  - passa i dati al programma o al filtro successivo



# BUFFERED STREAM/DATA STREAM

- implementano una bufferizzazione per stream di input e output,
- i dati vengono scritti e letti in blocchi di bytes, invece che un solo blocco per volta
- miglioramento significativo della performance



# ASSIGNMENT I

- scrivere un programma che dato in input una lista di directories, comprima tutti i file in esse contenuti, con l'utility *gzip*
- ipotesi semplificativa:
  - zippare solo i file contenuti nelle directories passate in input,
  - non considerare ricorsione su eventuali sottodirectories
- il riferimento ad ogni file individuato viene passato ad un task, che deve essere eseguito in un threadpool
- individuare nelle API JAVA la classe di supporto adatta per la compressione
- NOTA: l'utilizzo dei threadpool è indicato, perchè i task presentano un buon mix tra I/O e computazione
  - **I/O heavy**: tutti i file devono essere letti e scritti
  - **CPU-intensive**: la compressione richiede molta computazione
- facoltativo: comprimere ricorsivamente i file in tutte le sottodirectories



# Reti e Laboratorio III

## Modulo Laboratorio III

### AA. 2022-2023

docente: Laura Ricci

[laura.ricci@unipi.it](mailto:laura.ricci@unipi.it)

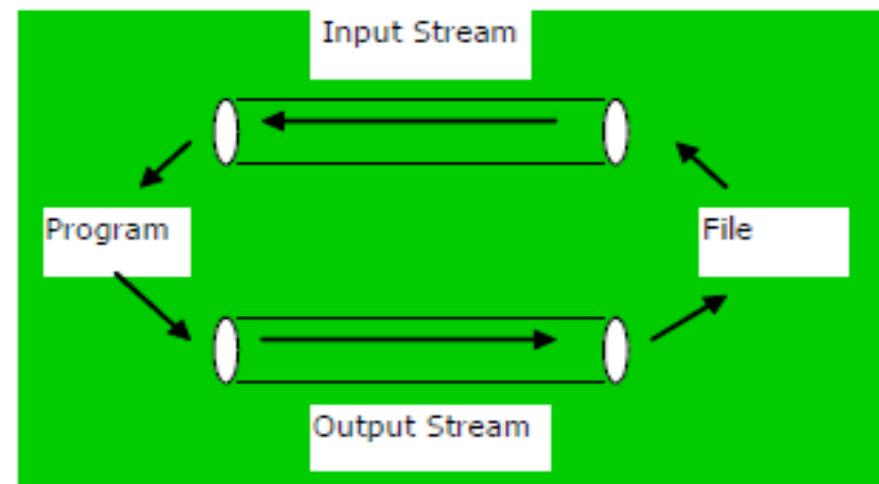
## Lezione 2

**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 accesso sequenziale, mantengono l'ordinamento FIFO
  - di uso generale
  - adatto a trasferire byte o caratteri
  - bloccante

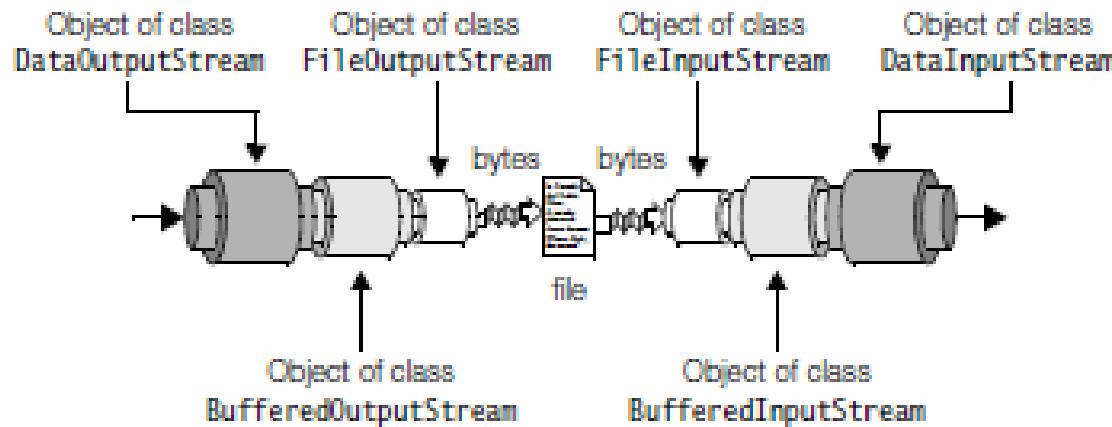


# IL PACKAGE JAVA.IO: OBIETTIVI

- 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



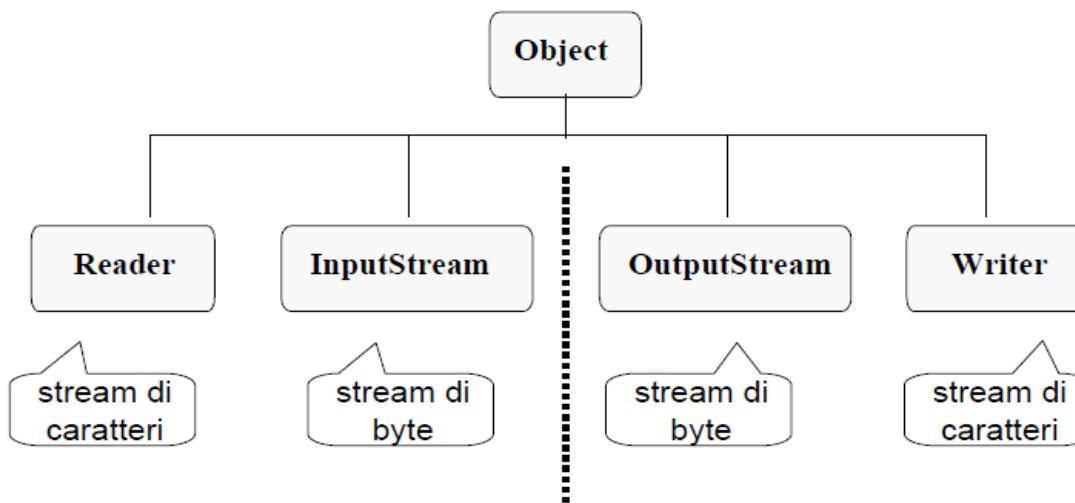
# IL PACKAGE JAVA.IO: OBIETTIVI



- 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

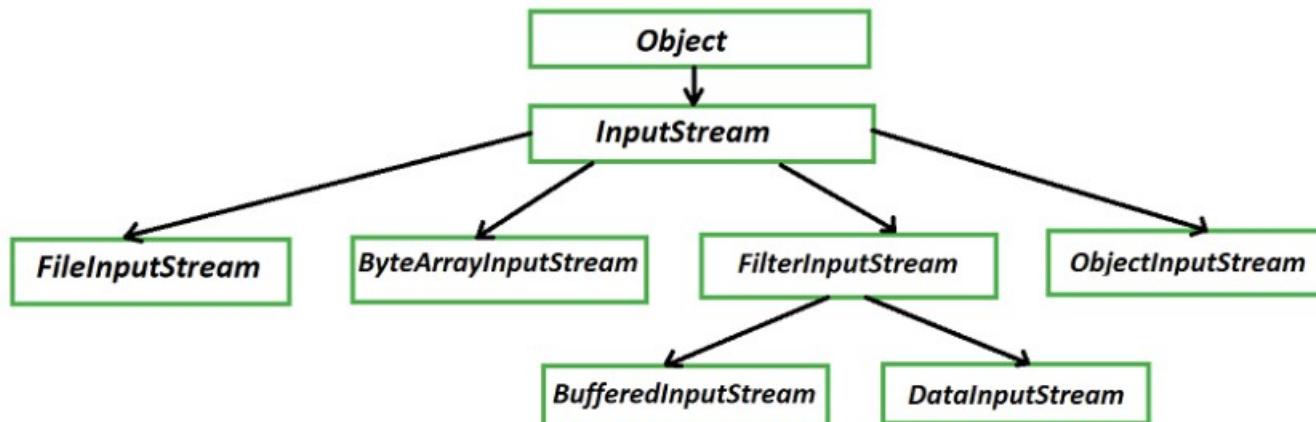
# IL PACKAGE JAVA.IO

- `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)))
```



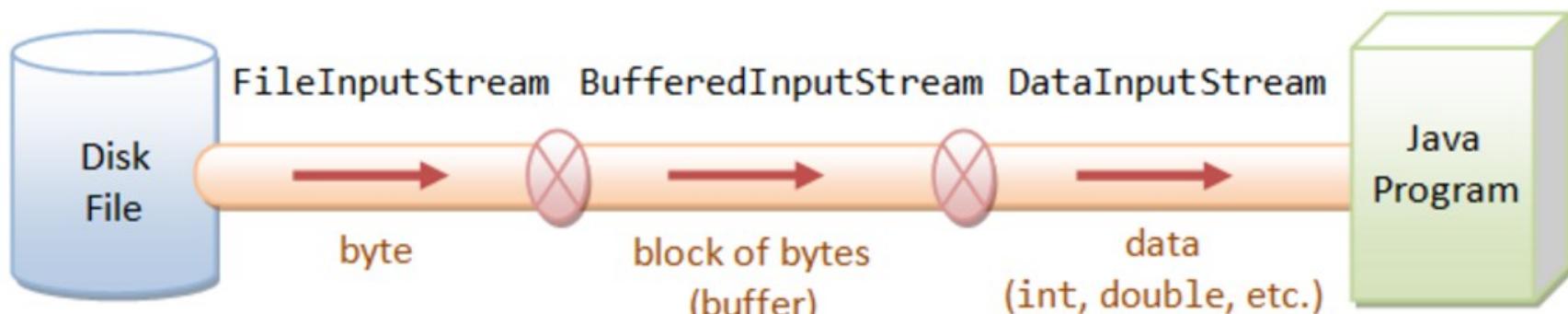
i tempi di esecuzione del programma si  
abbassano notevolmente

File size is 16473 bytes  
Elapsed Time is 1.2581 msec



# JAVA: FORMATTED DATA STREAM

```
import java.io.*;  
  
public class TestDataInputStream {  
    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)))) {
```

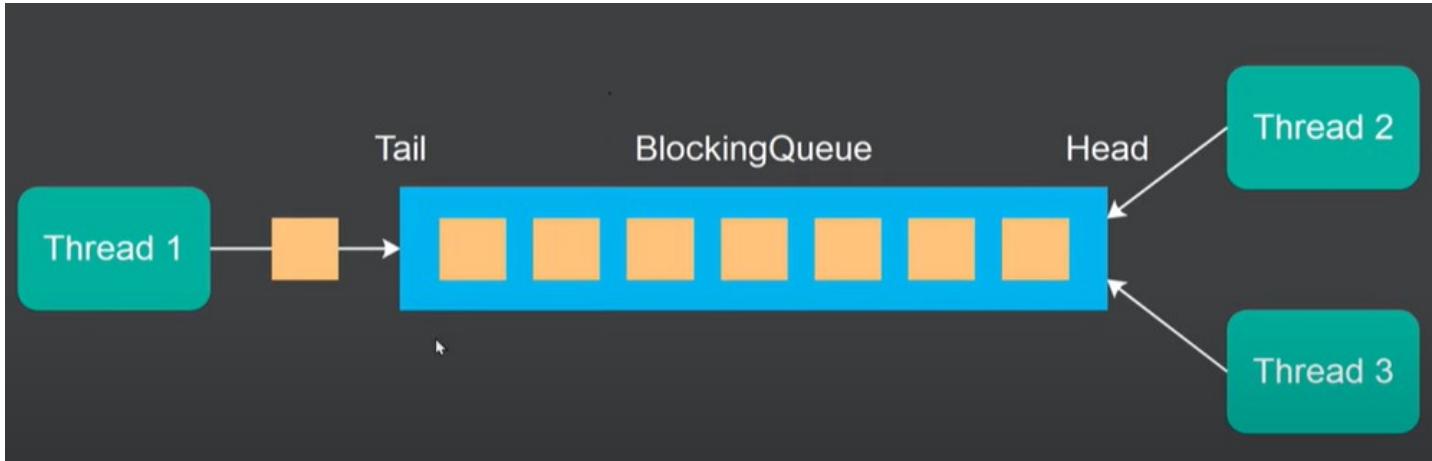


# JAVA: FORMATTED DATA STREAM

```
import java.io.*;  
  
public class TestDataInputStream {  
    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



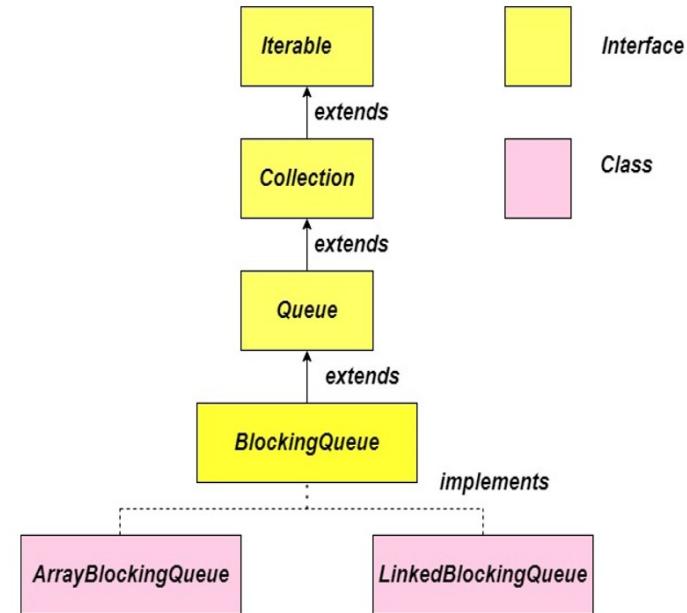
- **BlockingQueue** (`java.util.concurrent`): una JAVA interface che rappresenta una coda (inserimento alla fine, estrazione all'inizio)
- ...ma quale è la differenza con la interface `Queue<E>` (package JAVA.UTIL)?
  - pensata per essere utilizzata in un ambiente multithreaded
  - permettere una corretta 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();

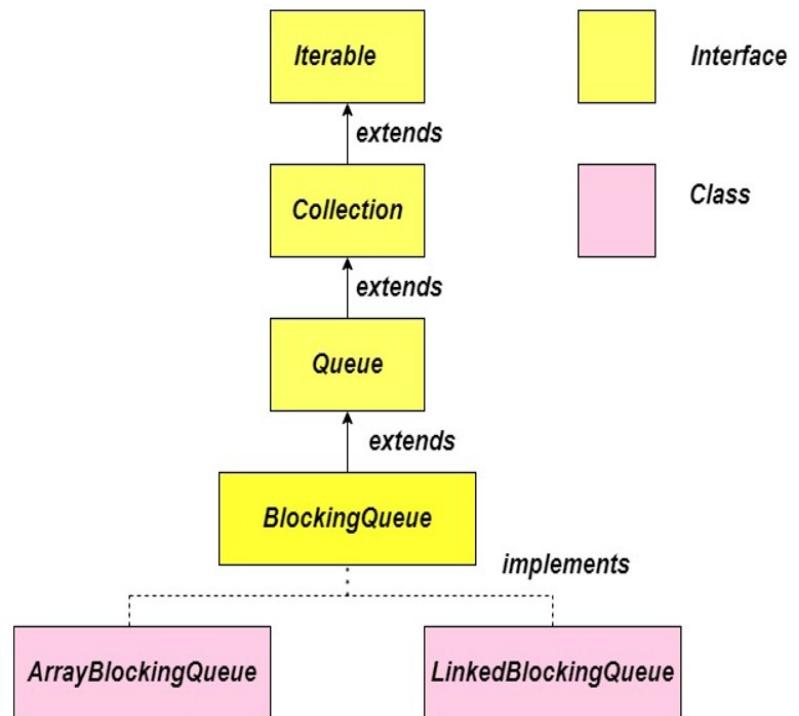
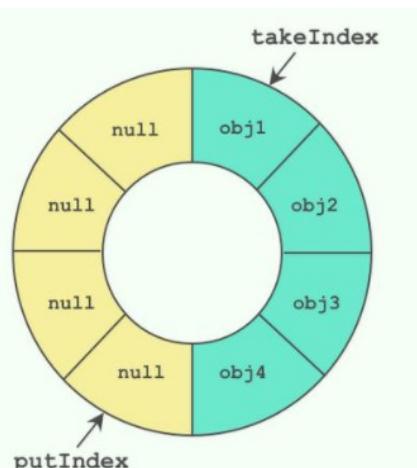
        // java.util.concurrent.DelayQueue
        // java.util.concurrent.LinkedTransferQueue
        // java.util.concurrent.PriorityBlockingQueue
        // java.util.concurrent.SynchronousQueue
    }
}
```



# QUALI CODE UTILIZZEREMO MAGGIORMENTE?

## ArrayBlockingQueue

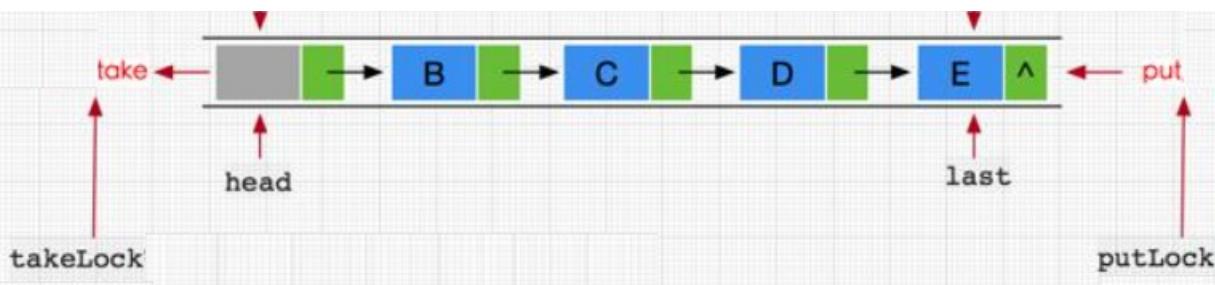
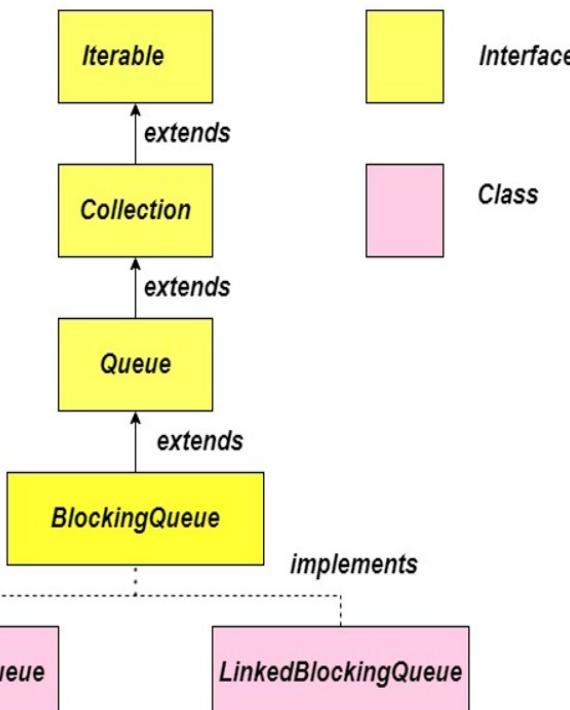
- dimensione limitata, definita in fase di inizializzazione
- memorizza gli elementi all'interno di un oggetto Array
  - nessun ulteriore oggetto creato
  - non sono possibili inserzioni/rimozioni in parallelo
  - una sola lock per tutta la struttura)



# 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
<b>Insert</b>	<code>add(o)</code>	<code>offer(o)</code>	<code>put(o)</code>	<code>offer(o, timeout, timeunit)</code>
<b>Remove</b>	<code>remove(o)</code>	<code>poll()</code>	<code>take()</code>	<code>poll(timeout, timeunit)</code>
<b>Examine</b>	<code>element()</code>	<code>peek()</code>		



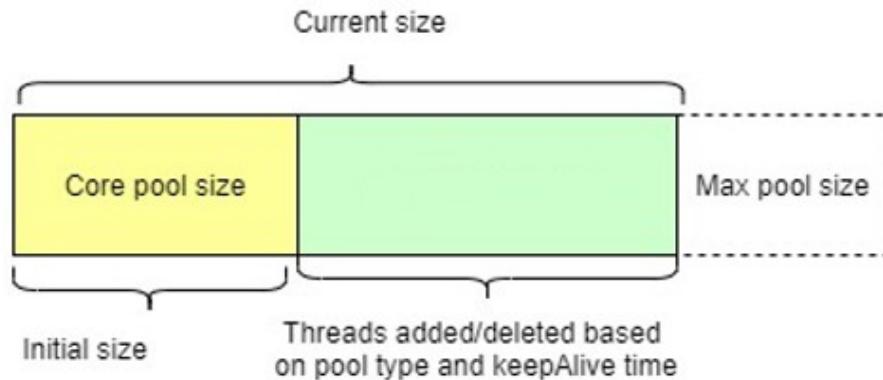
# LA CLASSE THREAD POOL EXECUTOR

```
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`



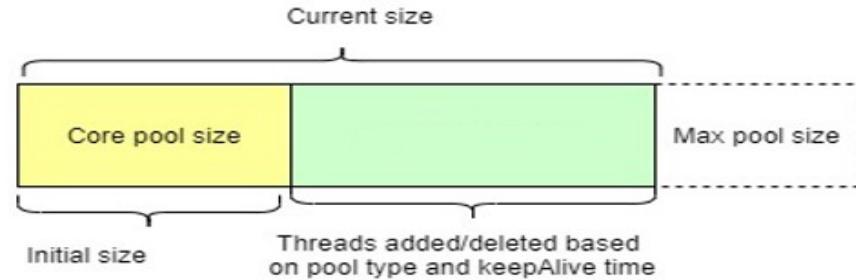
# THREAD POOL EXECUTOR: RIASSUNTO

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,

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

- 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)

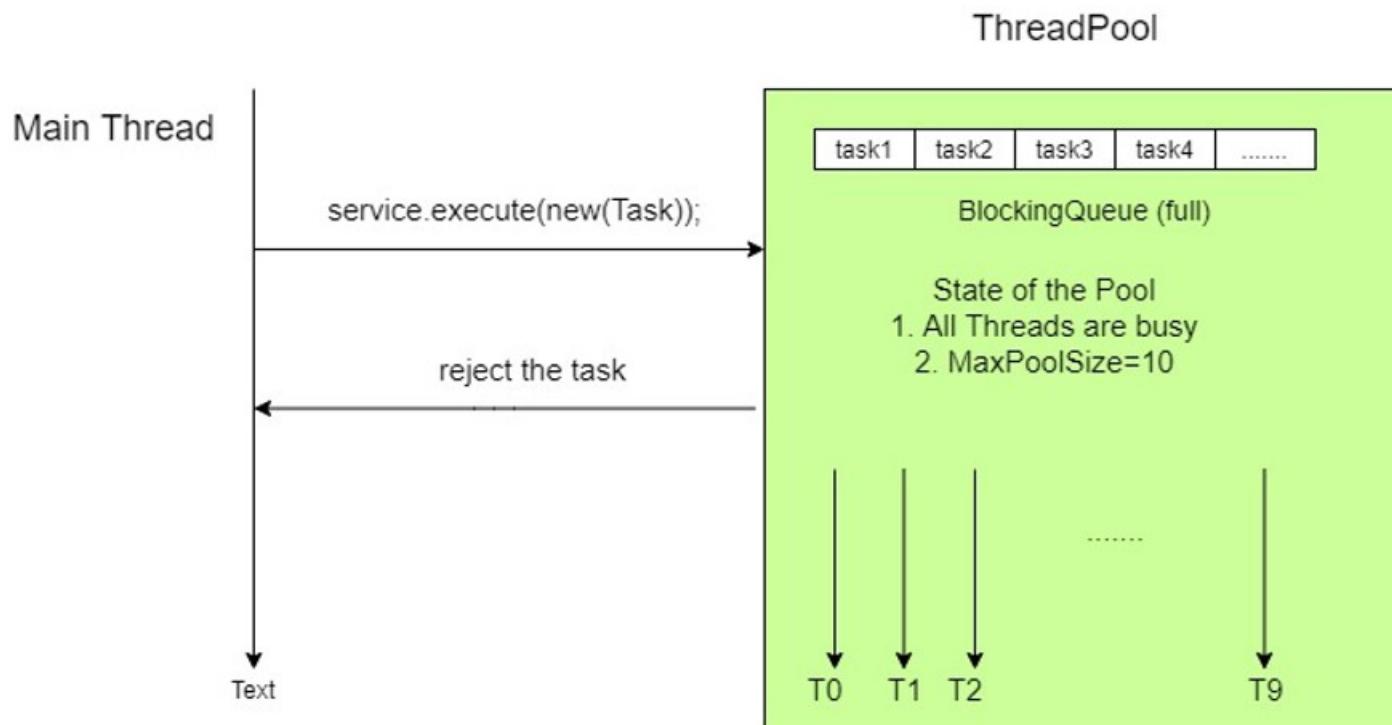
```
public static ExecutorService newFixedThreadPool(int nThreads) {  
    return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS, ....)  
  
public static ExecutorService newCachedThreadPool() {  
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS, ....);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

# 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 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 if 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.

# Reti e Laboratorio III

## Modulo Laboratorio III

**AA. 2022-2023**

**docente: Laura Ricci**

**[laura.ricci@unipi.it](mailto:laura.ricci@unipi.it)**

## **Lezione 3**

### **Callable, Scheduled Tasks, Monitor**

**29/9/2022**

# THREAD CHE RESTITUISCONO RISULTATI

- un oggetto di tipo Runnable
  - incapsula un'attività che viene eseguita in modo asincrono
  - il metodo run è un metodo asincrono, senza parametri e che non restituisce un valore di ritorno
- Interface Callable: consente di definire un task che può restituire un risultato e sollevare eccezioni
  - come “accedere” al risultato, in modo asincrono?
  - Future interface: contiene metodi per reperire, in modo asincrono, il risultato di una computazione asincrona. cioè
    - 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; }
```

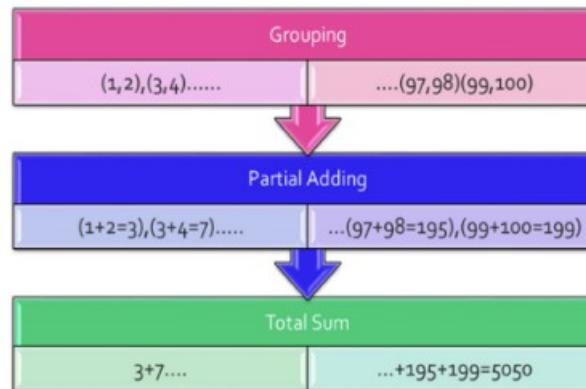
- contiene il solo metodo `call()`, analogo al metodo `run()` dell'interfaccia `Runnable`
- il codice del task è implementato nel metodo `call()`
- a differenza del metodo `run()`, il metodo `call()` può
  - restituire un valore
  - sollevare eccezioni
- il parametro di tipo `<V>` indica **il tipo del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



# DIVIDE ET IMPERA CON MULTITHREADING

calcolare la somma di tutti i numeri da 1 a n

- soluzione sequenziale: loop che itera da 1 a 100 e calcola la somma
- seguendo il pattern divide and conquer :
  - individuare sottointervalli dell'intervallo 1-100
  - creare un task diverso per ogni intervallo: calcola la somma per quell'intervallo
  - sottomettere i task ad un theradpool, somme parziali in parallelo
  - raccogliere le somme parziali per calcolare la somma totale.



# THREAD CHE RESTITUISCONO RISULTATI

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

public class Calculator implements Callable<Integer> {

    private int a;
    private int b;

    public Calculator(int a, int b) {
        this.a = a;
        this.b = b;
    }

    public Integer call() throws Exception {
        Thread.sleep((long)(Math.random() * 15000));
        return a + b;
    }
}
```



# THREAD CHE RESTITUISCONO RISULTATI

```
import java.util.ArrayList; import java.util.List; import java.util.concurrent.*;  
  
public class Adder {  
  
    public static void main(String[] args) throws ExecutionException,InterruptedException{  
  
        // Create thread pool using Executor Framework  
  
        ExecutorService executor = Executors.newFixedThreadPool(5);  
  
        List<Future<Integer>> list = new ArrayList<Future<Integer>>();  
  
        for (int i = 0; i < 10; i=i+2) { // Create new Calculator object  
  
            Calculator c = new Calculator(i, i + 1);  
  
            list.add(executor.submit(c));}  
  
        int s=0;  
  
        for (Future<Integer> f : list) {  
  
            try { System.out.println(f.get());  
  
                s=s+f.get();  
  
            } catch (Exception e) {}}  
  
        System.out.println("la somma e'" +s);  
  
        executor.shutdown(); } }
```



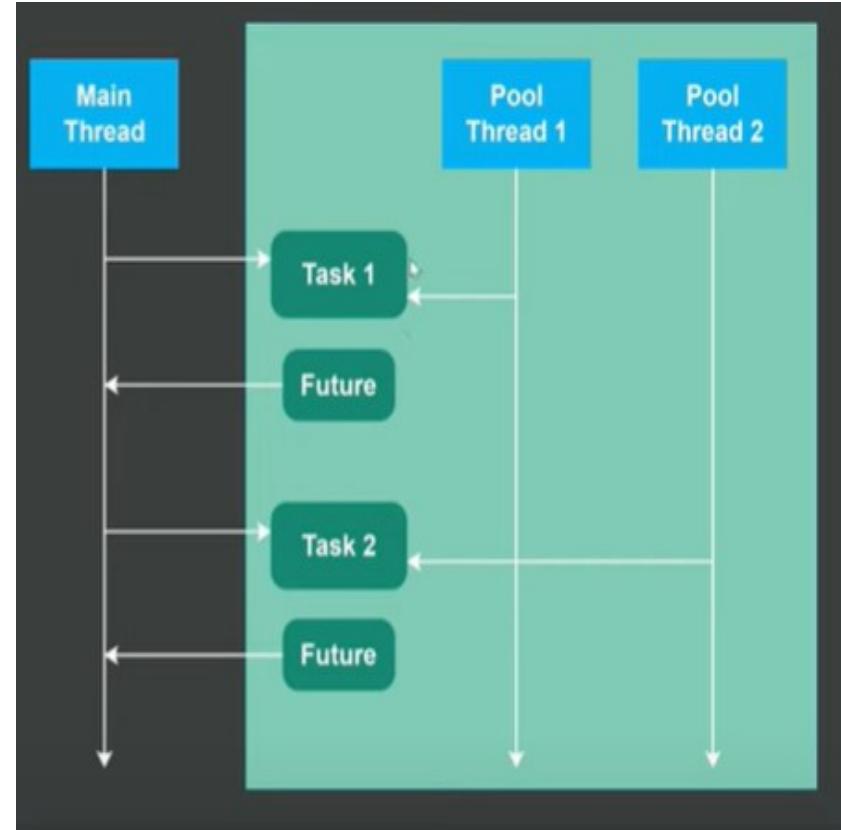
# THREAD CHE RESTITUISCONO RISULTATI

```
import java.util.ArrayList; import java.util.List; import java.util.concurrent.*;  
  
public class Adder {  
  
    public static void main(String[] args) throws ExecutionException, InterruptedException {  
  
        ExecutorService executor = Executors.newFixedThreadPool(5);  
  
        List<Future<Integer>> list = new CopyOnWriteArrayList<Future<Integer>>();  
  
        for (int i = 0; i < 10; i=i+2) {  
  
            Calculator c = new Calculator(i, i + 1);  
  
            list.add(executor.submit(c));}  
  
        int s=0;  
  
        while (!list.isEmpty() )  
  
            for (Future<Integer> f : list) {  
  
                if (f.isDone())  
  
                    {System.out.println(f.get());  
  
                     s=s+f.get();  
  
                     list.remove(f);}}  
  
        System.out.println("la somma e'" +s); executor.shutdown();}}  
}
```



# L'INTERFACCIA FUTURE

- sottomettere 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



# 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

# 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)
- 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
  - dopo un certo periodo di tempo (delay)
  - periodicamente
- schedule(Runnable command, long delay, TimeUnit unit)
  - esegue un task Runnable (o Callable) dopo un certo intervallo di tempo
- scheduleAtFixedRate(Runnable command, long initialDelay, long delay, TimeUnit unit)
  - esegue un task dopo un intervallo iniziale, poi lo ripete periodicamente.
  - se il tempo di esecuzione del task è maggiore del periodo specificato, le sue seguenti esecuzioni possono essere ritardate.
- scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)
  - esegue un task dopo un intervallo iniziale, poi lo ripete periodicamente con un intervallo dato tra la terminazione di una esecuzione e l'inizio della successiva

# 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);}}}
```



# UN TASK “COUNTDOWN”

```
public class CountDownClock implements Runnable {  
    private String clockName;  
  
    public CountDownClock(String clockName) {  
        this.clockName = clockName;  
    }  
  
    public void run() {  
        String threadName = Thread.currentThread().getName();  
        for (int i = 5; i >= 0; i--) {  
            System.out.printf("%s -> %s: %d\n", threadName, clockName, i);  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException ex) {  
                ex.printStackTrace();  
            }  
        }  
    }  
}
```



# COUNTDOWN SCAGLIONATI NEL TEMPO

```
import java.util.concurrent.*;  
  
public class ConcurrentScheduledTaskExample {  
  
    public static void main(String[] args) {  
  
        ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(3);  
  
        CountDownClock clock1 = new CountDownClock("A");  
  
        CountDownClock clock2 = new CountDownClock("B");  
  
        CountDownClock clock3 = new CountDownClock("C");  
  
        scheduler.scheduleWithFixedDelay(clock1, 3, 10, TimeUnit.SECONDS);  
        scheduler.scheduleWithFixedDelay(clock2, 3, 15, TimeUnit.SECONDS);  
        scheduler.scheduleWithFixedDelay(clock3, 3, 20, 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

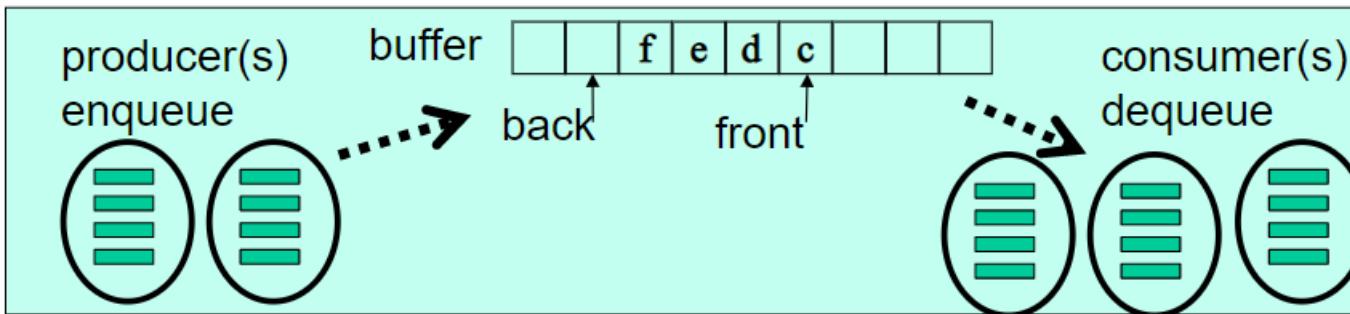


# ALTERNATIVE PER DEFINIRE CLASSI THREAD SAFE

- 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
      - SynchronousQueue
      - CopyOnWriteArrayList
      - CopyOnWriteArraySet
      - ConcurrentHashMap
  - i monitor
  - le lock a basso livello (non le vedremo)



# IL PROBLEMA DEL PRODUTTORE CONSUMATORE



- un classico problema che descrive due (o più thread) che condividono un buffer, di dimensione fissata, usato come una coda
  - il produttore P produce un nuovo valore, lo inserisce nel buffer e torna a produrre valori
  - il consumatore C consuma il valore (lo rimuove dal buffer) e torna a richiedere valori
  - garantire che il produttore non provi ad aggiungere un dato nelle coda se è piena ed il consumatore non provi a rimuovere un dato da una coda vuota
- generalizzazione per più produttori e più consumatori

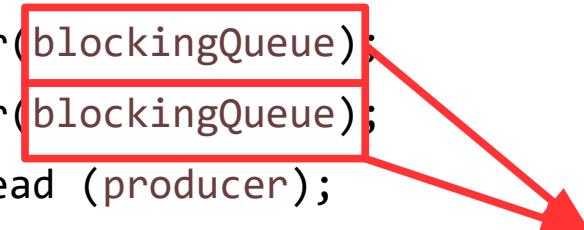
# PRODUTTORE CONSUMATORE: SINCRONIZZAZIONE

- l'interazione esplicita tra threads avviene in JAVA mediante l'utilizzo di **oggetti condivisi**
  - la **coda** che memorizza i messaggi scambiati tra P e C è condivisa
- necessari costrutti per **sospendere** un thread T quando **una condizione** non è verificata e **riattivare** T quando diventa vera
  - il produttore si sospende se la coda è piena
  - si riattiva quando c'è una posizione libera
- due tipi di sincronizzazione:
  - **implicita**: la mutua esclusione sull'oggetto condiviso è garantita dall'uso di lock (implicite o esplicite)
  - **esplicita**: occorrono altri meccanismi

# PRODUTTORE/CONSUMATORE CON BLOCKINGQUEUES

```
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 riferimento alla struttura dati condivisa si passa ad entrambi i thread

```
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()} }
```



```
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();}
        }
    }
}
```



# IL MONITOR

- meccanismo linguistico ad alto livello per la sincronizzazione
  - idea introdotta negli anni '70 safe (*Per Brinch Hansen, Hoare 1974*)
- incapsula un **oggetto condiviso** e le operazioni che vengono invocate dai threads su di esso, in modo concorrente
- funzionalità offerte dal monitor
  - **mutua esclusione** sulla struttura: lock implicite gestite dalla JVM: un solo thread per volta accede all'oggetto condiviso
  - **coordinazione** tra i thread
    - meccanismi per la sospensione sullo stato dell'oggetto condiviso, simili a variabili di condizione: wait
    - meccanismi per la notifica di una condizione ai thread sospesi su quella condizione + notify/notifyall



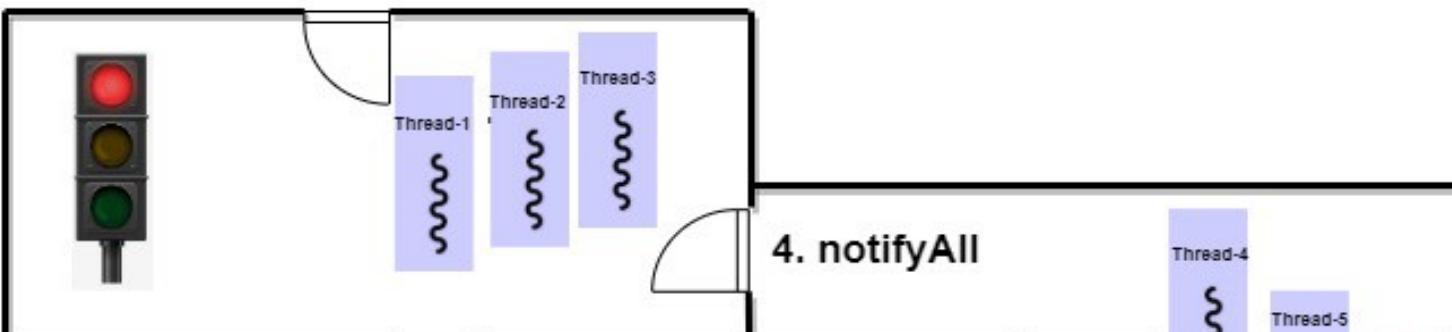
# IL MONITOR

- JAVA built-in monitor: classe di oggetti utilizzabili concorrentemente in modo thread safe
  - meccanismi di sincronizzazione “ad alto livello”
- come viene implementato? ad **ogni oggetto** (non **int** o **long**, solo gli oggetti), cioè ad ogni **istanza di una classe**, viene associata
  - una “**intrinsic lock**” o **lock implicita**
    - acquisita con metodi o blocchi di codice `synchronized`. Garantisce la mutua esclusione nell'accesso all'oggetto
    - gestione automatica della coda di attesa, da parte della JVM
  - una “**wait queue**” gestita dalla JVM
    - `wait`
    - `notify/notifyAll`

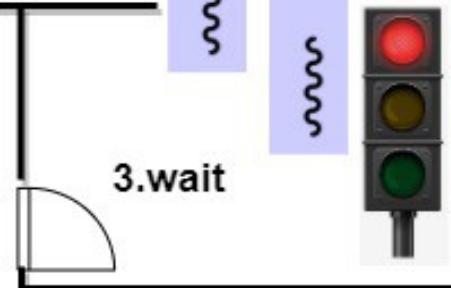


# UN'OCCHIATA ALL'INTERNO DI UN MONITOR

1. Entra nell'oggetto monitor



4. notifyAll



2. Acquisisci la lock

3.wait

Critical Section

5. Rilascia la lock

6. Lascia l'oggetto monitor

due code gestite in modo隐式:

Entry Set

- threads in attesa di acquisire la lock

Wait Set

- threads che hanno eseguito una wait  
e sono in attesa di una notifyAll

# METODI SINCRONIZZATI

- i metodi di un built-in monitor possono essere resi thread safe annotandoli con la parola chiave synchronized
- coda thread-safe, implementata con monitor

```
public class MessageQueue {  
    public MessageQueue(int size)  
    public synchronized void produce(Object x)  
    public synchronized Object consume()
```

- l'esecuzione di un metodo synchronized richiede automaticamente l'acquisizione della lock intrinseca associata all'oggetto
- l'intero codice del metodo sincronizzato viene serializzato rispetto agli altri metodi sincronizzati definiti per lo stesso oggetto
  - solo una thread alla volta può essere eseguire uno dei metodi synchronized del monitor sulla stessa istanza di una classe



```
public synchronized void someMethod()  
{ // Do work}
```

metodo synchronized : quando viene invocato

- tenta di acquisire la lock intrinseca associata all'istanza dell'oggetto su cui esso è invocato
  - se l'oggetto è bloccato il thread viene sospeso nella coda associata all'oggetto fino a che il thread che detiene la lock la rilascia
- la lock viene rilasciata al ritorno del metodo
  - normale
  - eccezionale, ad esempio con una uncaught exception.

- i costruttori non devono essere dichiarati synchronized
  - il compilatore solleva una eccezione
  - per default, solo il thread che crea l'oggetto accede ad esso mentre l'oggetto viene creato
- non ha senso specificare synchronized nelle interfacce
- synchronized non è ereditato da overriding
  - metodo nella sottoclasse deve essere esplicitamente definito synchronized, se necessario
- la lock è associata ad un'istanza dell'oggetto, non alla classe, metodi su oggetti che sono istanze diverse della stessa classe possono essere eseguiti in modo concorrente!

# WAITING AND COORDINATION MECHANISMS

- JAVA fornisce 3 metodi di base per coordinare i thread
- invocati su un oggetto, appartengono alla classe **Object**
- occorre acquisire la lock intrinseca prima di invocarli, altrimenti viene sollevata l'eccezione **IllegalMonitorException()**
  - eseguiti all'interno di metodi sincronizzati
  - se non si mette il riferimento ad un oggetto, il riferimento隐式 è **this**

## **void wait()**

- sospende il thread fino a che un altro thread invoca una `notify()` /`notifyAll()` sullo stesso oggetto.
- implementa una “attesa passiva” del verificarsi di una condizione
- rilascia la lock sull'oggetto

## **void notify()**

- sveglia un singolo thread in attesa su questo oggetto
- nop se nessun thread è in attesa

## **void notifyAll()**

- sveglia tutti i thread in attesa su questo oggetto, che competono per riacquisire della lock



# PRODUTTORE CONSUMATORE CON MONITOR

```
public class MessageQueue {  
    int putptr, takeptr, count;  
    final Object[] items;  
    public MessageQueue(int size){  
        items = new Object[size];  
        count=0;putptr=0;takeptr=0;}  
    public synchronized void produce(Object x)  
    { while (count == items.length)  
        try {  
            wait();}  
        catch(Exception e) {}  
        // gestione puntatoricoda  
        items[putptr] = x; putptr++;++count;  
        if (putptr == items.length) putptr = 0;  
        System.out.println("Message Produced"+x);  
        notifyAll();}
```



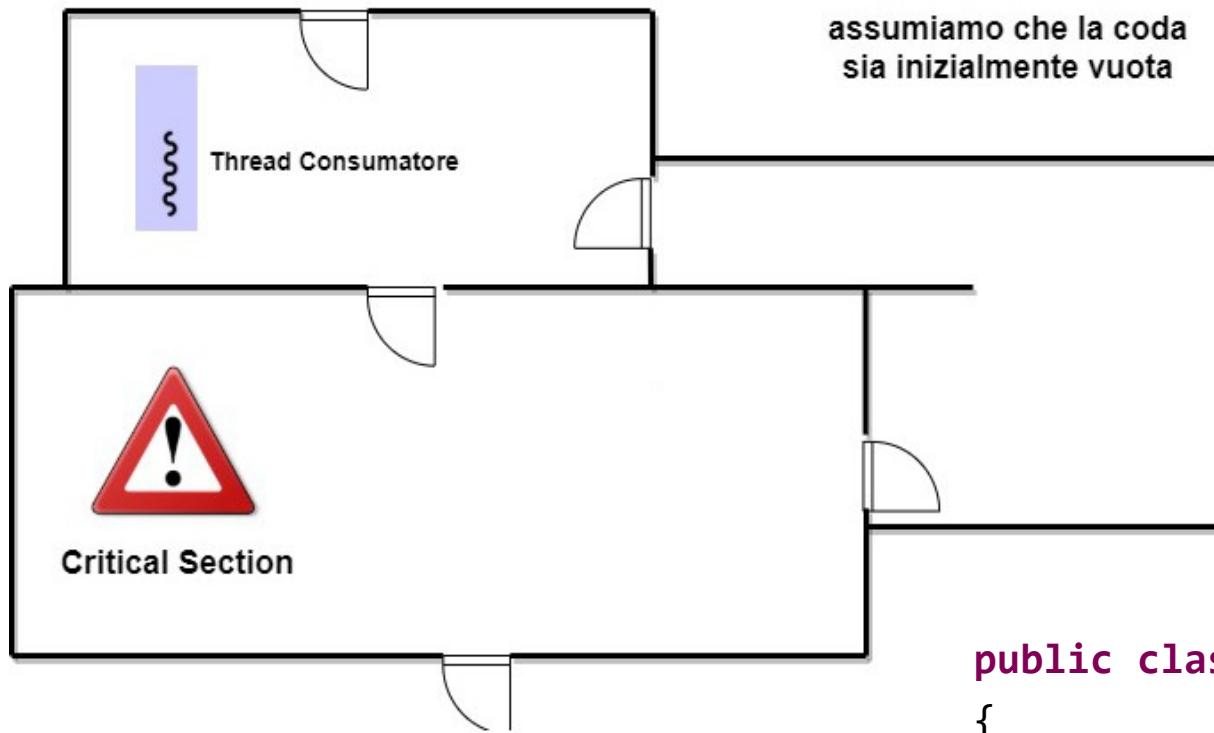
# PRODUTTORE CONSUMATORE CON MONITOR

```
public synchronized Object consume() {  
    while (count == 0)  
        try {  
            wait();}  
        catch(InterruptedException e) {}  
    // gestione puntatori coda  
    Object data = items[takeptr]; takeptr=takeptr+1; --count;  
    if (takeptr == items.length) {takeptr = 0;}  
    notifyAll();  
    System.out.println("Message Consumed"+data);  
    return data;  
}
```



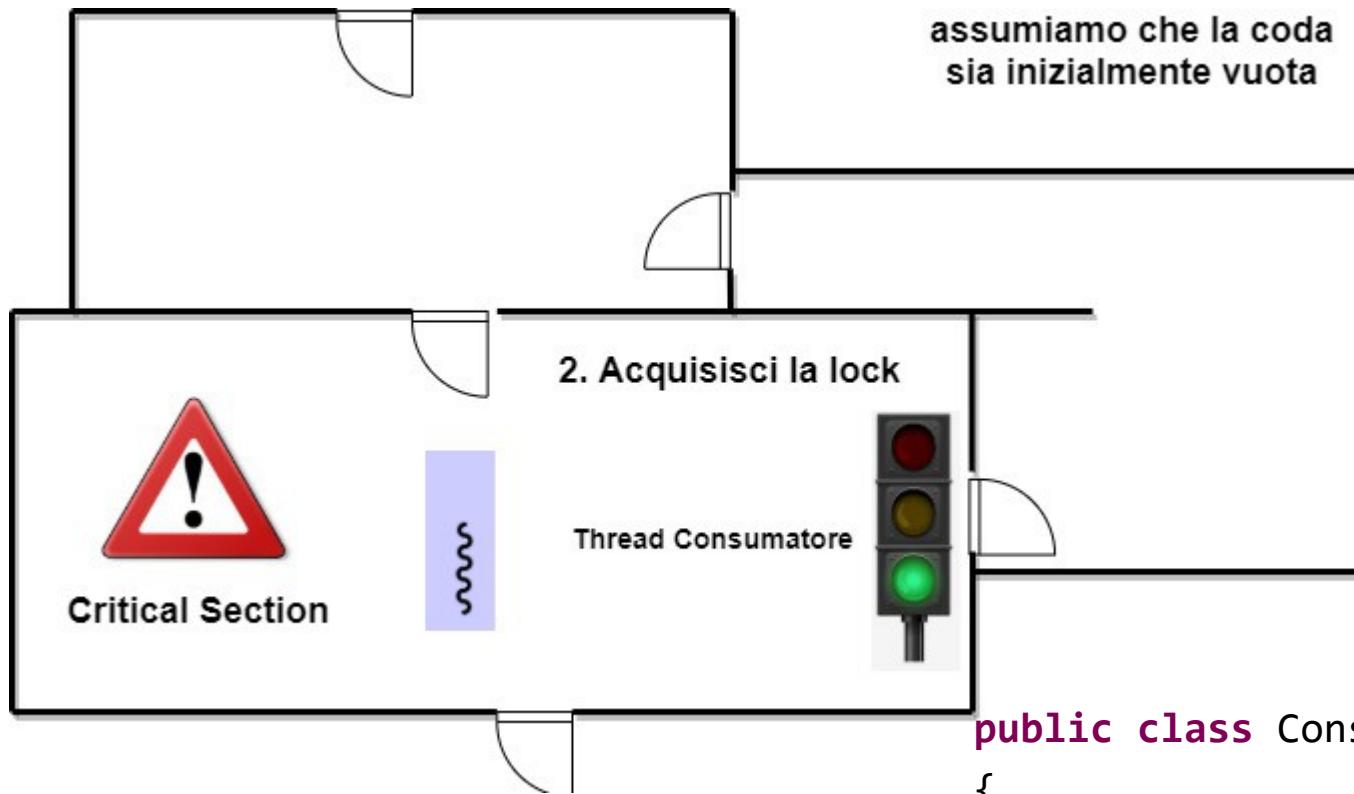
# PRODUTTORE CONSUMATORE “ILLUSTRATO”

1. Entra nell'oggetto monitor



```
public class Consumer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++){  
            Object o=queue.consume();  
            ... }  
    }  
}
```

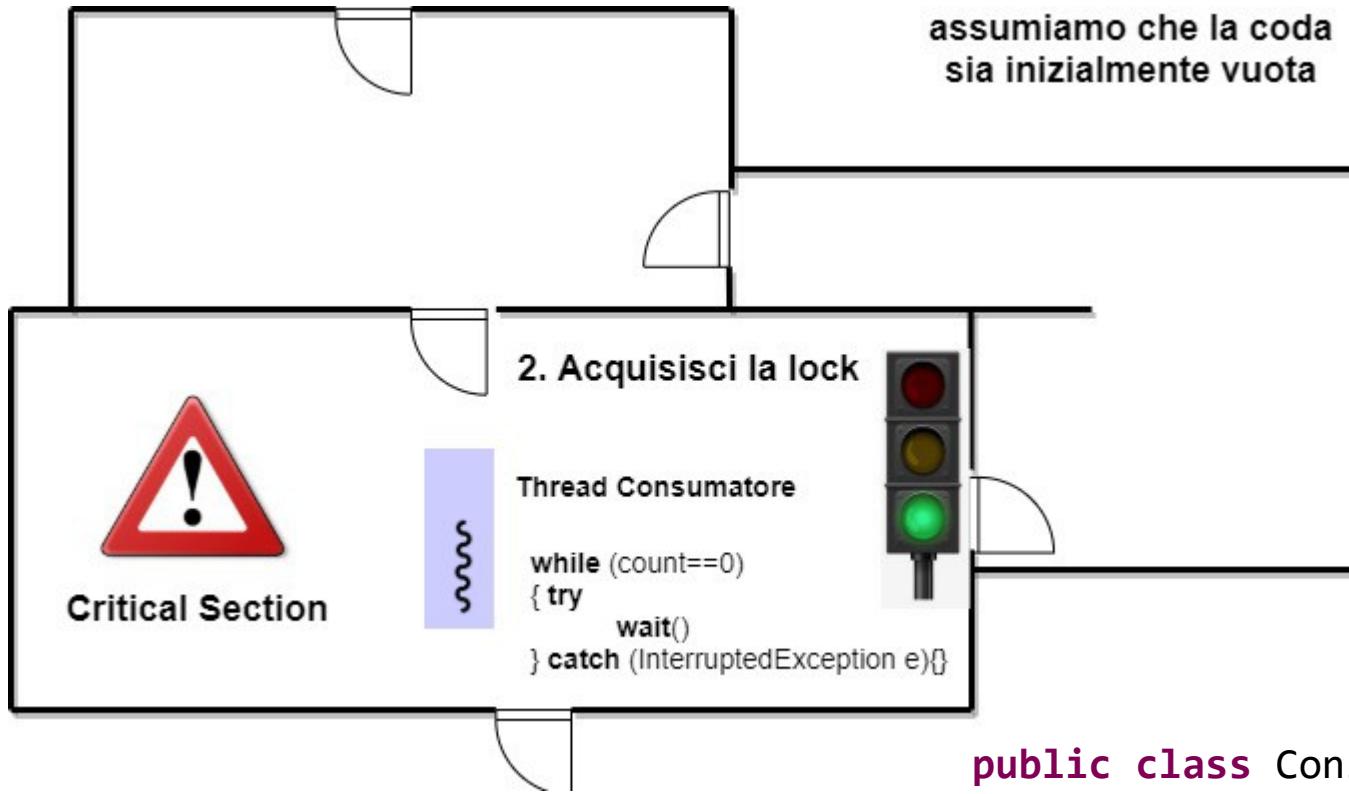
# PRODUTTORE CONSUMATORE “ILLUSTRATO”



```
public class Consumer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++){  
            Object o=queue.consume();  
            ... }  
    }  
}
```

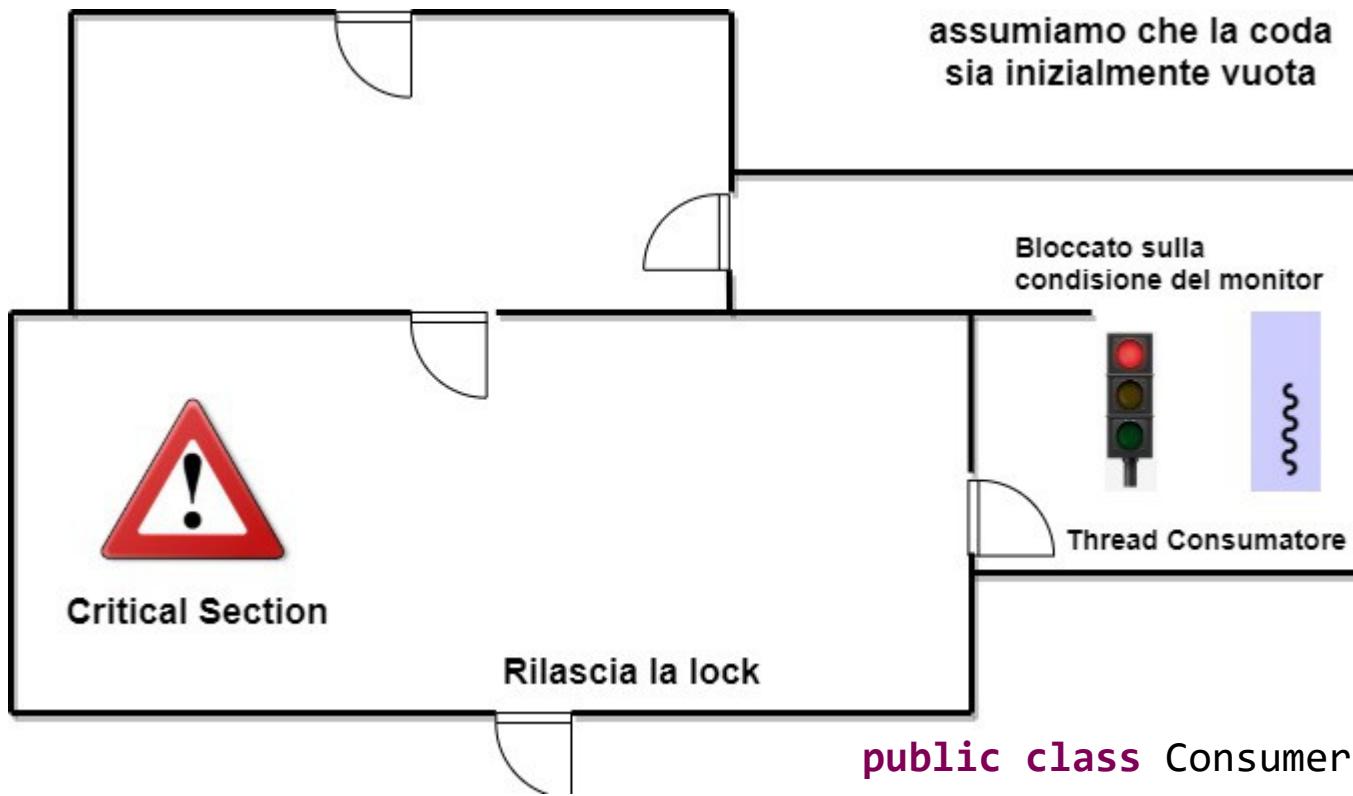


# PRODUTTORE CONSUMATORE “ILLUSTRATO”



```
public class Consumer extends Thread
{
    public void run(){
        for(int i=0;i<10;i++){
            Object o=queue.consume();
            .... }
    }
}
```

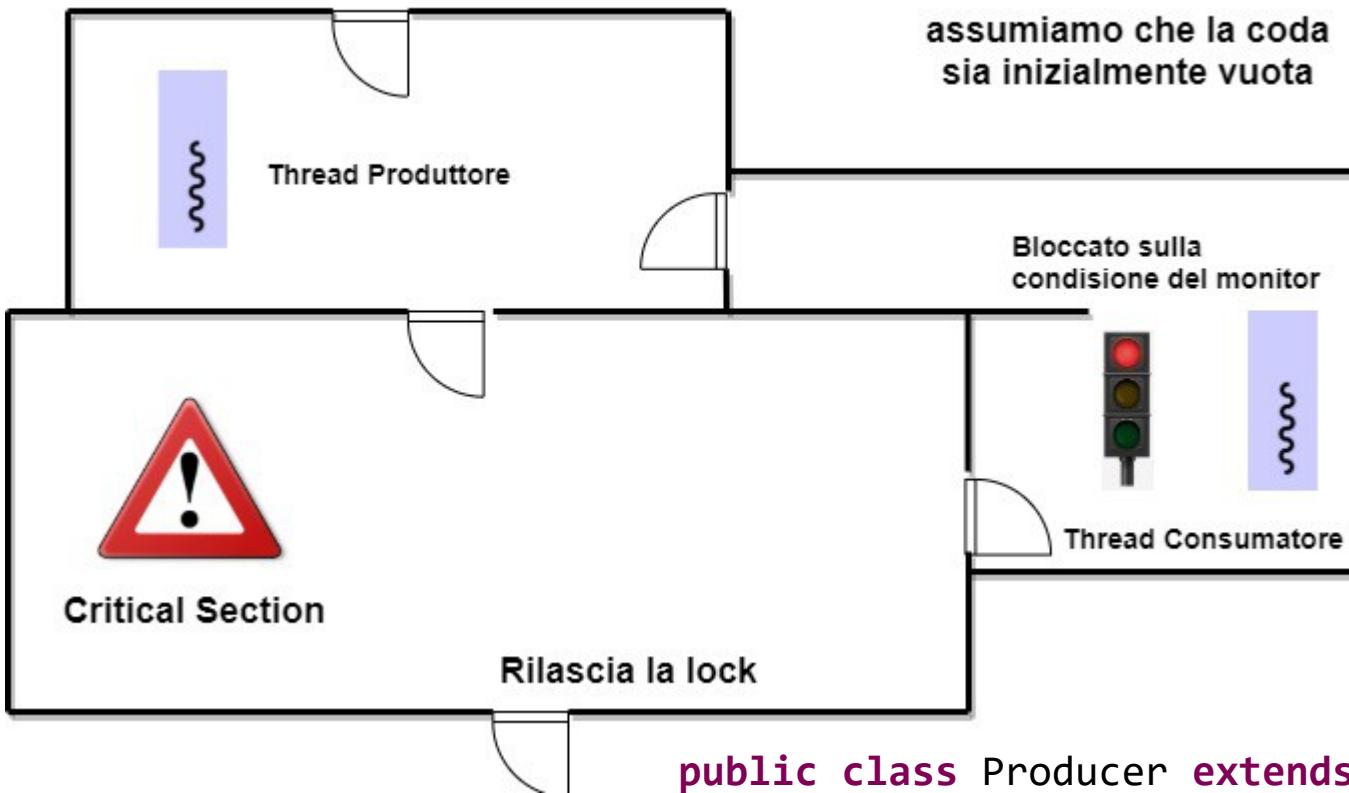
# PRODUTTORE CONSUMATORE “ILLUSTRATO”



```
public class Consumer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++){  
            Object o=queue.consume();  
            .... }  
    }  
}
```

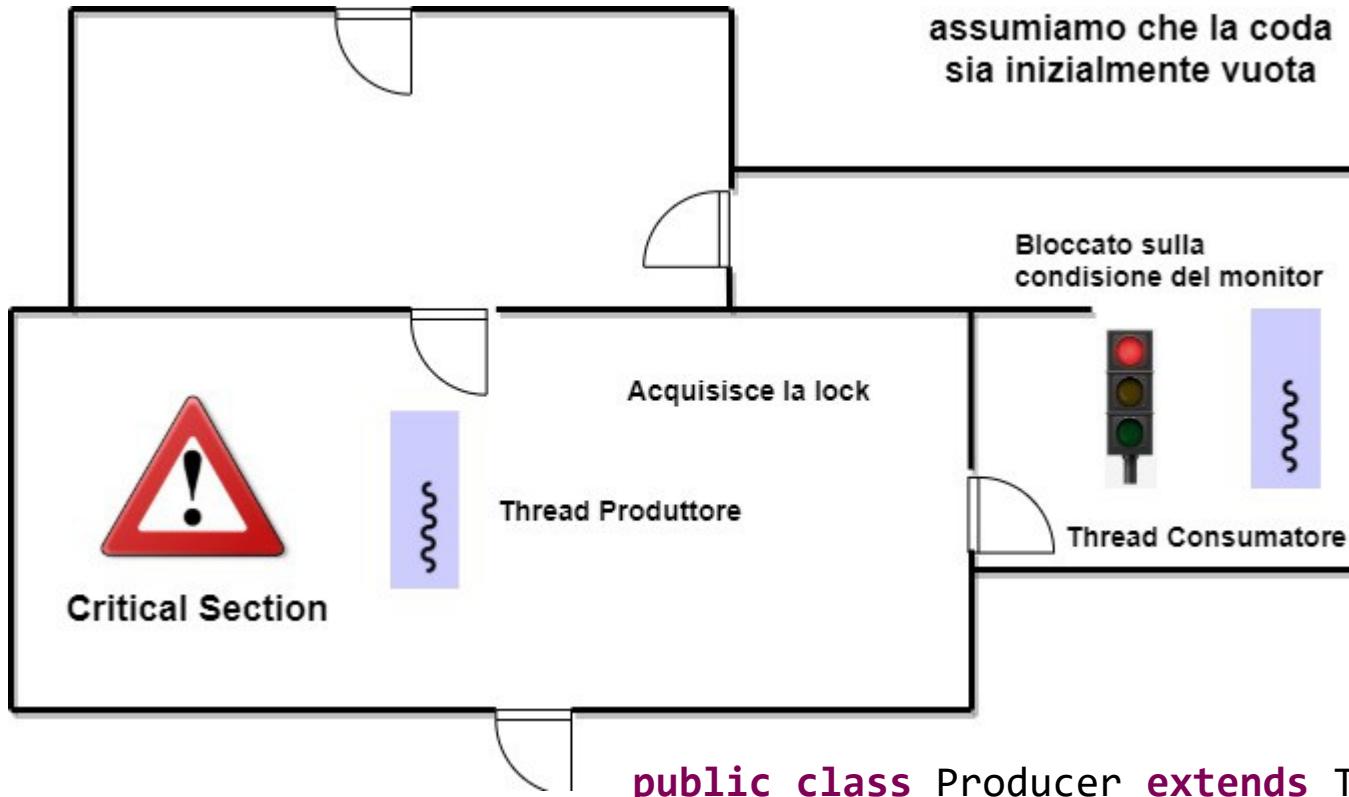
# PRODUTTORE CONSUMATORE “ILLUSTRATO”

## 1. Entra nell'oggetto monitor



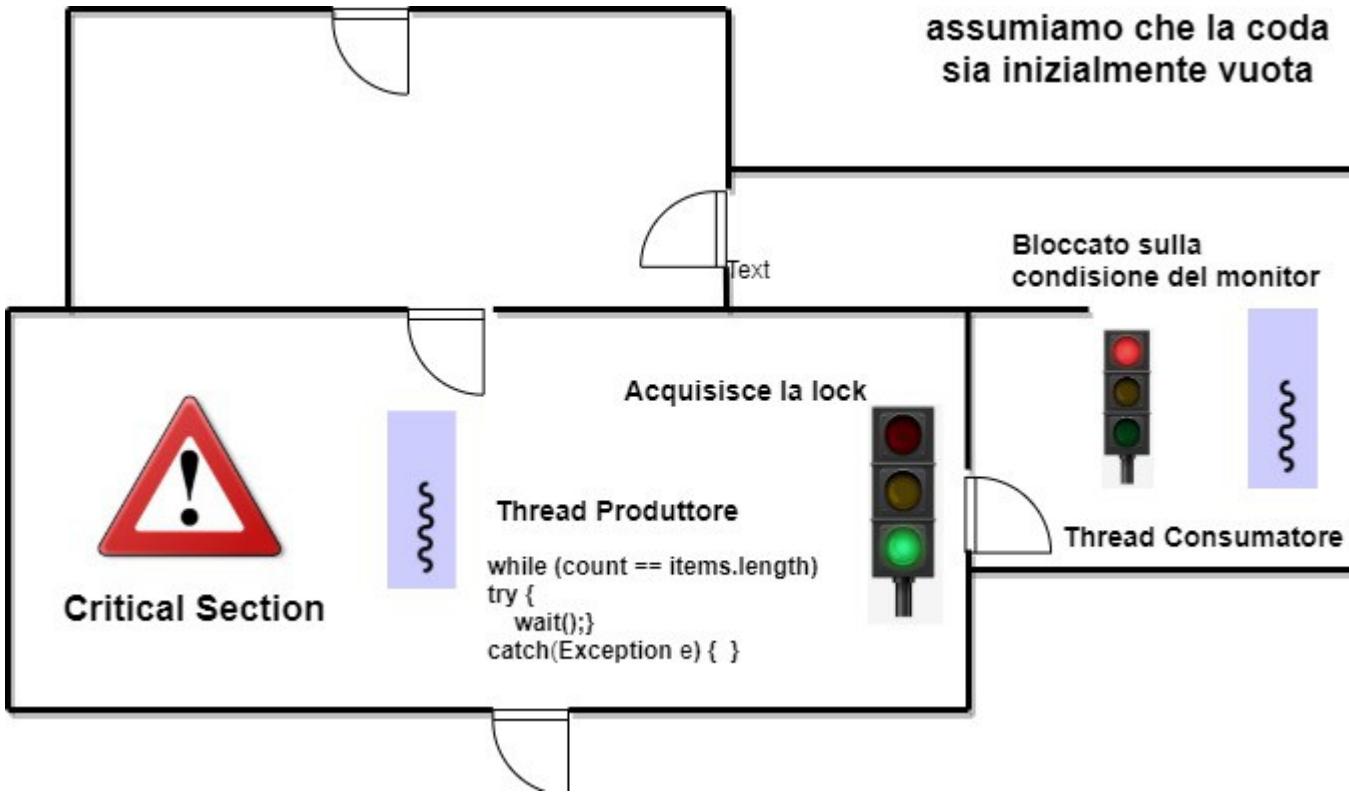
```
public class Producer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++)  
        {queue.produce("MSG#"+count+Thread.currentThread());  
        .....  
    }  
}
```

# PRODUTTORE CONSUMATORE “ILLUSTRATO”



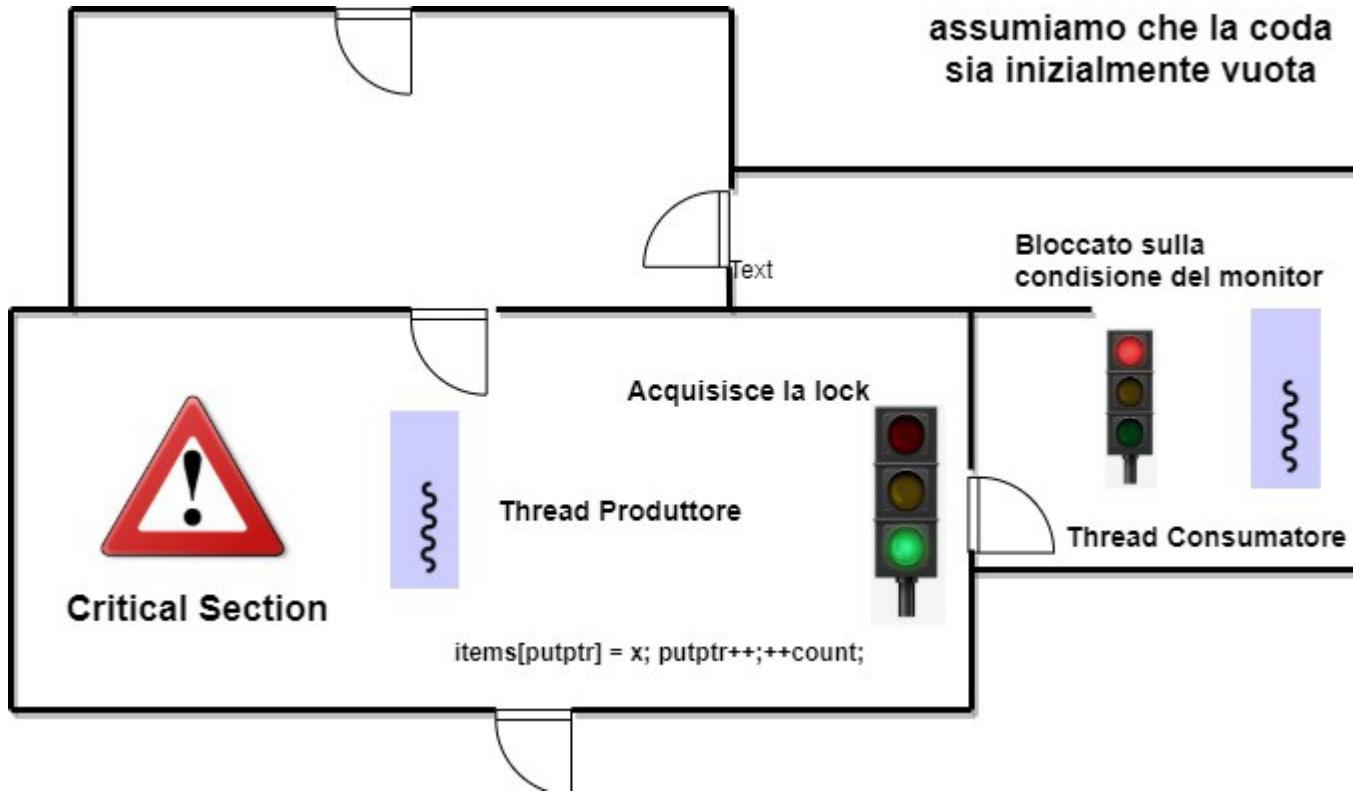
```
public class Producer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++)  
        {queue.produce("MSG#"+count+Thread.currentThread());  
        .....  
    }  
}
```

# PRODUTTORE CONSUMATORE “ILLUSTRATO”



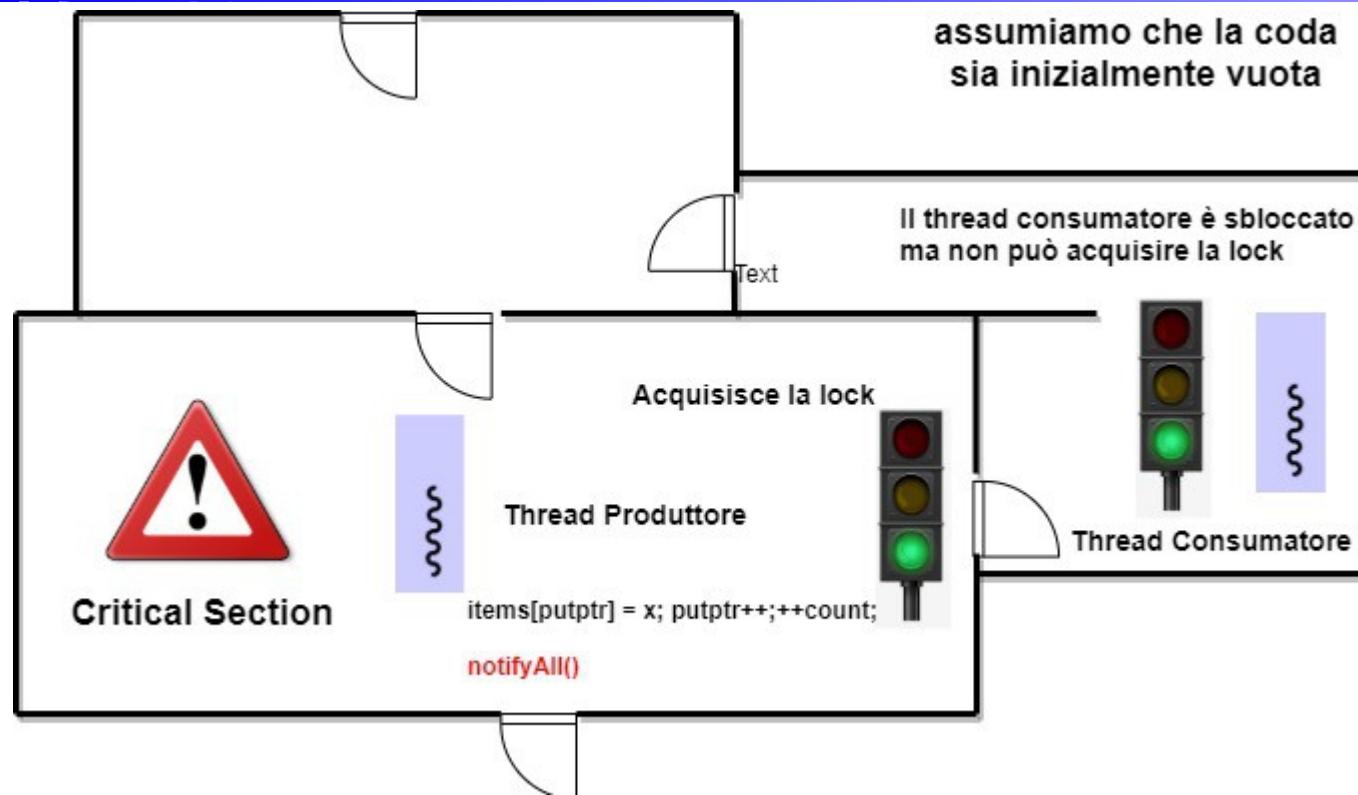
```
public class Producer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++)  
        {queue.produce("MSG#" + count + Thread.currentThread())  
        .....  
    }  
}
```

# PRODUTTORE CONSUMATORE “ILLUSTRATO”



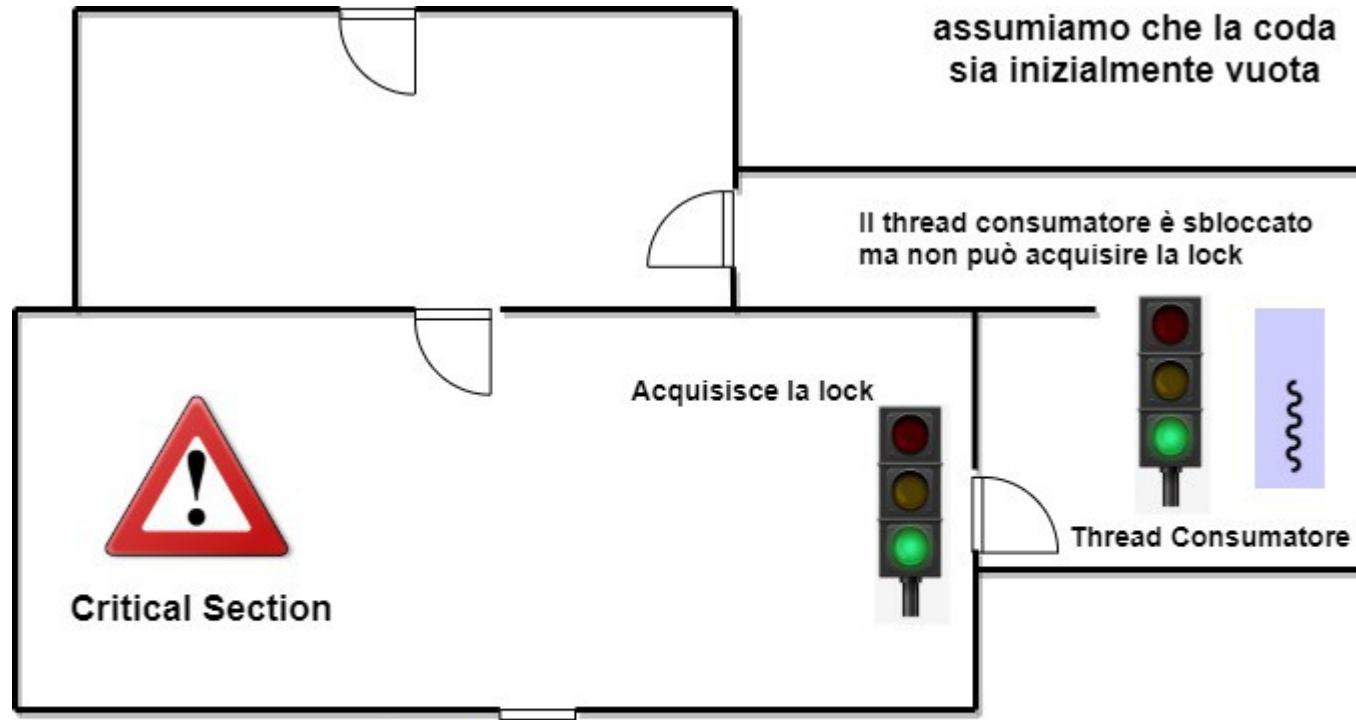
```
public class Producer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++)  
        {queue.produce("MSG#"+count+Thread.currentThread()  
        ....  
    }  
}
```

# PRODUTTORE CONSUMATORE “ILLUSTRATO”



```
public class Producer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++)  
        {queue.produce("MSG#"+count+Thread.currentThread()  
        .....  
    }  
}
```

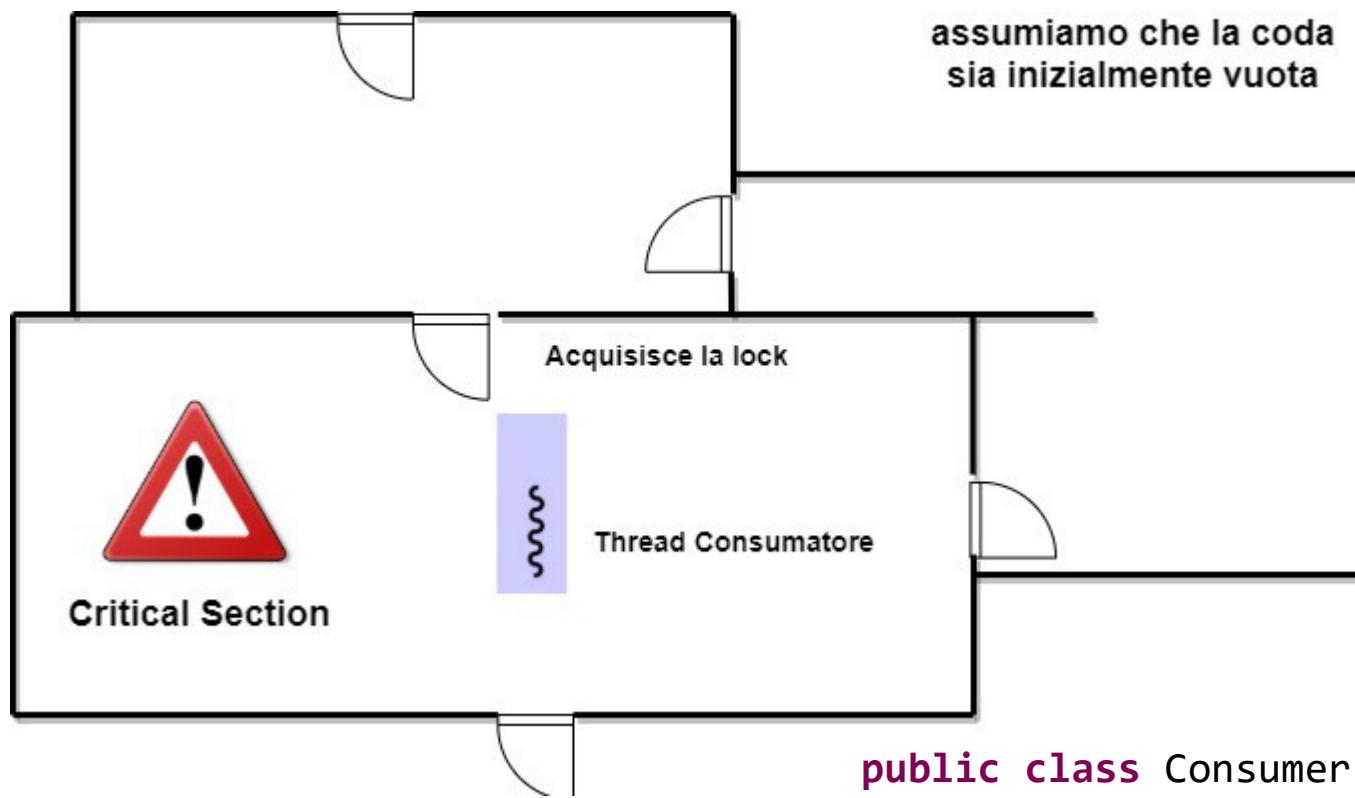
# PRODUTTORE CONSUMATORE “ILLUSTRATO”



Lascia l'oggetto monitor

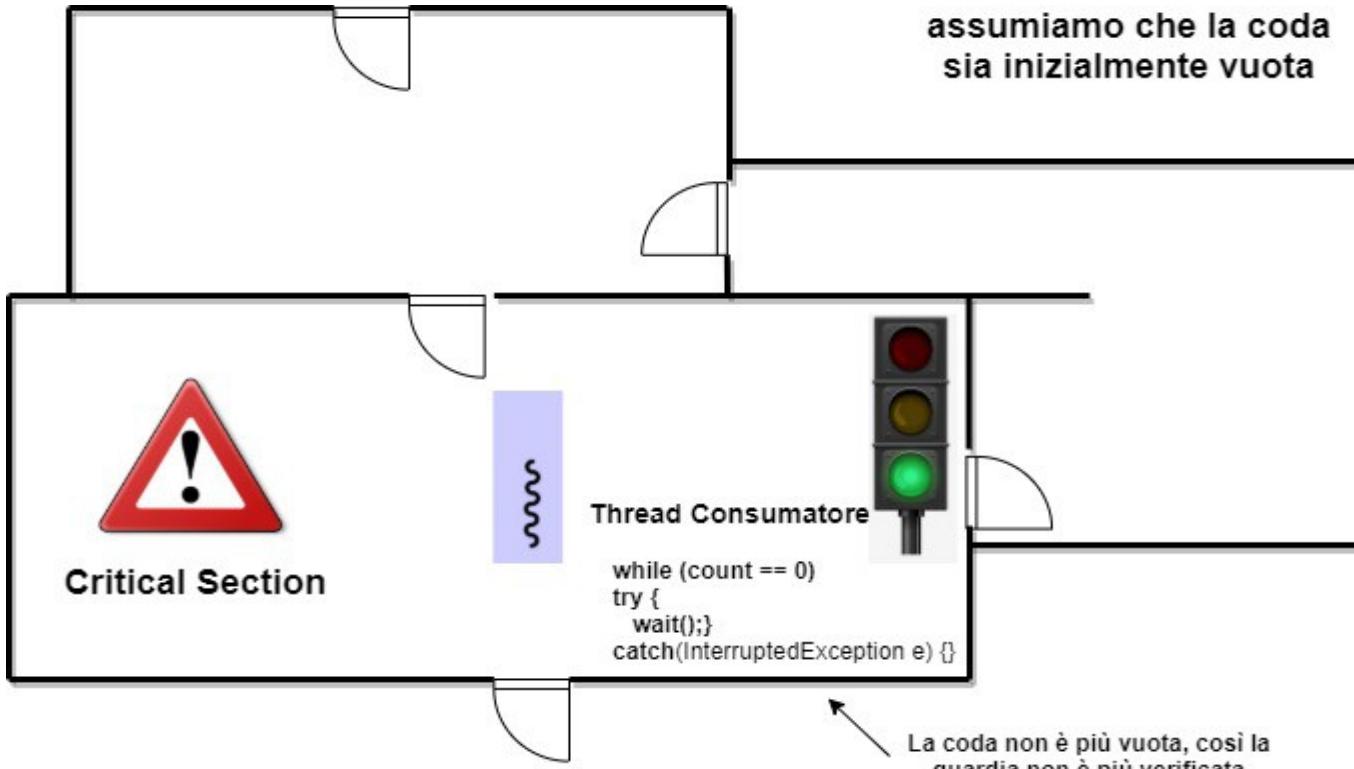
```
public class Producer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++)  
        {queue.produce("MSG#"+count+Thread.currentThread())  
        .....  
    }  
}
```

# PRODUTTORE CONSUMATORE “ILLUSTRATO”



```
public class Consumer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++){  
            Object o=queue.consume();  
            .... }  
    }  
}
```

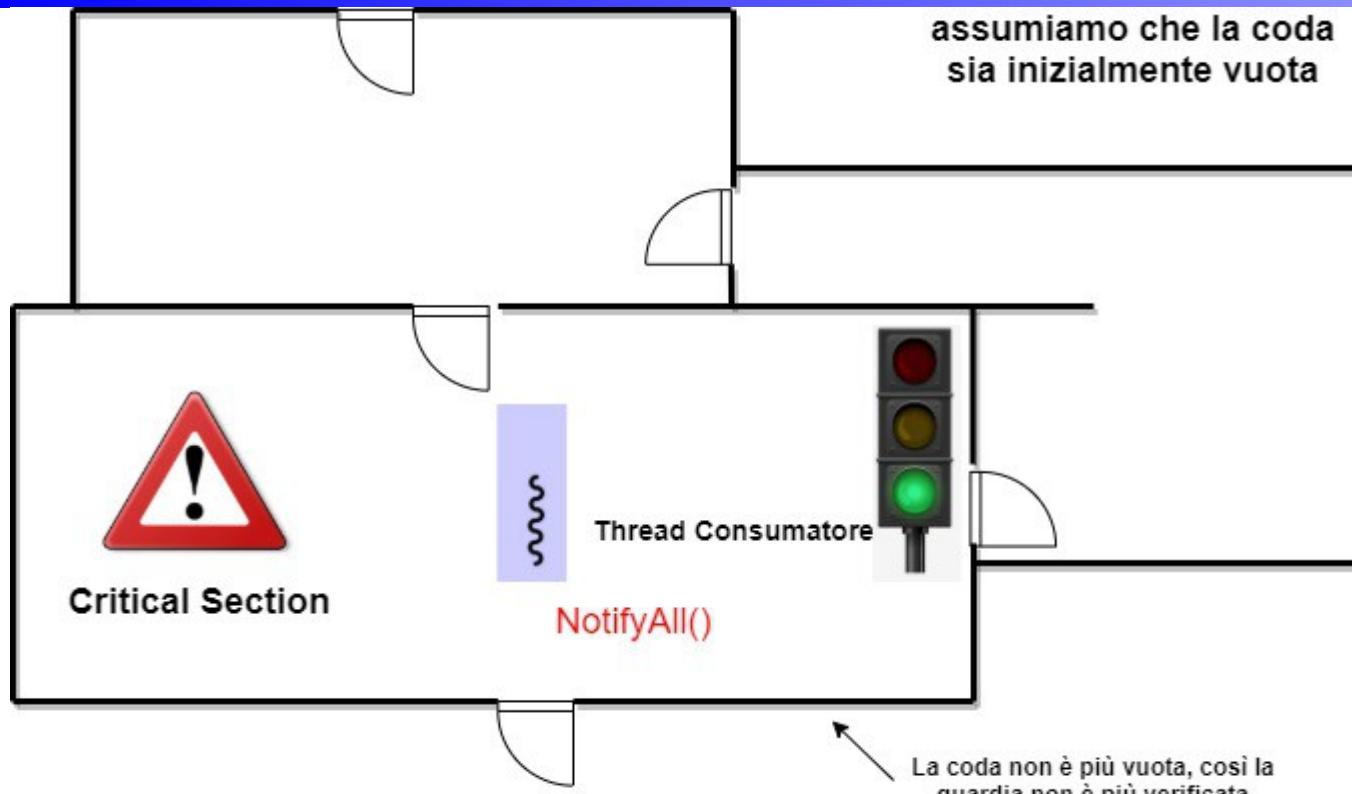
# PRODUTTORE CONSUMATORE “ILLUSTRATO”



La coda non è più vuota, così la guardia non è più verificata

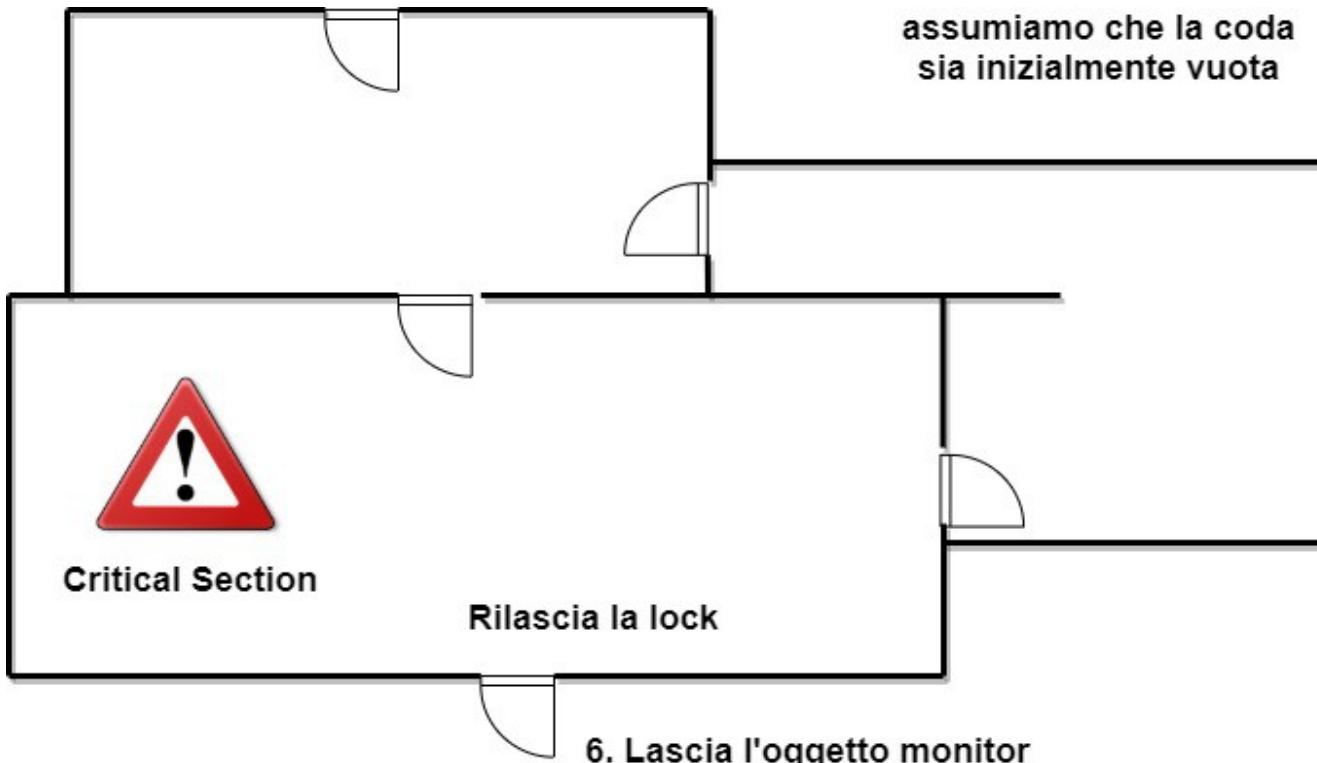
```
public class Consumer extends Thread
{
    public void run(){
        for(int i=0;i<10;i++){
            Object o=queue.consume();
            ....}
    }
}
```

# PRODUTTORE CONSUMATORE “ILLUSTRATO”



```
public class Consumer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++){  
            Object o=queue.consume();  
            .... }  
    }  
}
```

# PRODUTTORE CONSUMATORE “ILLUSTRATO”



```
public class Consumer extends Thread  
{  
    public void run(){  
        for(int i=0;i<10;i++){  
            Object o=queue.consume();  
            .... }  
    }  
}
```

# ASSIGNMENT 3: GESTIONE LABORATORIO

Il laboratorio di Informatica del Polo Marzotto è utilizzato da tre tipi di utenti, studenti, tesisti e professori ed ogni utente deve fare una richiesta al tutor per accedere al laboratorio. I computers del laboratorio sono numerati da 1 a 20. Le richieste di accesso sono diverse a seconda del tipo dell'utente:

- a) i professori accedono in modo esclusivo a tutto il laboratorio, poichè hanno necessità di utilizzare tutti i computers per effettuare prove in rete.
- b) i tesisti richiedono l'uso esclusivo di un solo computer, identificato dall'indice  $i$ , poichè su quel computer è installato un particolare software necessario per lo sviluppo della tesi.
- c) gli studenti richiedono l'uso esclusivo di un qualsiasi computer.

I professori hanno priorità su tutti nell'accesso al laboratorio, i tesisti hanno priorità sugli studenti.

Nessuno però può essere interrotto mentre sta usando un computer (prosegue nella pagina successiva)



# ASSIGNMENT 3: GESTIONE LABORATORIO

Scrivere un programma JAVA che simuli il comportamento degli utenti e del tutor. Il programma riceve in ingresso il numero di studenti, tesisti e professori che utilizzano il laboratorio ed attiva un thread per ogni utente. Ogni utente accede k volte al laboratorio, con k generato casualmente. Simulare l'intervallo di tempo che intercorre tra un accesso ed il successivo e l'intervallo di permanenza in laboratorio mediante il metodo sleep della classe Thread. Il tutor deve coordinare gli accessi al laboratorio. Il programma deve terminare quando tutti gli utenti hanno completato i loro accessi al laboratorio.

Simulare gli utenti con dei thread e incapsulare la logica di gestione del laboratorio all'interno di un monitor.



# Reti e Laboratorio III

## Modulo Laboratorio III

**AA. 2022-2023**

**docente: Laura Ricci**

**[laura.ricci@unipi.it](mailto:laura.ricci@unipi.it)**

## **Lezione 4**

# **Synchronized and Concurrent Collections**

**06/10/2022**

# ANCORA SUL MONITOR: WAIT E NOTIFY

```
public synchronized void act()
    throws InterruptedException
{
    while (!cond) wait();
    // modify monitor data
    notifyAll()
}
```

“regola d'oro” testare sempre la condizione relativa al wait all'interno di un ciclo

- poichè la coda di attesa è unica per tutte le condizioni, un thread potrebbe essere stato risvegliato in seguito al verificarsi di una condizione che poi diventa nuovamente falsa
  - la condizione su cui il thread T è in attesa si è verificata
  - però un altro thread la ha resa di nuovo invalida, dopo che T è stato risvegliato
- il ciclo può essere evitato solo se si è sicuri che questo non accada

# LOCK INTRINSECHE: BLOCCHI SINCRONIZZATI

- se non si intende sincronizzare un intero metodo, si può **sincronizzare un blocco di codice** all'interno di un metodo
- un esempio semplice:

```
class Program {  
    public void foo() {  
        synchronized(this){  
            ...} } }  
}
```

- sincronizzare un intero metodo equivale ad inserire il codice del metodo di un blocco sincronizzato su this
  - l'oggetto riferito tra parentesi è un “monitor object”
  - un thread
  - acquisisce la lock implicita sull'oggetto this, quando entra nel blocco sincronizzato
  - la rilascia quando termina il blocco sincronizzato.



# WAIT/NOTIFY E BLOCCHI SINCRONIZZATI

- attendere del verificarsi di una condizione su un oggetto diverso da `this`

```
synchronized (obj)  
    while (!condition)  
        {try {obj.wait();}  
         catch (InterruptedException ex){...}}
```

- segnalare una condizione

```
synchronized(obj){  
    condition=.....;  
    obj.notifyAll();}
```

- ne vedremo un uso concreto nel caso di classi conditionally thread safe

# MONITOR E LOCK: CONFRONTI

- monitor: vantaggi
  - l'unità di sincronizzazione è il metodo: tutte le sincronizzazioni sono visibili esaminando segnatura dei metodi
  - costrutti strutturati. diminuisce la complessità del programma concorrente: deadlocks, mancato rilascio di lock, maggior manutenibilità del software
- monitor: svantaggi: “coarse grain” synchronization, per-object synchronization, può diminuire il livello di concorrenza
- lock esplicite, vantaggi:
  - maggior numero di funzioni disponibili, maggiore flessibilità
    - tryLock() il thread prova ad acquisire una lock, ma non mi blocca
    - read/write locks: multiple reader single writer
- lock esplicite, svantaggi: codice poco leggibile, se usate in modo non strutturato



# JAVA COLLECTION FRAMEWORK: BREVE RIPASSO

- un insieme di classi che consentono di lavorare con gruppi di oggetti, ovvero collezioni di oggetti
  - classi contenitore
  - introdotte a partire dalla release 1.2
  - contenute nel package `java.util`
  - rivedere le implementazioni delle collezioni più importanti con lo scopo di utilizzare nel progetto le strutture dati più adeguate
- che c'è di nuovo in questo corso?
  - **synchronized collections**
  - **concurrent collections**

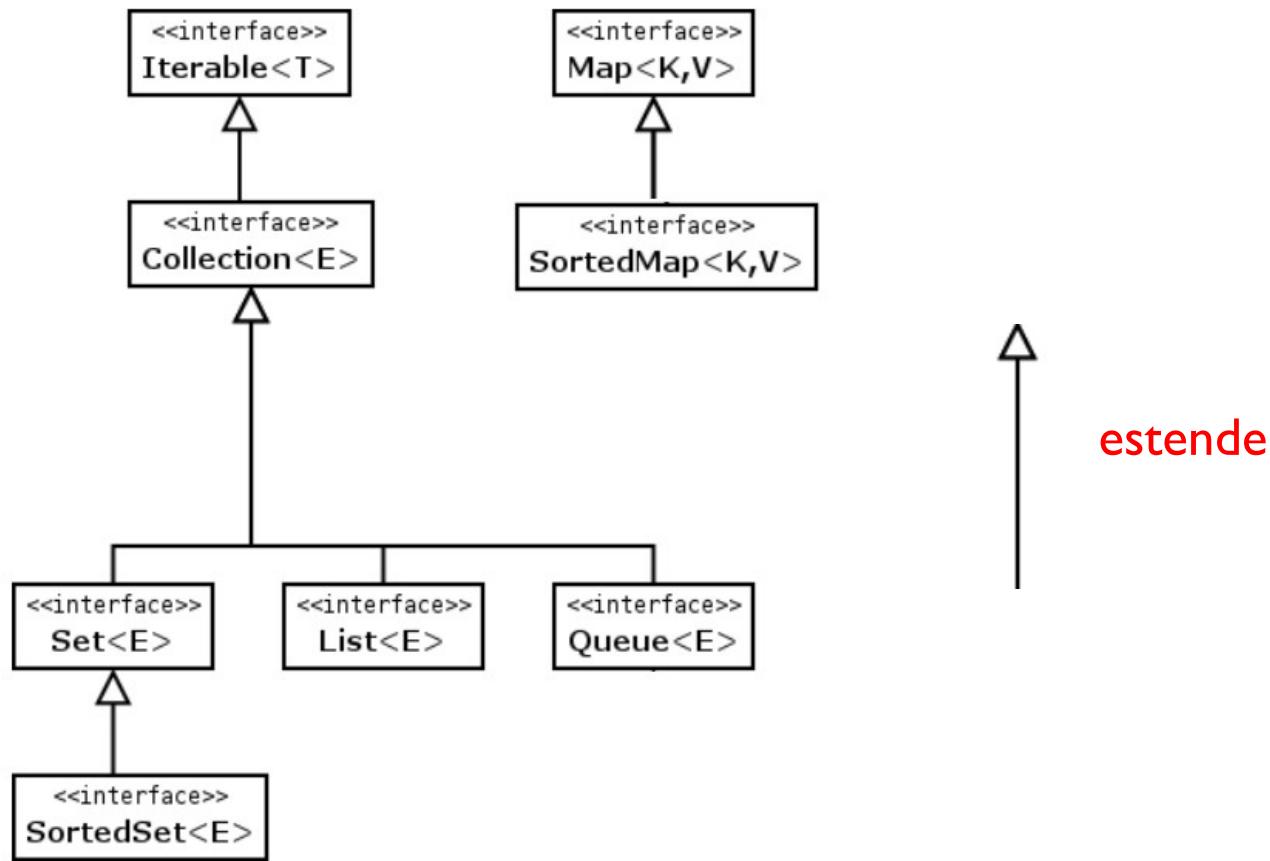


# JAVA COLLECTION FRAMEWORK: BREVE RIPASSO

- un insieme di classi che consentono di lavorare con gruppi di oggetti, ovvero collezioni di oggetti
  - classi contenitore
  - introdotte a partire dalla release 1.2
  - contenute nel package `java.util`
  - rivedere le implementazioni delle collezioni più importanti con lo scopo di utilizzare nel progetto le strutture dati più adeguate
- che c'è di nuovo in questo corso?
  - **synchronized collections**
  - **concurrent collections**



# JAVA COLLECTION: INTERFACES IN JAVA.UTIL

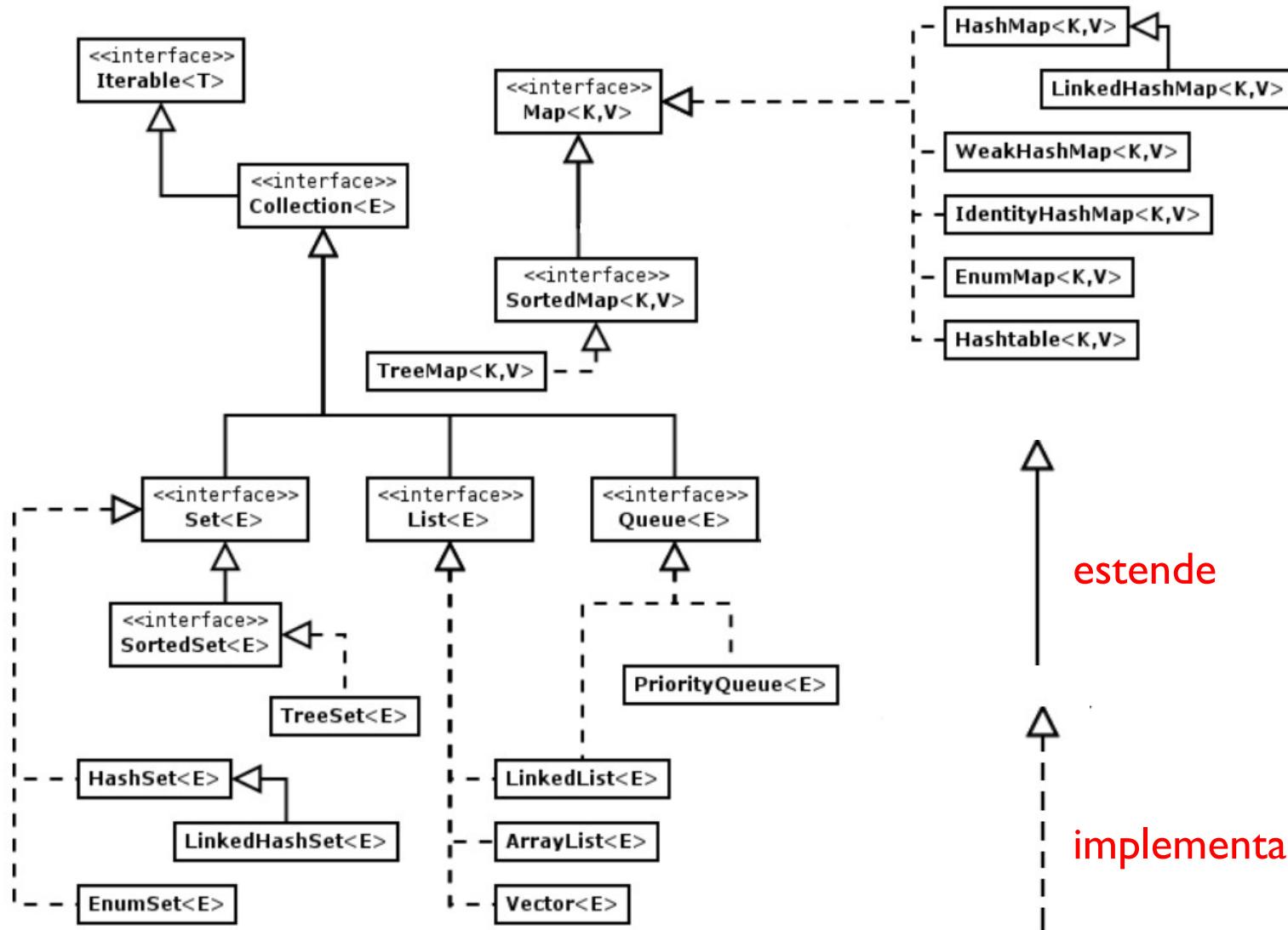


- riportate solo le interfacce principali presenti in `java.util`

tre strutture principali per rappresentare una collezione di valori, rappresentate da altrettante interfacce

- **list**: una collezione ordinata (una sequenza) di valori; possono esistere duplicati
- **set**: una collezione dove ciascun valore appare una sola volta: non ci sono duplicati, e, in generale, i valori non sono ordinati
  - prevista però una interfaccia in cui i valori possono essere ordinati
- **map**: una collezione in cui vi è un mapping da chiavi a valori. Le chiavi sono uniche
  - le chiavi possono essere ordinate

# JAVA COLLECTION: INTERFACES AND CLASSES



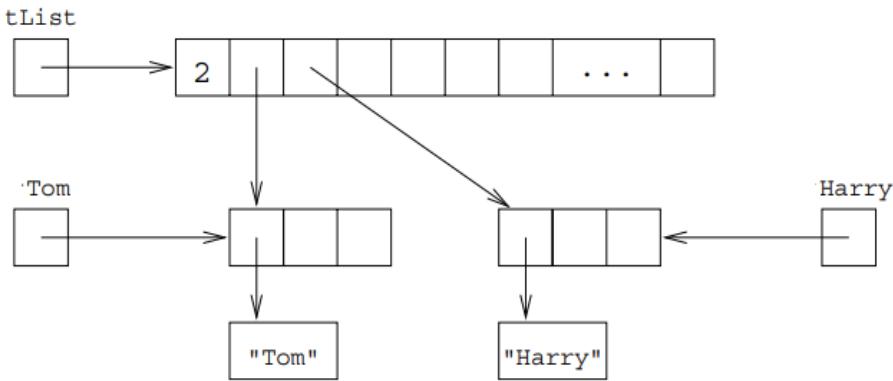
# DISTRICARSI NELLA GIUNGLA DELLE CLASSI

- interfaccia: [Map](#)
  - [HashMap](#) (implementazione di Map) non è un'implementazione di [Collection](#), ma è comunque una struttura dati molto usata
  - realizza una struttura dati “dizionario” che associa termini chiave (univoci) a valori
- [Collections](#) (con la 's' finale !) contiene metodi utili per l'elaborazione di collezioni di qualunque tipo:
  - ordinamento
  - calcolo di massimo e minimo
  - rovesciamento, permutazione, riempimento di una collezione
  - confronto tra collezioni (elementi in comune, sottocollezioni, ...)
  - aggiungere un wrapper di sincronizzazione ad una collezione



# JAVA COLLECTION FRAMEWORK: ARRAYLIST

```
public class Person {  
    String name;  
    int age;  
    String type;  
  
    public Person (String name, int age, String type)  
    {this.name=name;  
     this.age=age;  
     this.type=type;}  
  
    public String toString ( ) {  
        return this.name+this.age+this.type;}  
}
```

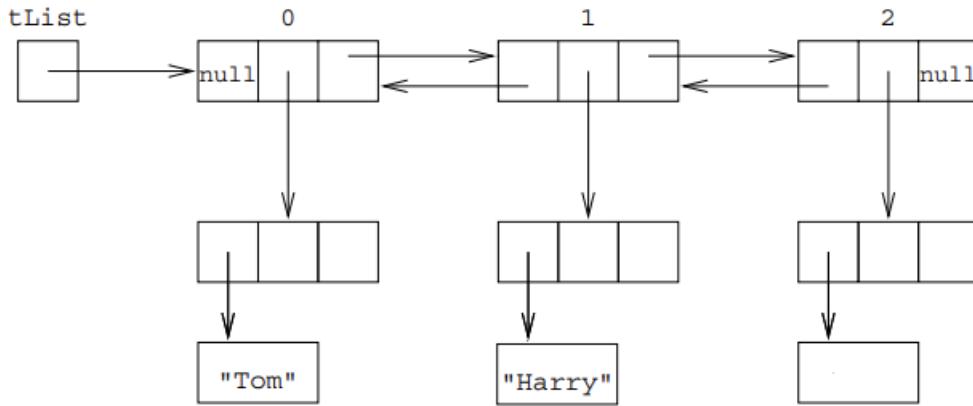


```
import java.util.*;  
  
public class PersonList {  
  
    public static void main (String args[])  
    { Person Tom = new Person("Tom", 45, "professor");  
        Person Harry = new Person("Harry", 20, "student");  
  
        List <Person> tList=new  
            ArrayList<Person> ();  
  
        tList.add(Tom);  
        tList.add(Harry);  
        System.out.println(Tom);  
        System.out.println(Harry);  
        System.out.println(pList.size()); } }
```

# JAVA COLLECTION FRAMEWORK: LINKEDLIST

```
public class Person {  
  
    String name;  
  
    int age;  
  
    String type;  
  
    public Person (String name, int age, String type)  
    {this.name=name;  
  
     this.age=age;  
  
     this.type=type;}  
  
    public String toString ( ) {  
  
        return this.name+this.age+this.type;}  
  
    }  

```



```
import java.util.*;  
  
public class PersonList {  
  
    public static void main (String args[])  
    { Person Tom = new Person("Tom", 45,"professor");  
  
        Person Harry = new Person ("Harry", 20,"student");  
  
        List <Person> tList=new  
                           LinkedList<Person> ();  
  
        tList.add(Tom);  
  
        tList.add(Harry);  
  
        System.out.println(Tom);  
  
        System.out.println(Harry);  
  
        System.out.println(tList.size());    } }  

```

# JAVA ITERATORS

- GoF “Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation”
- usato per accedere agli elementi di una collezione, uno alla volta
  - deve conoscere (e poter accedere) alla rappresentazione interna della classe che implementa la collezione
- l’interfaccia `Collection` contiene il metodo `iterator()` che restituisce un iteratore per una collezione
  - le diverse implementazioni di `Collection` implementano il metodo `iterator()` in modo diverso
  - l’interfaccia `Iterator` prevede tutti i metodi necessari per usare un iteratore, senza conoscere alcun dettaglio implementativo



# USARE GLI ITERATORI

- schema generale per l'uso di un iteratore

```
import java.util.*;  
  
public class PersonList {  
  
    public static void main (String args[])  
    { Person Tom = new Person("Tom", 45, "professor");  
        Person Harry = new Person("Harry", 20, "student");  
        List <Person> pList=new LinkedList<Person> ();  
        ....  
        Iterator <Person> tIterator = pList.iterator();  
        while (tIterator.hasNext())  
        { Person tPerson = (Person) tIterator.next();  
            System.out.println(tPerson);  
        } } }
```

- l'iteratore non ha alcuna funzione che lo “resetti”
- una volta iniziata la scansione, non si può fare tornare indietro l'iteratore
- una volta finita la scansione, è necessario creare uno nuovo iteratore



# USARE GLI ITERATORI SU HASHMAP

```
public class Employee{  
    private String id;  
    private String name;  
    private String department;  
  
    public Employee(String id, String name, String department) {  
        this.id = id;  
        this.name = name;  
        this.department = department;  
    }  
  
    public String toString() {  
        return "[" + this.id + " : " + this.name + " : " + this.department +  
    "]"; } }
```



# USARE GLI ITERATORI SU HASHMAP

```
import java.util.*;

public class EmployeeIterator {

public static void main (String args[])

{ HashMap<String, Employee> employeeMap = new HashMap<String, Employee>();

employeeMap.put("emp01", new Employee("emp01", "Tom", "IT"));

employeeMap.put("emp02", new Employee("emp02", "Jhon", "Supply Chain"));

employeeMap.put("emp03", new Employee("emp03", "Oliver", "Marketing"));

employeeMap.put("emp04", new Employee("emp04", "Mary", "IT"));

Set<Map.Entry<String, Employee>> entrySet = employeeMap.entrySet();

Iterator<Map.Entry<String, Employee>> iterator = entrySet.iterator();

System.out.println("Iterate through mappings of HashMap");

while( iterator.hasNext() ){

    Map.Entry<String, Employee> entry = iterator.next();

    System.out.println( entry.getKey() + " => " + entry.getValue() ); }}}
```



- sviluppato in parte da Doug Lea
  - disponibile per tre anni come insieme di librerie JAVA non standard
  - quindi integrazione in JAVA 5.0
- tra i package principali, in rosso alcuni argomenti di questa e della prossima lezione
  - `java.util.concurrent`
    - Executor, concurrent collections, semaphores,...
  - `java.util.concurrent.atomic`
    - AtomicBoolean, AtomicInteger,...
  - `java.util.concurrent.locks`
    - Condition
    - Lock
    - ReadWriteLock

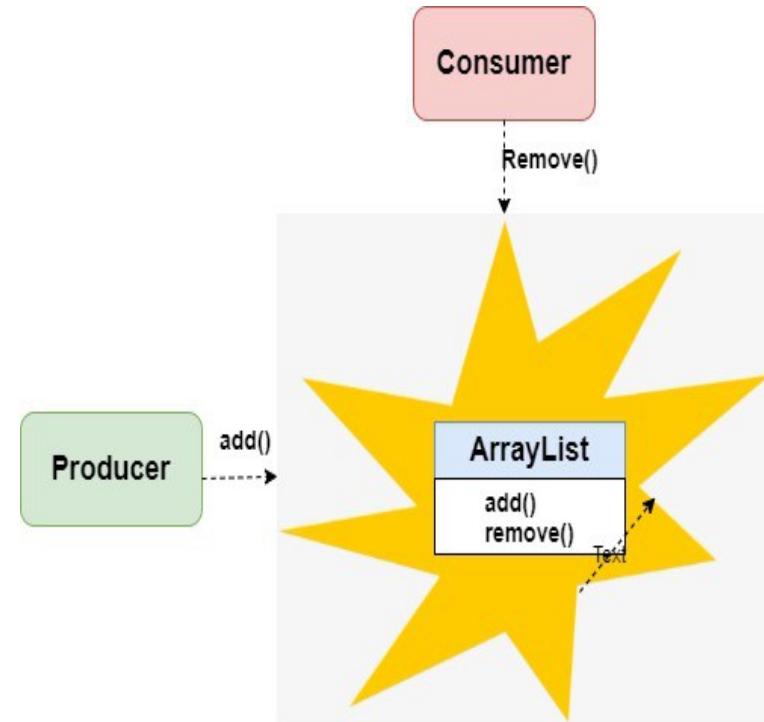
# JAVA COLLECTIONS E THREAD SAFENESS

- collezioni non thread safe, non offrono alcun supporto per la sincronizzazione dei threads
  - `java.util.Map`
  - `Java.util.LinkedList`
  - `java.util.ArrayList`
- quali soluzioni per la sincronizzazione? Alternative possibili
  - thread safe collections, sincronizzate automaticamente da JAVA
    - `java.util.Vector`
    - `java.util.Hashtable`
  - synchronized collections
  - concurrent collections
    - introdotte in `java.util.concurrent`



# COLLEZIONI NON THREAD SAFE

- ArrayList non è una classe threadsafe
  - add non è una operazione atomica
    - determina quanti elementi ci sono nella lista
    - determina il punto esatto del nuovo elemento
    - incrementa il numero di elementi della lista
    - se si eseguono due add in modo concorrente lo stato della struttura può essere inconsistente
  - analogamente per la remove
  - Vector è una classe threadsafe: *ma quanto costa la sincronizzazione?*



# VECTOR ED ARRAYLIST: “UNDER THE HOOD”

```
import java.util.ArrayList;
import java.util.List;
import java.util.Vector;
public class VectorArrayList {
    public static void addElements(List<Integer> list)
    {for (int i=0; i< 1000000; i++)
     {list.add(i);}}
    public static void main (String args[])
    {
        final long start1 =System.nanoTime();
        addElements(new Vector<Integer>());
        final long end1=System.nanoTime();
        final long start2 =System.nanoTime();
        addElements(new ArrayList<Integer>());
        final long end2=System.nanoTime();
        System.out.println("Vector time "+ (end1-start1));
        System.out.println("ArrayList time "+ (end2-start2)); }}
```

Vector time 74494150  
ArrayList time 48190559



# SYNCHRONIZED COLLECTIONS

- synchronized collection wrappers
  - “incapsulano” ogni metodo in un blocco sincronizzato
  - trasformano una Collection non thread safe in una **thread-safe**
  - utilizzano un'unica “mutual exclusion lock” intrinseca per tutta la collezione, gestita dalla JVM
- metodi definiti nella interfaccia Collections
- “conditionally thread safe” collections

Collections Method
<code>synchronizedCollection(coll)</code>
<code>synchronizedCollection(list)</code>
<code>synchronizedCollection(map)</code>
<code>synchronizedCollection(map)</code>



# SYNCHRONIZED COLLECTIONS: VALUTAZIONE

```
import java.util.ArrayList;
import java.util.List;
import java.util.Collections;
public class VectorArrayList {
    public static void addElements(List<Integer> list)
    {for (int i=0; i< 1000000; i++)
        {list.add(i);}}
    public static void main (String args[])
    {final long start1 =System.nanoTime();
    addElements(new ArrayList<Integer>());
    final long end1=System.nanoTime();
    final long start2 =System.nanoTime();
    addElements(Collections.synchronizedList(new ArrayList<Integer>()));
    final long end2=System.nanoTime();
    System.out.println("ArrayList time "+(end1-start1));
    System.out.println("SynchronizedArrayList time "+(end2-start2));}}
```

ArrayList	time 50677689
SynchronizedArrayList	time 62055651

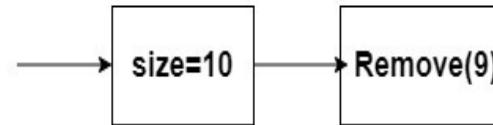


# CLASSI CONDITIONALLY THREAD SAFE

```
public class UnsafeVector{  
    public static <T> T getLast (Vector<T> list){  
        int lastIndex = list.size() - 1;  
        return (list.get(lastIndex)); } }
```



```
public static void deleteLast (Vector<T> list){  
    int lastIndex = list.size() - 1;  
    list.remove(lastIndex); } }
```



- la thread safety garantisce che la safety delle singole operazioni operazioni sulla collezione, ma...
- funzioni che **coinvolgono più di una operazione** possono non essere thread-safe
- Vector è una collezione thread-safe, però perchè in caso di accessi concorrenti, questo programma genera ArrayIndexOutOfBoundsException?

# CLASSI CONDITIONALLY THREAD SAFE

```
if(!synchList.isEmpty())  
    synchList.remove(0);
```

- `isEmpty()` e `remove()` sono entrambe operazioni atomiche, ma la loro combinazione non lo è.
- scenario di errore:
  - una lista con un solo elemento.
  - il primo thread verifica che la lista non è vuota e viene deschedulato prima di rimuovere l'elemento.
  - un secondo thread rimuove l'elemento, il primo thread torna in esecuzione e prova a rimuovere un elemento non esistente
- Java Synchronized Collections sono **conditionally thread-safe**.
  - le operazioni individuali sulle collezioni sono safe, ma funzioni composte da più di una operazione singola possono risultarla.



# USO DEI BLOCCHI SINCRONIZZATI

- richiesta una sincronizzazione esplicita da parte del programmatore per sincronizzare una sequenza di operazioni

```
synchronized(synchList) {  
    if(!synchList.isEmpty())  
        synchList.remove(0);  
}
```

- tipico esempio di utilizzo di **blocchi sincronizzati**
- il thread che esegue l'operazione composta acquisisce la lock sulla struttura `synchList` più di una volta:
  - quando esegue il blocco sincronizzato
  - quando esegue i metodi della collezione

ma...il comportamento corretto è garantito perchè le lock sono rientranti



# USO DEI BLOCCHI SINCRONIZZATI

@ThreadSafe

```
public class UnsafeVector{  
    public static <T> T getLast (Vector<T> list) {  
        synchronized (list)  
        {  
            int lastIndex = list.size() - 1;  
            return (list.get(lastIndex));  
        }  
    }  
    public static void deleteLast (Vector<T> list) {  
        synchronized (list)  
        {  
            int lastIndex = list.size() - 1;  
            list.remove(lastIndex);  
        }  
    }  
}
```



# ITERATORI E ECCEZIONI

- eccezione sollevata dagli iteratori su collezioni, se la collezione viene modificata prima che l'iterazione sia completata

- può essere sollevata anche se il programma è sequenziale

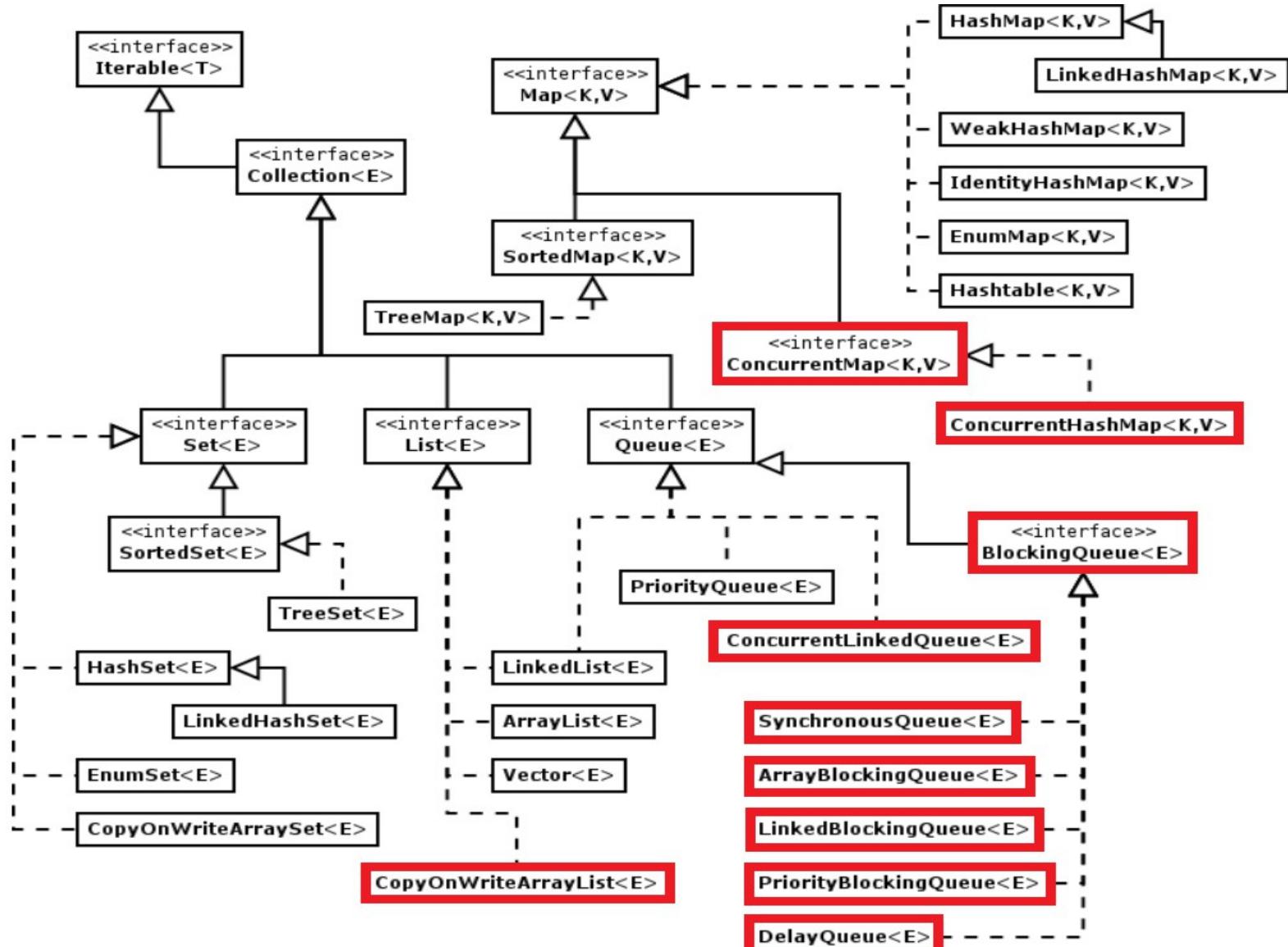
```
for (E element: list)  
    if (isBad(element))  
        list.remove(element) //ConcurrentModificationException
```

- anche se la collezione è sincronizzata, l'iteratore su di essa può non esserlo

```
synchronized(syncList) {  
    Iterator iterator = syncList.iterator();  
    // do stuff with the iterator here  
}
```



# CONCURRENT COLLECTIONS (IN ROSSO)



# CONCURRENT COLLECTIONS

- evoluzione delle precedenti librerie basata sulla esperienza nel loro utilizzo
- fine-grain locking, non bloccano l'intera collezione
  - concurrent reads, writes parzialmente concorrenti
- iteratori fail safe/weakly consistent
  - restituiscono tutti gli elementi che erano presenti nella collezione quando l'iteratore è stato creato
  - possono restituire o meno elementi aggiunti in concorrenza
- forniscono alcune utili operazioni atomiche composte da più operazioni elementari
- Blocking Queue è una concurrent collections
- la più utilizzata: ConcurrentHashMap <K,V>, sincronizzazione ottimizzata, diverse operazioni atomiche
  - put-if-absent, remove-if-equal, replace-if-equal



# HASH MAP E HASH TABLE

- ConcurrentHashMap
  - introdotta in JAVA 5, nella libreria `java.util.concurrent`
  - thread safe, è una evoluzione di HashTable ed HashMap
- collezioni che memorizzano coppie chiave/valore
  - usando la tecnica dell'hashing
- differenza principale
  - HashTable (da JAVA 1.0) è *threadsafe*
  - HashMap (da JAVA 1.2) non è *threadsafe*
  - perchè un'altra collezione thread safe?



# CONCURRENT HASH MAP: MOTIVAZIONE

```
import java.util.Map;  
  
import java.util.concurrent.ConcurrentHashMap;  
  
import java.util.concurrent.ExecutorService;  
  
import java.util.concurrent.Executors;  
  
import java.util.concurrent.TimeUnit;  
  
public class TestCollections {  
  
    public static void main(String[] args) throws Exception {  
  
        evaluatingThePerformance(new Hashtable<String, Integer>(), 5);  
  
        evaluatingThePerformance(Collections.synchronizedMap  
  
            (new HashMap<String, Integer>()), 5);  
  
        evaluatingThePerformanceWithSynchronizedBlock  
  
            (new HashMap<String, Integer>(), 5);  
  
        evaluatingThePerformance  
  
            (new ConcurrentHashMap<String, Integer>(), 5);  
    }  
}
```



# CONCURRENT HASH MAP: MOTIVAZIONE

```
public static void evaluatingThePerformance  
        (Map<String, Integer> maptoEvalPerf, int size) throws Exception {  
  
    long averageTime = 0;  
  
    for(int j=0; j<size;j++) {  
  
        ExecutorService executorService = Executors.newFixedThreadPool(size);  
  
        long startTime = System.nanoTime();  
  
        for(int i=0; i<size;i++) {  
  
            executorService.execute(new Runnable () {public void run()  
                {performTest(mptoEvalPerf);}});}  
  
        executorService.shutdown();  
  
        executorService.awaitTermination(Long.MAX_VALUE, TimeUnit.DAYS);  
  
        long entTime = System.nanoTime();  
  
        long totalTime = (entTime - startTime) / 1000000L;  
  
        averageTime += totalTime;  
  
        System.out.println("500K entried added/retrieved by each thread in "+ totalTime+ " ms");  
    }  
  
    System.out.println("For " + mptoEvalPerf.getClass() + " the average time is " +  
                        averageTime / 5 + "ms\n");} }
```



# CONCURRENT HASH MAP: MOTIVAZIONE

```
public static void performTest(Map<String, Integer> mapToEvaluateThePerformance)
{
    for(int i=0; i<500000; i++) {
        Integer randomNumber = (int) Math.ceil(Math.random() * 550000);
        Integer value =
            mapToEvaluateThePerformance.get(String.valueOf(randomNumber));
        // Put value
        mapToEvaluateThePerformance.put(String.valueOf(randomNumber), randomNumber);
    }
}
```

- test: 500000 put e get sulla tabella passata come parametri
- attiviamo un threadpool con k threads che eseguono in parallelo il codice precedente e prendiamo i tempi di esecuzione
- ripetiamo l'esperimento un certo numero di volte e calcoliamo la media dei tempi impiegati



# CONCURRENT HASH MAP: MOTIVAZIONE

```
public static void performSynchronizedTest(Map<String, Integer>
                                         maptoEvaluateThePerformance) {

    for(int i=0; i<500000; i++) {
        Integer randomNumber = (int) Math.ceil(Math.random() * 550000);
        synchronized (maptoEvaluateThePerformance) {
            Integer Value =
                maptoEvaluateThePerformance.get(String.valueOf(randomNumber));
        }
        synchronized (maptoEvaluateThePerformance) {
            maptoEvaluateThePerformance.put
                (String.valueOf(randomNumber), randomNumber);
        }
    }
}
```



- il metodo `evaluatingThePerformanceWithSynchronizedBlock` invoca `performSynchronizedTest`, invece di `performTest` (nel run della Runnable)
- per il resto è uguale al metodo della slide precedente

# CONCURRENT HASH MAP: MOTIVAZIONE

500K entries added/retrieved by each thread in 1274 ms

500K entries added/retrieved by each thread in 1314 ms

500K entries added/retrieved by each thread in 1279 ms

.....

**For class `java.util.Hashtable` the average time is 1283 ms**

500K entries added/retrieved by each thread in 1288 ms

500K entries added/retrieved by each thread in 1157 ms

500K entries added/retrieved by each thread in 1249 ms

.....

**For class `java.util.Collections$SynchronizedMap` the average time is 1237 ms**

500K entries added/retrieved in 1238 ms

500K entries added/retrieved in 1392 ms

500K entries added/retrieved in 1375 ms

.....

**For class `java.util.HashMapWith SYNCHRONIZED BLOCKS` the average time is 1423 ms**

500K entries added/retrieved by each thread in 550 ms

500K entries added/retrieved by each thread in 440 ms

500K entries added/retrieved by each thread in 458 ms

.....

**For class `java.util.concurrent.ConcurrentHashMap` the average time is 524 ms**

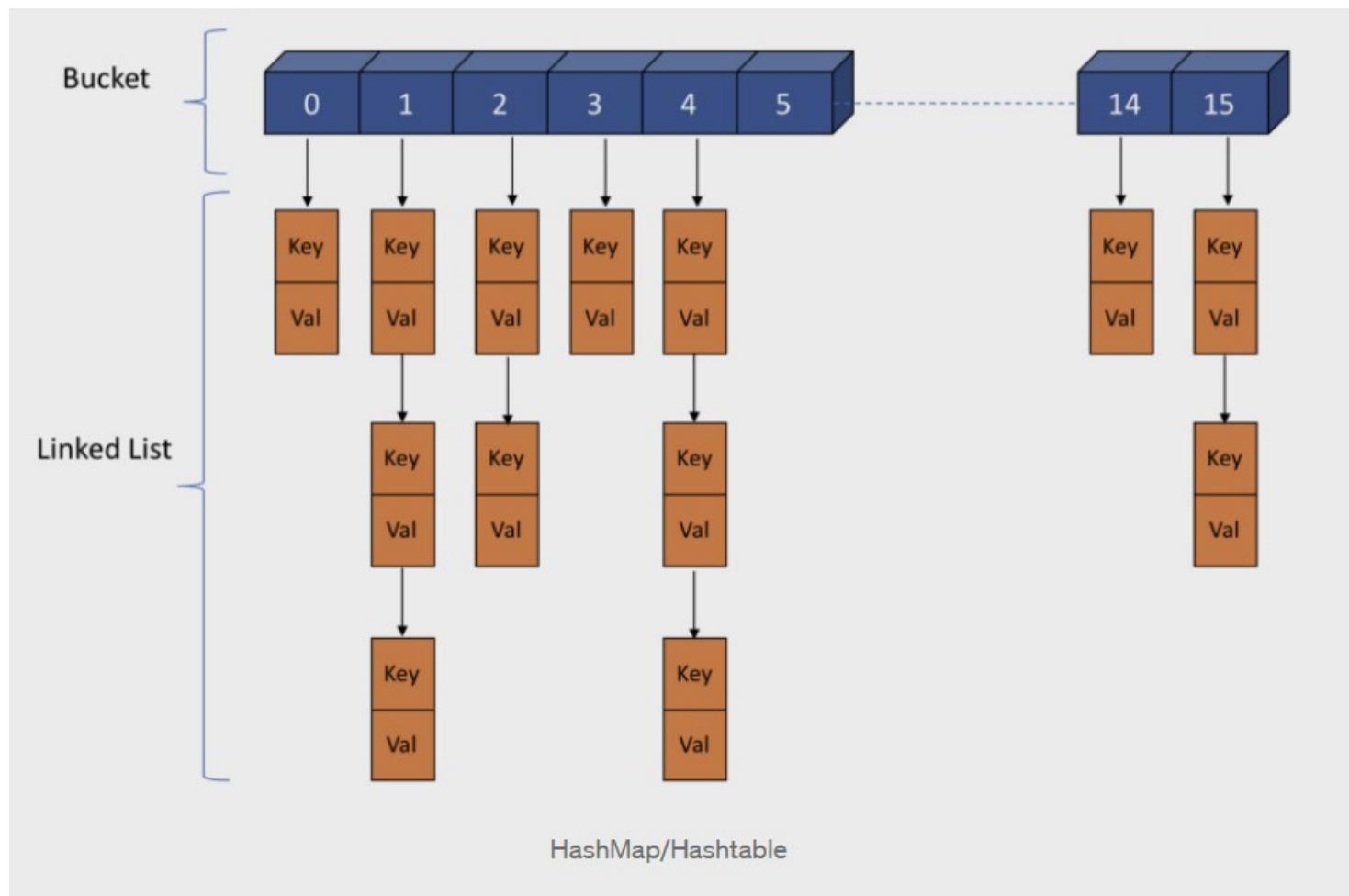


# CONCURRENT HASH MAP

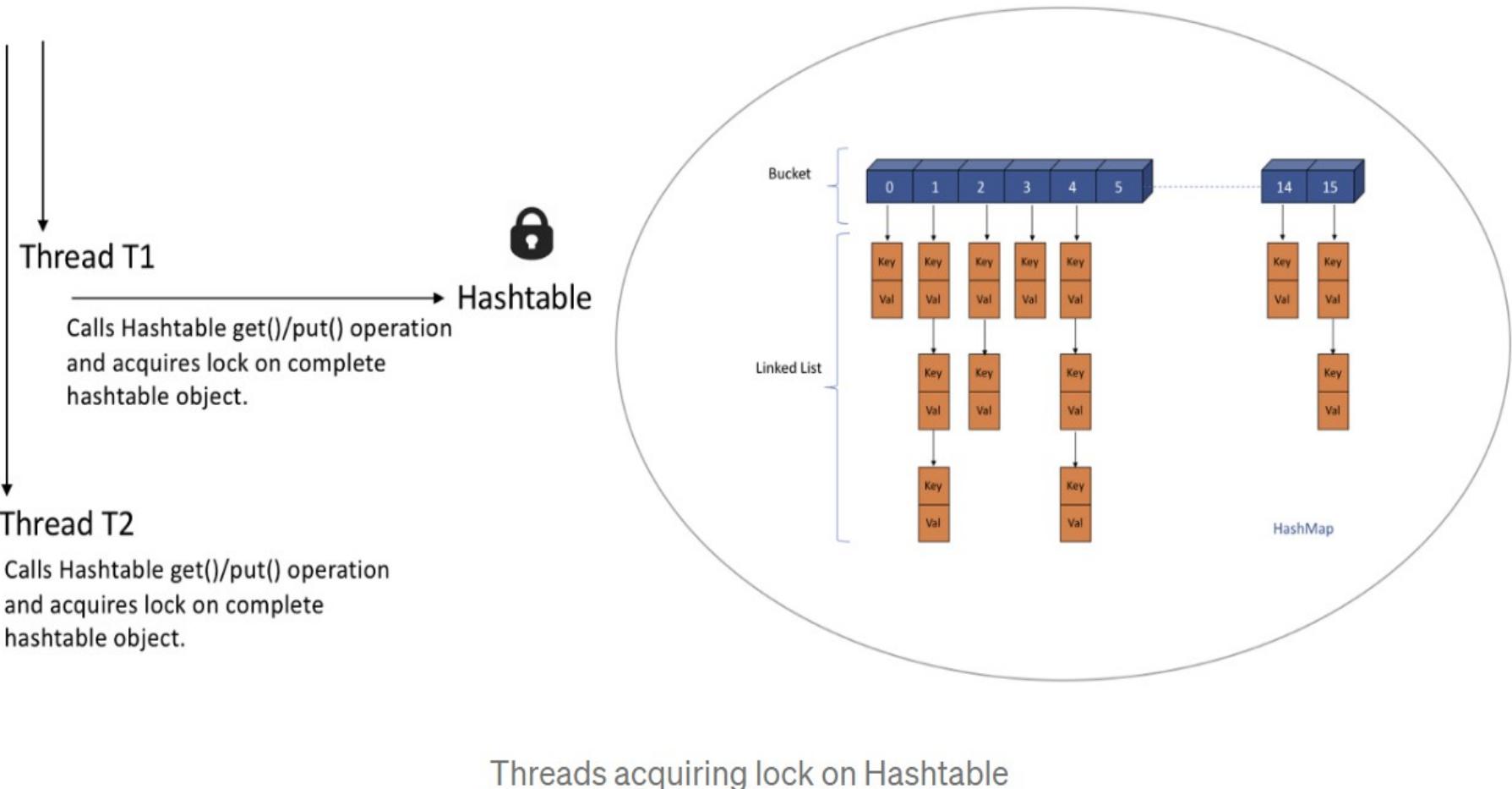
- le slide precedenti mostrano che la ConcurrentHashMap è molto più efficiente delle precedenti versioni di tabelle chiave-valore
- idea fondamentale: usare una diversa strategia di locking che offre migliore concorrenza e scalabilità
  - introduce un array di segmenti: ogni segmento punta ad una HashMap
  - fine grained locking
    - una lock per ogni segmento, **lock striping**
    - numero di segmenti determina il livello di concorrenza
  - modifiche simultanee possibili, se modificano segmenti diversi
    - 16 o più threads possono operare in parallelo su segmenti diversi
    - lettori possono accedere in parallelo a modifiche



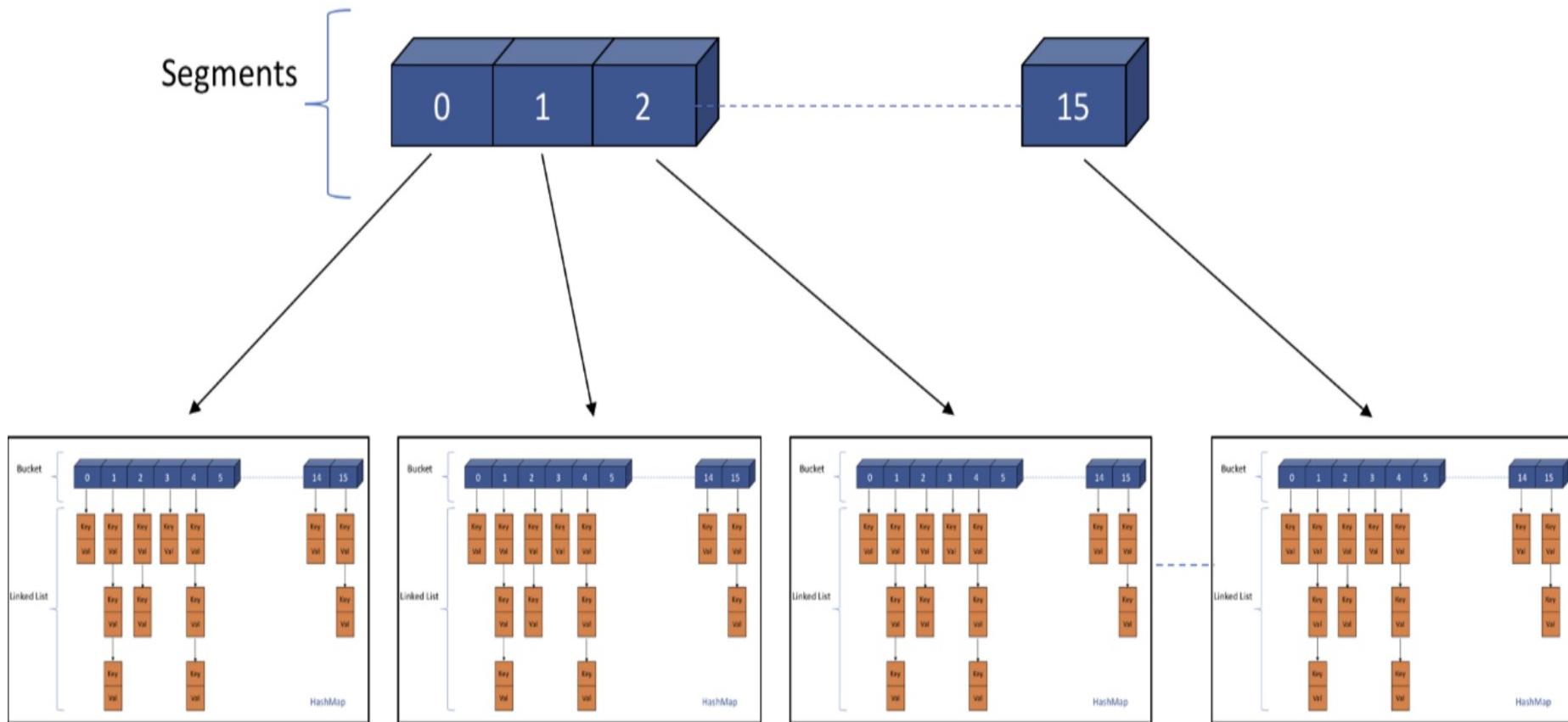
# HASHMAP INTERNAL



# HASHTABLE AND THREADS

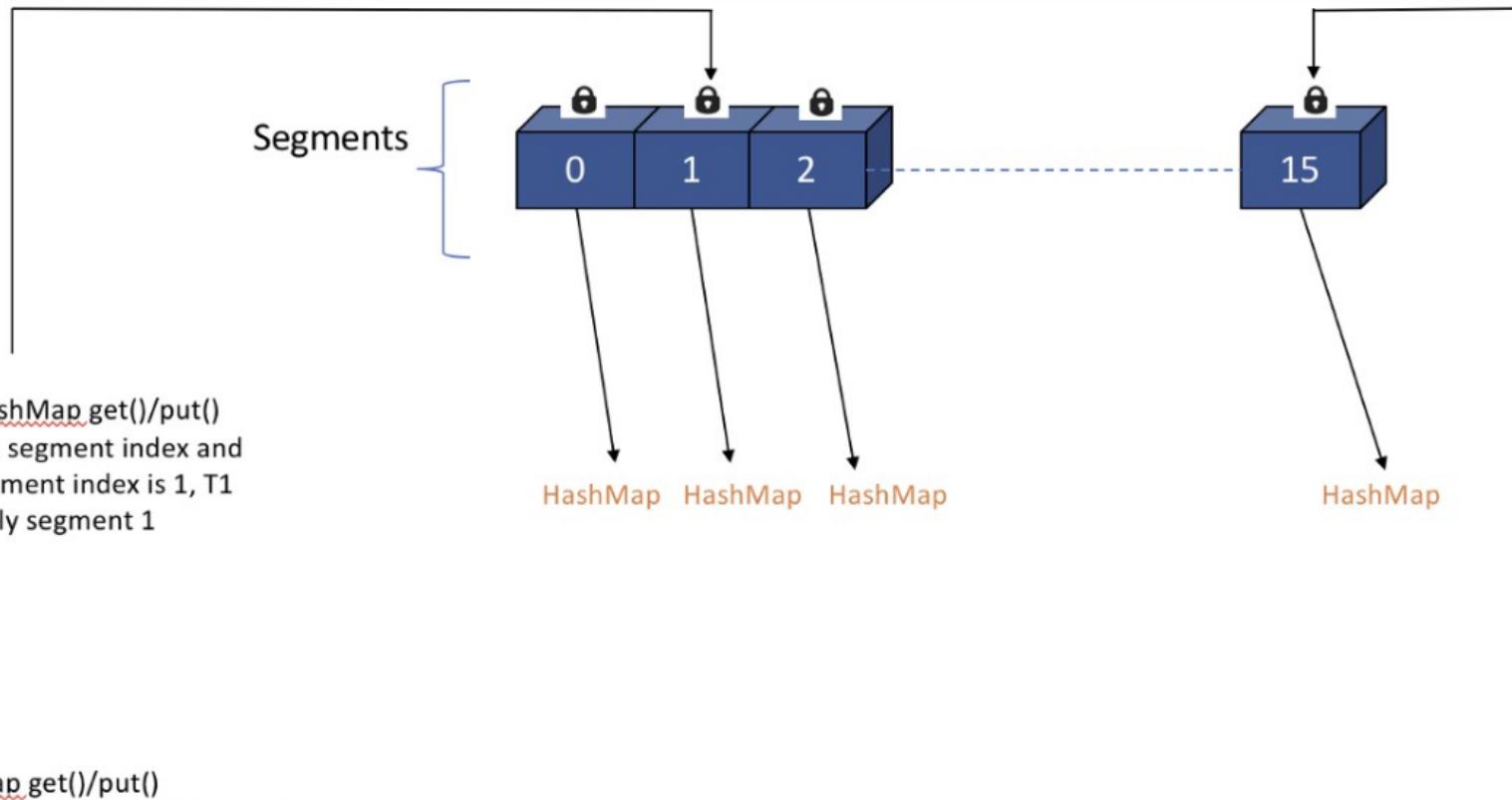


# CONCURRENTHASHMAP INTERNAL



- struttura utilizzata fino a JAVA7
- da JAVA8 alberi bilanciati invece di linked lists

# CONCURRENTHASHMAP INTERNAL



Threads acquiring lock on ConcurrentHashMap

# OPERAZIONI COMPOSTE ATOMICHE

le concurrent collections offrono inoltre un insieme di **operazioni composte atomiche**

- sequenze di operazioni di uso comune
- definite come una operazione unica
- la JVM traduce la singola operazione “ad alto livello” in una sequenza di operazioni a più basso livello
- garantisce inoltre la corretta sincronizzazione su tale operazione
  - ...secondo la filosofia “do not re-invent the wheel...”



# OPERAZIONI COMPOSTE: ATOMICITA'

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

public class CHashMap {
    private Map<String, Object> theMap =
        new ConcurrentHashMap<>();
    public Object getOrCreate(String key) {
        Object value = theMap.get(key);
        try { Thread.sleep(5000);
            } catch(Exception e) {};
        if (value == null) {
            value = new Object();
            theMap.put(key, value); }
        return value.hashCode();}}
```

```
public class Main {
    public static void main(String [] args)
    { CHashMap ex= new
        CHashMap();
    Thread t1 = new Thread (new Runnable()
    {public void run()
    {System.out.println
        (ex.getOrCreate("5"));}});;
    t1.start();
    Thread t2 = new Thread (new Runnable()
    {public void run()
    {System.out.println
        (ex.getOrCreate("5"));}});;
    t2.start();
    }}
```

- GetOrCreate **non è atomica**
- t1 e t2 stampano due valori diversi, entrambi associati alla stessa chiave, 5



# OPERAZIONI COMPOSTE ATOMICHE

- soluzione: utilizzare istruzioni atomiche composte
- funzioni del tipo “query-then-update”, “test-and-set”
- nel nostro caso è utile la `putIfAbsent`

```
public interface ConcurrentMap<K,V> extends Map<K,V> {  
    V putIfAbsent(K key, V value);  
        // Insert into map only if no value is mapped from K  
        // returns the previous value associated to the key  
        // or null if there is no mapping for that key  
  
    boolean remove(K key, V value);  
        // Remove only if K is mapped to V  
  
    boolean replace(K key, V oldValue, V newValue);  
        // Replace value only if K is mapped to oldValue  
  
    V replace(K key, V newValue);  
        // Replace value only if K is mapped to some value    }
```



# OPERAZIONI COMPOSTE: PUTIFABSENT

```
import java.util.*;  
import java.util.concurrent.*;  
  
public class Main1 {  
  
    static Map<String, Object> theMap = new ConcurrentHashMap<>();  
  
    public static void main(String [] args)  
    { Thread t1 = new Thread (  
  
        new Runnable() {public void run()  
  
            {Object obj1 = new Object();  
             System.out.println(theMap.putIfAbsent("5",obj1));}});  
  
        t1.start();  
  
        Thread t2 = new Thread (new Runnable() {public void run()  
  
            {Object obj2 = new Object();  
             System.out.println(theMap.putIfAbsent("5",obj2));}});  
  
        t2.start();}  
}
```

# COLLECTIONI ED ITERATORI

- le Collection JAVA supportano diversi tipi di iteratori
  - si distinguono riguardo al comportamento di una collezione in presenza di “**concurrent modification**”
  - cosa accade quando la collezione viene modificata, mentre un iteratore la sta scorrendo, e questa modifica arriva “dall'esterno” dell'iteratore?
- **fail-fast**
  - se c'è una modifica strutturale (inserzione, rimozione, aggiornamento), dopo che l'iteratore è stato creato, l'iteratore la rileva e solleva una `ConcurrentModificationException`
  - fallimento immediato dell'operatore, per evitare comportamenti non deterministici
  - la maggior parte delle collezioni “non-concurrenti” sono fail-fast  
`Vector`, `ArrayList`, `HashMap`, ed altre....



# FAIL FAST: HASHMAP

```
import java.util.HashMap; import java.util.Iterator;import java.util.Map;
public class FailFastExample {
    public static void main(String[] args)
    { Map<String, String> cityCode = new HashMap<String, String>();
        cityCode.put("Delhi", "India");
        cityCode.put("Moscow", "Russia");
        cityCode.put("New York", "USA");
        Iterator iterator = cityCode.keySet().iterator();
        while (iterator.hasNext()) {
            System.out.println(cityCode.get(iterator.next()));
            cityCode.put("Istanbul", "Turkey"); }}}
```

India

```
Exception in thread "main"
java.util.ConcurrentModificationException
at java.util.HashMap$HashIterator.nextNode(Unknown Source)
at java.util.HashMap$KeyIterator.next(Unknown Source)
at FailFastExample.main(FailFastExample.java:19)
```



# COLLECTIONI ED ITERATORI

- fail-safe (“snapshot”) introdotti in JAVA 1.5 con le concurrent collections
  - creano una copia della collezione, al momento della creazione dell'iteratore e lavorano su questa copia
  - non sollevano ConcurrentModificationException
  - l'iteratore accede ad una versione non aggiornata della collezione
  - CopyOnWriteArrayList
- weakly consistent introdotti in JAVA 1.5 con le concurrent collections
  - l'iteratore e modifiche operano sulla stessa copia
  - no ConcurrentModificationException, comportamento fail-safe
  - l'iteratore considera gli elementi che esistevano al momento della costruzione dell'iteratore e può riflettere le modifiche che sono avvenute dopo la costruzione dell'iteratore, anche se non è garantito
  - ConcurrentHashMap, ...



# WEAK CONSISTENCY: CONCURRENT HASH MAP

da JavaDocs;

*“The view's iterator is a “weakly consistent” iterator that will never throw ConcurrentModificationException, and guarantees to traverse elements as they existed upon construction of the iterator, and may (but is not guaranteed to) reflect any modifications subsequent to construction.”*

- l'iteratore
  - non clona la struttura al momento della creazione
  - la collezione può catturare le modifiche effettuate sulla collezione dopo la sua creazione
    - non solleva ConcurrentModificationException
- alcuni metodi, size() e isEmpty()
  - possono restituire un valore “approssimato”
  - “weakly consistent behaviour”



# UN ITERATORE WEAKLY CONSISTENT

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.Iterator;
public class FailSafeItr {
    public static void main(String[] args)
    {ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<String, Integer>();
        map.put("ONE", 1);
        map.put("TWO", 2);
        map.put("THREE", 3);
        map.put("FOUR", 4);
        Iterator <String> it = map.keySet().iterator();
        while (it.hasNext()) {
            String key = (String)it.next();
            System.out.println(key + " : " + map.get(key));
            // Notice, it has not created separate copy
            // It will print 7
            map.put("SEVEN", 7); }}}
// the program prints ONE : 1 FOUR : 4 TWO : 2 THREE : 3 SEVEN : 7
```



# FAIL SAFE: COPYONWRITEARRAYLIST

- come risulta evidente dal nome, effettua una copia dell'array tutte le volte che viene effettuata una operazione di modifica (add, set, etc..)
- “snapshot style iterator”: usa un riferimento ad una copia dello stato dell'array nel momento in cui l'iteratore è creato
  - l'array riferito non viene mai modificato durante la vita dell'iteratore: l'iteratore non cattura le modifiche effettuate dopo la sua creazione
- thread-safe: ogni thread lavora su una propria copia
- fail safe: non solleva ConcurrentModificationException
- operazione di copia molto costosa
  - è adatto quando ci sono più accessi in lettura che modifiche



# FAIL SAFE: COPYONWRITEARRAYLIST

- riprendiamo l'esempio mostrato nella lezione scorsa: divide et impera con multithreading
- calcolare la somma di tutti i numeri da 1 a n
  - individuare sottointervalli dell'intervallo 1-100
  - creare un task diverso per ogni intervallo: calcola la somma per quell'intervallo
  - sottomettere i task ad un theradpool, somme parziali in parallelo
  - raccogliere le somme parziali per calcolare la somma totale.
- soluzione diversa: le somme parziali non vengono raccolte sequenzialmente, partendo dalla prima somma, ma nell'ordine in cui esse sono disponibili.



# FAIL SAFE: COPYONWRITEARRAYLIST

```
import java.util.ArrayList; import java.util.List; import java.util.concurrent.*;  
  
public class Adder {  
  
    public static void main(String[] args) throws ExecutionException, InterruptedException  
    { ExecutorService executor = Executors.newFixedThreadPool(5);  
  
        List<Future<Integer>> list = new CopyOnWriteArrayList<Future<Integer>>();  
  
        for (int i = 0; i < 10; i=i+2) {  
  
            Calculator c = new Calculator(i, i + 1);  
  
            list.add(executor.submit(c)); }  
  
        int s=0;  
  
        while (!list.isEmpty() )  
  
        for (Future<Integer> f : list) {  
  
            if (f.isDone()) {System.out.println(f.get());  
  
                s=s+f.get();  
  
                list.remove(f); } }  
  
        System.out.println("la somma e'" +s);  
  
        executor.shutdown();}}}
```



# ASSIGNMENT 4: CONTEGGIO OCCORRENZE

- scrivere un programma che conta le occorrenze dei caratteri alfabetici (lettere dalla “A” alla “Z”) in un insieme di file di testo. Il programma prende in input una serie di percorsi di file testuali e per ciascuno di essi conta le occorrenze dei caratteri, ignorando eventuali caratteri non alfabetici (come per esempio le cifre da 0 a 9). Per ogni file, il conteggio viene effettuato da un apposito task e tutti i task attivati vengono gestiti tramite un pool di thread. I task registrano i loro risultati parziali all’interno di una ConcurrentHashMap.

Prima di terminare, il programma stampa su un apposito file di output il numero di occorrenze di ogni carattere. Il file di output contiene una riga per ciascun carattere ed è formattato come segue:

```
<carattere1>,<numero di occorrenze>
<carattere2>,<numero di occorrenze>
...
<caratteren>,<numero di occorrenze>
```



# ASSIGNMENT 4: CONTEGGIO OCCORRENZE

- esempio di file di output:

a,1281

b,315

c,261

d,302

...

- si allega un archivio compresso contenente file testuali per effettuare i test



# Reti e Laboratorio III

## Modulo Laboratorio III

### AA. 2022-2023

docente: Laura Ricci

[laura.ricci@unipi.it](mailto:laura.ricci@unipi.it)

## Lezione 5

### InetAddress

## Stream Sockets for clients

13/10/2022

# NETWORK APPLICATIONS

alcune “killer network applications”

- web browser
- SSH
- email
- social networks
- teleconferences (Skype, Zoom, GoToMeeting, Meet, Teams,...)
- program development environments: GIT
- collaborative work: Overleaf
- multiplayer games: War of Warcraft
- P2P File sharing: BitTorrent
- blockchain: cryptocurrencies (Bitcoin), supply chain,...
- metaverse, e molte altre....

scopo del corso è mettervi in grado di sviluppare una semplice applicazione di rete.

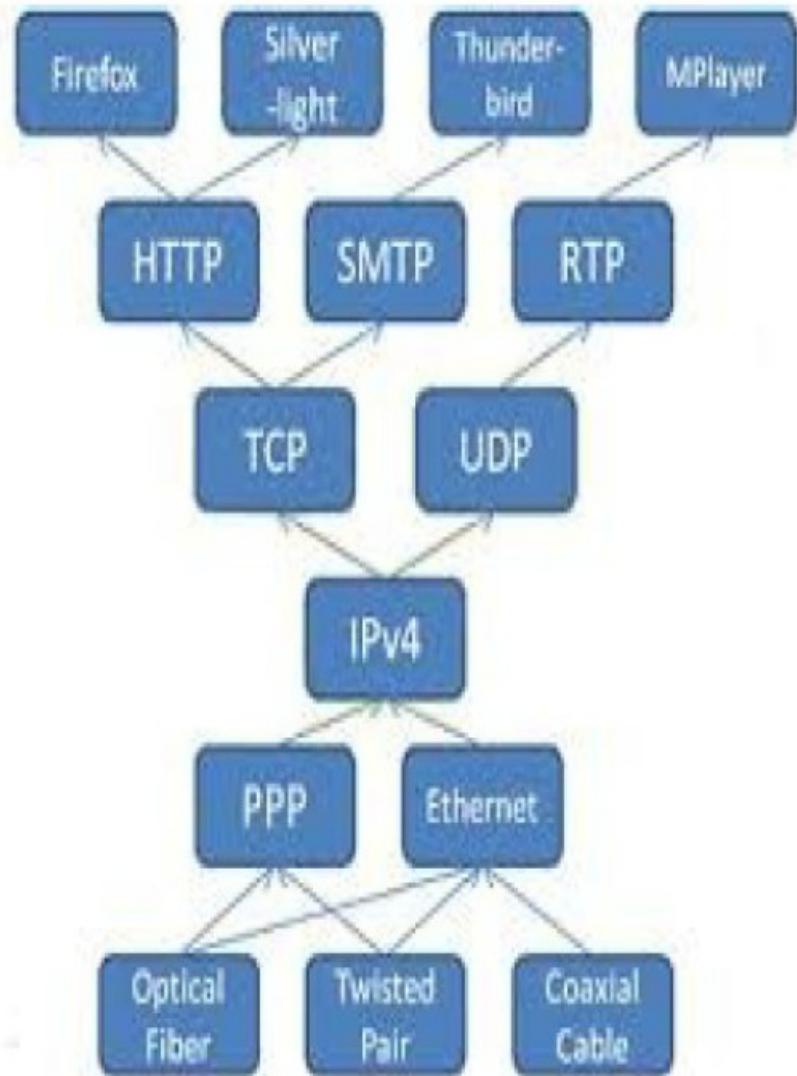
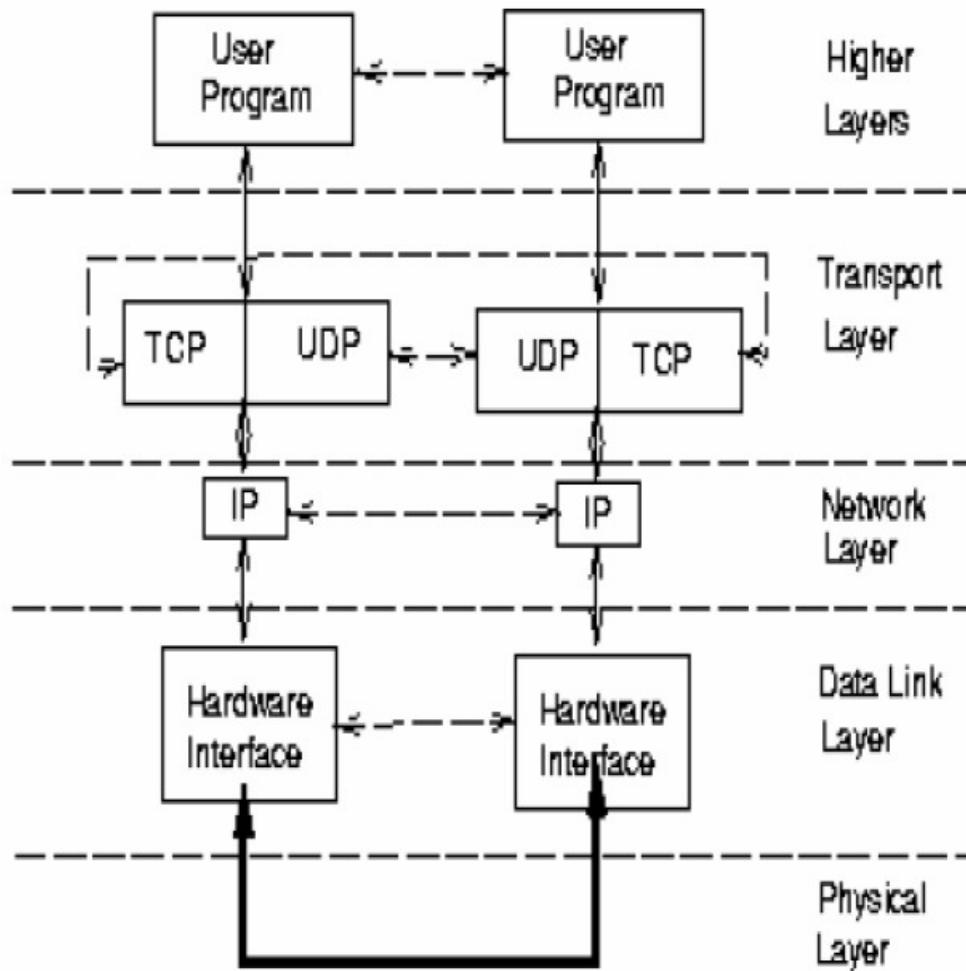


# NETWORK APPLICATIONS

- due o più **processi** (non thread!) in esecuzione su **hosts diversi**, distribuiti geograficamente sulla rete, **comunicano** e **cooperano** per realizzare una funzionalità globale:
- **cooperazione**: scambio informazioni utile per perseguire l'obiettivo globale, quindi implica comunicazione
- **comunicazione**: utilizza protocolli, ovvero insieme di regole che i partners devono seguire per comunicare correttamente.
- in questo corso utilizzeremo i protocolli di livello trasporto:
  - **connection-oriented**: TCP, Transmission Control Protocol
  - **connectionless**: UDP, User Datagram Protocol

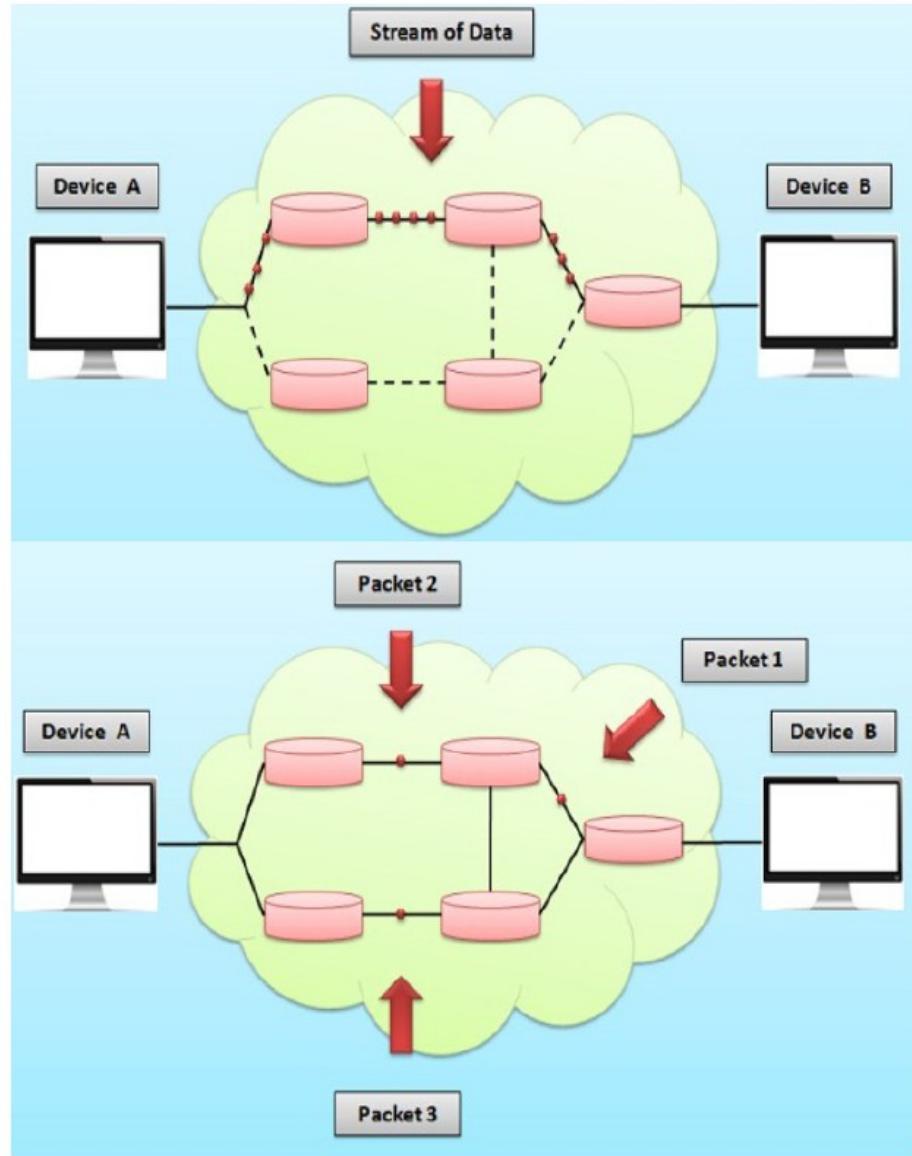


# NETWORK LAYERS: DAL MODULO DI TEORIA



# TIPI DI COMUNICAZIONE

- Connection Oriented (TCP)
  - come una chiamata telefonica
  - una connessione stabile (canale di comunicazione dedicato) tra mittente e destinatario
  - stream socket
- Connectionless (UDP)
  - come l'invio di una lettera
  - non si stabilisce un canale di comunicazione dedicato
  - ogni messaggio viene instradato in modo indipendente dagli altri
  - datagramsocket



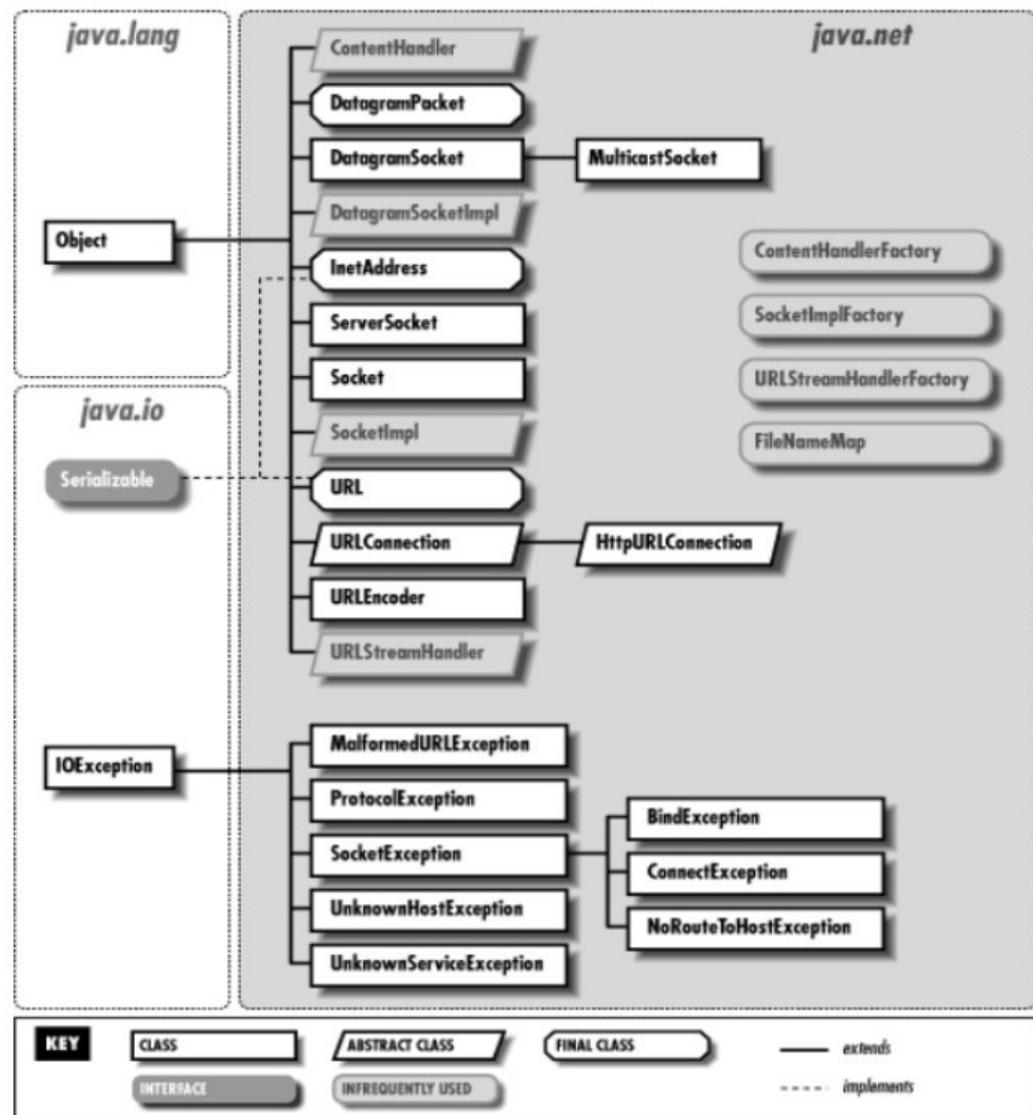
# JAVA.NET: NETWORKING IN JAVA

## connection-oriented

- connessione modellata come stream
- asimmetrici
  - client side: Socket class
  - server side:
    - ServerSocket class
    - Socket class

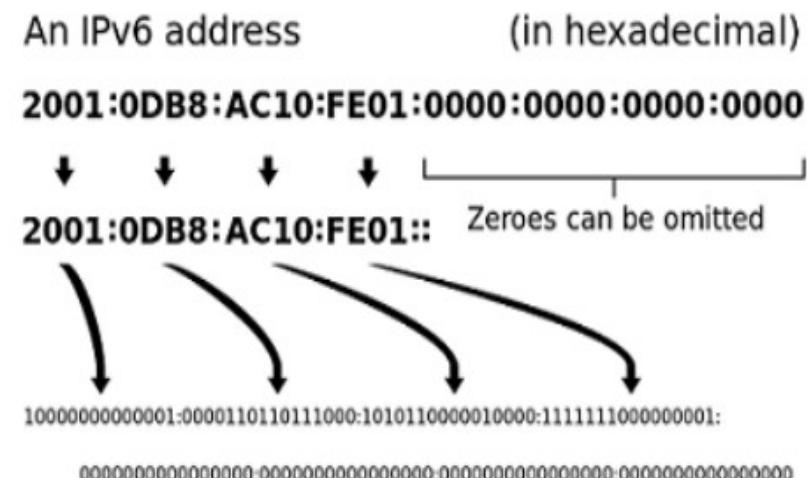
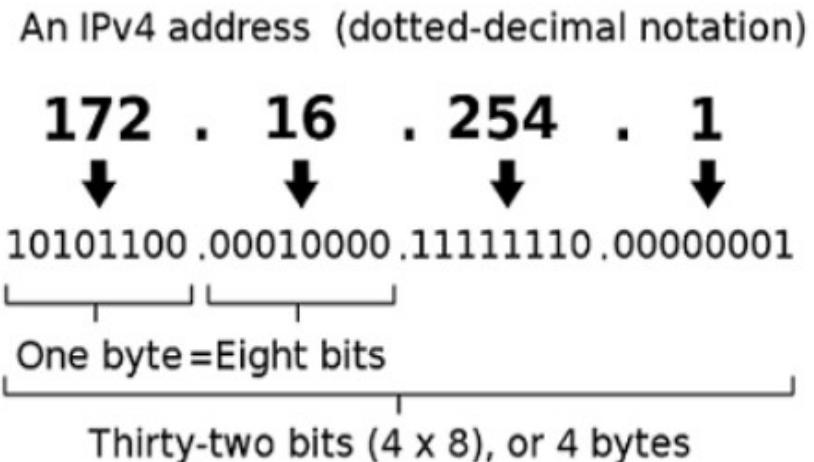
## connectionless

- simmetrici: sia per il client che per il server
  - datagramSocket
  - datagrampacket



# IP (INTERNET PROTOCOL) ADDRESS

- IPV4, 4 bytes:  $2^{32}$  indirizzi
  - dotted quad form
  - ogni byte interpretato come un numero decimale **senza segno**
  - alcuni indirizzi riservati, loopback address: 127.0.0.0, broadcast 255.255.255.255
- IPV6, 16 bytes:  $2^{128}$  indirizzi,
  - 8 blocchi di 4 cifre esadecimali



# DOMAIN NAMES

- gli indirizzi IP semplificano l'elaborazione effettuata dai routers, ma sono poco leggibili per gli utenti della rete
- soluzione
  - assegnare un **nome simbolico unico** ad ogni host della rete
  - utilizzare uno spazio **di nomi gerarchico**  
**fujih0.cli.di.unipi.it** (host fuji presente nell'aula H alla postazione 0, nel dominio **cli.di.unipi.it**)
  - livelli della gerarchia separati dal punto
  - nomi interpretati da destra a sinistra
  - un nome può essere mappato a più indirizzi IP
- indirizzi a lunghezza fissa verso nomi a lunghezza variabili
- **Domain Name System (DNS)** traduce nomi in indirizzi IP



# LA CLASSE INETADDRESS

```
public class InetAddress extends Object implements Serializable
```

- può gestire sia indirizzi IPv4 e IPv6
- usata per encapsulare in un unico oggetto di tipo InetAddress sia
  - l'indirizzo IP numerico: byte[] address
  - il nome di dominio per quell'indirizzo: String
- la classe non contiene alcun costruttore,
- allora, come posso creare oggetti di tipi InetAddress?
  - si utilizza una factory con metodi statici
  - i metodi si connettono al DNS per risolvere un hostname, ovvero trovare l'indirizzo IP ad esso corrispondente: necessaria una connessione di rete
  - possono sollevare UnknownHostException, se non riescono a risolvere il nome dell'host



# LA CLASSE INETADDRESS

getByName() lookup dell'indirizzo di un host

```
import java.net.*;                                     $ java FindIP
public class FindIP {                                 www.unipi.it/131.114.21.42
    public static void main (String[] args) {
        try {
            InetAddress address = InetAddress.getByName("www.unipi.it");
            System.out.println(address);
        } catch (UnknownHostException ex) {
            System.out.println("Could not find www.unipi.it"); }} }
```

getLocalHost() lookup dell'indirizzo locale

```
import java.net.*;                                     $Java MyAddress
public class MyAddress {                             DESKTOP-R5C46F3/192.168.1.196
    public static void main (String [] args)
    {try {
        InetAddress address = InetAddress.getLocalHost();
        System.out.println(address);
    } catch (UnknownHostException ex) {
        System.out.println("Could not find this computer's address"); }}
```



# LA CLASSE INETADDRESS

- `getAllByName()` lookup di tutti gli indirizzi di un host

```
import java.net.*;  
  
public class FindAllIP {  
  
    public static void main (String[] args) {  
  
        try { InetAddress [] addresses = InetAddress.getAllByName ("www.repubblica.it");  
  
            for (InetAddress address : addresses)  
                { System.out.println (address); }  
  
        } catch (UnknownHostException ex) {  
  
            System.out.println ("Could not find www.repubblica.it"); } } }  
  
$ java FindAllIP  
www.repubblica.it/18.66.196.45  
www.repubblica.it/18.66.196.118  
www.repubblica.it/18.66.196.94  
www.repubblica.it/18.66.196.112
```

- `getLocalHost()` restituisce l' InetAddress del local host

```
import java.net.*;  
  
public class MyAddress {  
  
    public static void main (String[] args) {  
  
        try {  
  
            InetAddress address = InetAddress.getLocalHost();  
  
            System.out.println (address);  
  
        } catch (UnknownHostException ex)  
  
            {System.out.println ("Could not find this computer address"); } } }
```



# INETADDRESS: CACHING

- i metodi descritti **effettuano caching** dei nomi/indirizzi risolti
  - l'accesso al DNS è una operazione potenzialmente molto costosa
  - nomi risolti con i dati nella cache, quando possibile (di default: per sempre)
  - anche i tentativi di risoluzione non andati a buon fine in cache
- permanenza dati nella cache:
  - 10 secondi se la risoluzione non ha avuto successo, spesso il primo tentativo di risoluzione fallisce a causa di un time out...
  - tempo illimitato altrimenti.
  - problemi: indirizzi dinamici.
- controllo dei tempi di permanenza in cache

```
java.security.Security.setProperty  
        ("networkaddress.cache.ttl","0");
```
- per i tentativi non andati a buon fine: networkaddress.cache.negative.ttl



# CACHING DI INDIRIZZI IP: “UNDER THE HOOD”

```
import java.net.InetAddress; import java.net.UnknownHostException;
import java.security.*;
public class Caching {
    public static final String CACHINGTIME="0";
    public static void main(String [] args) throws InterruptedException
    {Security.setProperty("networkaddress.cache.ttl",CACHINGTIME);
    long time1 = System.currentTimeMillis();
    for (int i=0; i<1000; i++){
        try {System.out.println(
            InetAddress.getByName("www.cnn.com").getHostAddress());}
        catch (UnknownHostException uhe)
            { System.out.println("UHE");} }
    long time2 = System.currentTimeMillis();
    long diff=time2-time1; System.out.println("tempo trascorso e'" +diff);}}
```

CACHINGTIME=0 tempo trascorso è 545

CACHINGTIME=1000 tempo trascorso è 85



# INETADDRESS: FACTORY METHODS

- metodi statici di una classe che restituiscono oggetti di quella classe
- i seguenti metodi contattano il DNS per la risoluzione di indirizzo/hostname

```
static InetAddress getLocalHost() throws UnknownHostException
```

```
static InetAddress getByName (String hostname) throws UnknownHostException
```

```
static InetAddress [] getAllByName (String hostName)
```

```
throws UnknownHostException
```

```
static InetAddress getLoopBackAddress()
```

- i seguenti metodi statici costruiscono oggetti di tipo InetAddress, ma non contattano il DNS (utile se DNS non disponibile e conosco indirizzo/host)
- nessuna garanzia sulla correttezza di hostname/IP, UnknownHostException sollevata solo se l'indirizzo è malformato

```
static InetAddress getByAddress(byte[] IPAddr[]) throws UnknownHostException
```

```
static InetAddress getByAddress (String hostName, byte[] IPAddr[])
```

```
throws UnknownHostException
```



# INETADDRESS: INSTANCE METHODS

- la classe InetAddress ha moltissimi “metodi di istanza” che possono essere utilizzati sull’istanza di un oggetto InetAddress (costruito con uno dei metodi della Factory)

boolean **equals**(Object other)

byte [] **getAddress**()

String **getHostAddress**()

String **getHostName**()

boolean **isLoopBackAddress**()

boolean **isMulticastAddress**()

boolean **isReachable**()

String **toString** ()

.... e molto altri (vedere le API)



# INETADDRESS: INSTANCE METHODS

```
import java.net.*; import java.util.Arrays; import java.io.*;

public class InetAddressInstance {

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

InetAddress ia1 = InetAddress.getByName("www.google.com");

byte [] address = ia1.getAddress();

System.out.println(Arrays.toString(address));

System.out.println(ia1.getHostAddress());

System.out.println(ia1.getHostName());

System.out.println(ia1.isReachable(1000));

System.out.println(ia1.isLoopbackAddress());

System.out.println(ia1.isMulticastAddress());

System.out.println(InetAddress.getByAddress(new byte[]{127,0,0,1}).isLoopbackAddress());

System.out.println(InetAddress.getByAddress(new byte[] {((byte)225,(byte)255,(byte)255,
(byte)255}).isMulticastAddress());}}
```

\$ Java InetAddressInstance  
[-114, -6, -76, -124]  
142.250.180.132  
www.google.com  
true  
false  
false  
true  
true



# UN PROGRAMMA UTILE: SPAM CHECKER

- diversi servizi monitorano gli spammers: **real-time black-hole lists** (RTBLs)
  - ad esempio: [sbl.spamhaus.org](http://sbl.spamhaus.org)
  - mantengono una lista di indirizzi IP che risultano, probabilmente, degli spammers
- per identificare se un indirizzo IP corrisponde ad uno spammer:
  - inversione dei bytes dell'indirizzo IP
  - concatena il risultato a [sbl.spamhaus.org](http://sbl.spamhaus.org)
  - esegui un DNS look-up
  - la query ha successo se e solo se l'indirizzo IP corrisponde ad uno spammer
- SpamCheck richiede a [sbl.spamhaus.org](http://sbl.spamhaus.org) se un indirizzo IPv4 è uno spammer noto
  - es una query DNS su `17.34.87.207.sbl.spamhaus.org` ha successo se l'indirizzo è uno spammer



# UN PROGRAMMA UTILE: SPAM CHECKER

```
import java.net.*;  
  
public class SpamCheck {  
    public static final String BLACKHOLE = "sbl.spamhaus.org";  
    public static void main(String[] args) throws UnknownHostException  
    { for (String arg: args) {  
        if (isSpammer(arg)) {  
            System.out.println(arg + " is a known spammer.");  
        } else {  
            System.out.println(arg + " appears legitimate."); }}}  
  
private static boolean isSpammer(String arg) {  
    try { InetAddress address = InetAddress.getByName(arg);  
        byte [ ] quad = address.getAddress();  
        String query = BLACKHOLE;  
        for (byte octet : quad) {  
            int unsignedByte = octet < 0 ? octet + 256 : octet;  
            query = unsignedByte + "." + query;  
        }  
        InetAddress.getByName(query);  
        return true;  
    } catch (UnknownHostException e) { return false; }}}
```

```
$java SpamCheck 23.45.65.88 141.250.89.99  
127.0.0.2  
  
23.45.65.88 appears legitimate.  
141.250.89.99 appears legitimate.  
127.0.0.2 is a known spammer
```



# IL PARADIGMA CLIENT/SERVER

servizio:

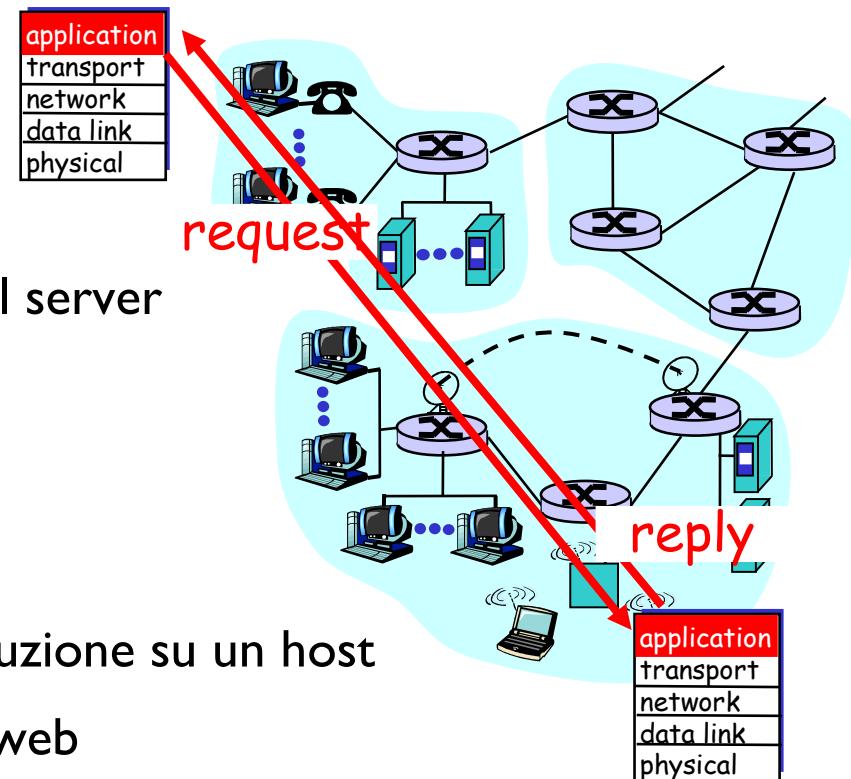
- software in esecuzione su una o più macchine.
- fornisce l'astrazione di un insieme di operazioni

client:

- un software che sfrutta servizi forniti dal server
  - web client      browser
  - e-mail client    mail-reader

server:

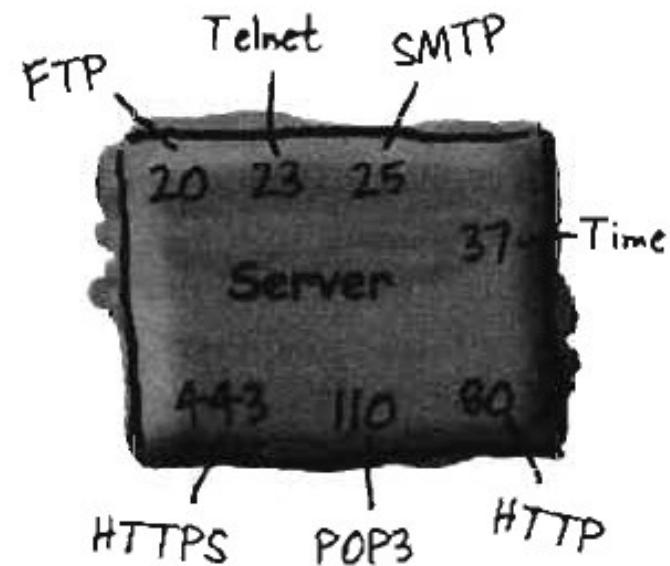
- istanza di un particolare servizio in esecuzione su un host
- ad esempio: server Web invia la pagina web richiesta, mail server consegna la posta al client



# IDENTIFICARE I SERVIZI

- occorre specificare:
  - l'host, tramite indirizzo IP (la rete all'interno della quale si trova l'host + l'host all'interno della rete)
  - la **porta** individua un servizio tra i tanti **servizi** (es: e-mail, ftp, http,...) attivi su un host
- ogni servizio individuato da una **porta**
  - intero tra 1 e 65535 (per TCP ed UDP)
  - non un **dispositivo fisico**, ma un'**astrazione** per individuare i singoli servizi (processi)
- porte 1-1023: riservate per **well-known services**.

Well-known TCP port numbers  
for common server applications



A server can have up to 65536 different server apps running, one per port.



# CONNETTERSI AD UN SERVIZIO

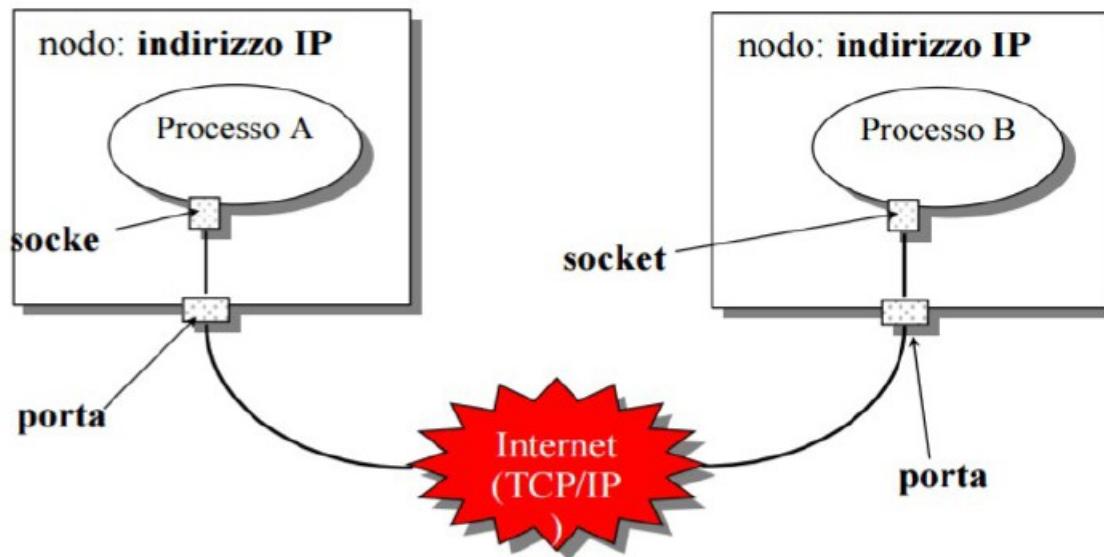
- socket: uno standard per connettere dispositivi **distribuiti, diversi, eterogenei**
- termine utilizzato in tempi remoti in telefonia.
  - la connessione tra due utenti veniva stabilita tramite un operatore
  - l'operatore inseriva fisicamente i due estremi di un cavo in due ricettacoli (sockets)
  - un socket per ogni utente



# SOCKET: UNO “STANDARD” DI COMUNICAZIONE

- una presa “standard” a cui un processo si può collegare per spedire dati
- un endpoint sull'host locale di un canale di comunicazione da/verso altri hosts
- introdotti in Unix BSD 4.2
- collegati ad una **porta locale**

## COMUNICAZIONE VIA SOCKET



# COME IL CLIENT ACCEDE AD UN SERVIZIO

- per usufruire di un servizio, il client apre un socket individuando
  - host + porta che identificano il servizio
  - invia/riceve messaggi su/da uno stream
- in JAVA: **java.net.Socket**
  - usa codice nativo per comunicare con lo stack TCP locale

```
public Socket(InetAddress host, int port) throws IOException
```

- crea un **socket** su una porta effimera e tenta di stabilire, tramite esso, una connessione con l'host individuato da InetAddress, sulla porta port.
- se la connessione viene rifiutata, lancia una eccezione di IO

```
public Socket (String host, int port) throws  
UnKnownHostException, IOException
```

come il precedente, l'host è individuato dal suo nome simbolico: interroga automaticamente il DNS)



# PORT SCANNER

- ricerca quale delle prime 1024 porte di un host è associata ad un servizio

```
import java.net.*;
import java.io.*;
public class LowPortScanner {
    public static void main(String[] args) {
        String host = args.length > 0 ? args[0] : "localhost";
        for (int i = 1; i < 1024; i++) {
            try {
                Socket s = new Socket(host, i);
                System.out.println("There is a server on port " + i + " of " + host);
                s.close();
            } catch (UnknownHostException ex) {
                System.err.println(ex);
                break;
            } catch (IOException ex) {
                // must not be a server on this port
            }}}
```

```
$java LowPortScanner
```

```
There is a server on port 80 of localhost
There is a server on port 135 of localhost
There is a server on port 445 of localhost
There is a server on port 843 of localhost
```



# PORT SCANNER: ANALISI

- il client richiede un servizio tentando di creare un socket su ognuna delle prime 1024 porte di un host
  - nel caso in cui non vi sia alcun servizio attivo, il socket non viene creato e viene invece sollevata un'eccezione
- il programma precedente effettua 1024 interrogazioni al DNS, una per ogni socket che tenta di creare, impiega molto tempo
- come ottimizzare il programma? utilizzare un diverso costruttore

```
public Socket(InetAddress host, int port) throws IOException
```

  - viene utilizzato l' InetAddress invece del nome dell'host per costruire i sockets
  - costruire l'InetAddress invocando InetAddress.getByName una sola volta, prima di entrare nel ciclo di scanning,



# MODELLARE UNA CONNESSIONE MEDIANTE STREAM

- una volta stabilita una connessione tra client e server devono scambiarsi dei dati. La connessione è modellata **come uno stream**.
- associare uno stream di input o di output ad un socket:

```
public InputStream getInputStream () throws IOException  
public OutputStream getOutputStream () throws IOException
```

- invio di dati: client/server leggono/scrivono dallo/sullo stream
  - un byte/una sequenza di bytes
  - dati strutturati/oggetti. In questo caso è necessario associare dei filtri agli stream
- ogni valore scritto sullo stream di output associato al socket viene copiato nel *Send Buffer* del livello TCP
- ogni valore letto dallo stream viene prelevato dal *Receive Buffer* del livello TCP



# INTERAGIRE CON IL SERVER TRAMITE SOCKET

- client implementato in JAVA, server in qualsiasi altro linguaggio
  - aprire un socket sock sulla porta su cui è attivo il servizio
  - utilizzare gli stream per la comunicazione con il servizio
- occorre conoscere il protocollo ed il formato dei dati scambiati, che sono codificati in un formato interscambiabile
  - testo
  - JSON
  - XML
- possibile conoscere il formato dei dati scambiati interagendo con il server tramite il protocollo telnet



# DAYTIME PROTOCOL (RFC 867)

- aprire una connessione sulla porta 13, verso il servizio time.nist.gov (NIST: National Institute of Standards and Technology)

```
$ telnet time.nist.gov 13
Trying 129.6.15.28...
Connected to time.nist.gov.
Escape character is '^]'.
```

```
56375 13-03-24 13:37:50 50 0 0 888.8 UTC(NIST) *
Connection closed by foreign host.
```

Format: JJJJJ YY-MM-DD HH:MM:SS TT L H msADV UTC(NIST) OTM

- JJJJJ: Modified Julian Date (days since Nov 17, 1858)
- TT: 00 means standard time and 50 means daylight savings time
- L: indicates whether a leap second will be added (1) or subtracted (2)
- H: health of the server (0: healthy; 1: up to 5 seconds off; ...)
- msADV: how long (ms) it estimates it's going to take for the response to return
- UTC (NIST): time-zone constant string
- OTM: almost a constant (an asterisk)



# DAYTIME PROTOCOL CLIENT

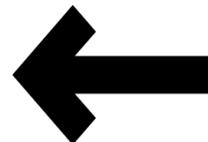
```
public class TimeClient {  
    public static void main(String[] args) {  
        String hostname = args.length > 0 ? args[0] : "time.nist.gov";  
        Socket socket = null;  
        try {  
            socket = new Socket(hostname, 13);  
            socket.setSoTimeout(15000);  
            InputStream in = socket.getInputStream();  
            StringBuilder time = new StringBuilder();  
            InputStreamReader reader = new InputStreamReader(in, "ASCII");  
            for (int c = reader.read(); c != -1; c = reader.read()) {  
                time.append((char) c);  
            }  
            System.out.println(time);  
        } catch (IOException ex) { System.out.println("could not connect to  
time.nist.gov");}  
    } finally {  
        if (socket != null) {  
            try {  
                socket.close();  
            } catch (IOException ex) { // ignore }}}}}
```

- setSoTimeout(<ms>): setta un timeout sul socket
- previene attese indeterminate di risposte dal server
  - solleva SocketTimeoutException (è una IOException)



# DAYTIME CON TRY WITH RESOURCES

```
try { socket = new Socket(hostname, 13);
    //read from the socket
} catch (IOException ex)
{System.out.println("could not connect to time.nist.gov");}
  
  
finally {
    if (socket != null) {
        try {
            socket.close();
        } catch (IOException ex) { // ignore }}}}}
```

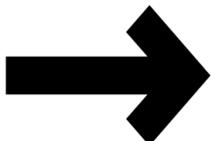


Java 6-: rilascio esplicito delle risorse

Java 7+: autochiusura tramite

try with resources

clausola finally non necessaria

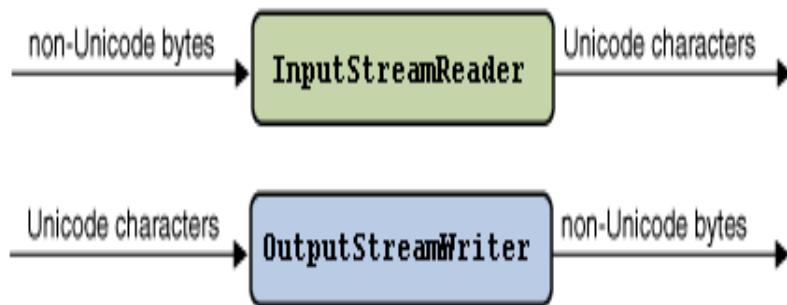


```
try (Socket socket = new Socket("time.nist.gov", 13))
{
    //read from the socket
} catch (IOException ex)
{ System.out.println("could not
connect to time.nist.gov");}
```



# DAYTIME: LEGGERE CARATTERI

- utilizza InputStreamReader
- istanziato su un InputStream
- parametro
  - codifica dei caratteri presenti sullo stream di byte (ASCII, UTF-8, UTF-16,...)
- traduce caratteri esterni nella codifica interna Unicode



```
.....  
InputStream in =  
    socket.getInputStream();  
  
StringBuilder time = new  
    StringBuilder();  
  
InputStreamReader reader = new  
    InputStreamReader(in, "ASCII");  
  
for (int c=reader.read();c != -1;  
{  
    time.append((char) c); }  
  
.....
```



# TRY WITH RESOURCES

- introdotto in JAVA 7, aggiornato in JAVA 9
- chiusura sistematica ed automatica delle risorse usate da un programma
- un blocco try con uno o più argomenti tra parentesi.
  - argomenti = risorse che devono essere chiuse quando il try block termina
  - le variabili che rappresentano le risorse non devono essere riutilizzate
- suppressed exceptions:
  - quando si verificano delle eccezioni sia nel blocco try-with-resources sia durante la chiusura, la JVM sopprime l'eccezione generata nella chiusura automatica.
- generalizzazione: implementazione della AutoCloseable interface



# TRY WITH RESOURCES

- una certa risorsa è chiusa “automaticamente”, dopo che è stata utilizzata
  - risorsa: file, stream, reader o socket
  - tecnicamente ogni oggetto che implementi l'interfaccia AutoClosable

```
try (FileWriter w = new FileWriter("file.txt")) {  
    w.write("Hello World"); }  
// w.close() is called automatically
```

- in questo esempio, w.close() viene chiamata indipendentemente dal fatto che la write sollevi o meno una eccezione
- concettualmente simile ad aggiungere w.close() in un blocco finally
- possibile usare più risorse in un blocco try with resources, vengono chiuse in senso inverso rispetto all'ordine con cui sono state dichiarate



# TRY WITH RESOURCES: ECCEZIONI

- nel seguente esempio

```
try (FileWriter w = new FileWriter("file.txt")) {  
    w.write("Hello World"); }  
    // w.close() is called automatically
```

- una eccezione può essere sollevata nei seguenti statement
  - `new FileWriter("file.txt")`
  - `w.write("Hello World")`
  - implicitamente da `w.close()`
- eccezione sollevata nel costruttore: nessun oggetto da chiudere, si propaga la eccezione senza eseguire la `write()`

```
try (FileWriter w = new FileWriter("file.txt")) {  
    w.write("Hello World");  
}  
// no call to w.close()
```



# TRY WITH RESOURCES: ECCEZIONI

- nel seguente esempio

```
try (FileWriter w = new FileWriter("file.txt")) {  
    w.write("Hello World"); }  
    // w.close() is called automatically
```

- eccezione sollevata nella write() : viene invocato w.close(), poi si propaga l'eccezione

```
try (FileWriter fw = new FileWriter("file.txt")) {  
    fw.write("Hello World");  
}  
// Implicit call to fw.close()
```



# TRY WITH RESOURCES: ECCEZIONI

- nel seguente esempio

```
try (FileWriter w = new FileWriter("file.txt")) {  
    w.write("Hello World"); }  
    // w.close() is called automatically
```

- eccezione sollevata nella write() : viene invocato w.close(), poi si propaga l'eccezione

```
try (FileWriter fw = new FileWriter("file.txt")) {  
    fw.write("Hello World");  
}  
// Implicit call to fw.close()
```



# TRY WITH RESOURCES: ECCEZIONI

- nel seguente esempio

```
try (FileWriter w = new FileWriter("file.txt")) {  
    w.write("Hello World"); }  
    // w.close() is called automatically
```

- eccezione sollevata nella chiamata implicita alla close() : viene propagata la eccezione

```
try (FileWriter fw = new FileWriter("file.txt")) {  
    fw.write("Hello World");  
}  
// Implicit call to fw.close()
```



# TRY WITH RESOURCES: SUPPRESSED EXCEPTIONS

- nel seguente esempio

```
try (FileWriter w = new FileWriter("file.txt")) {  
    w.write("Hello World"); }  
    // w.close() is called automatically
```

- cosa accade se la `w.write()` solleva un'eccezione ed anche la chiamata implicita alla `w.close()` la solleva?
- la prima eccezione “vince” sulla seconda e la seconda viene soppressa



# TRY WITH RESOURCES: SUPPRESSED EXCEPTIONS

```
import java.io.*;  
  
public class trywithresources  
{ public static void main (String args[])throws IOException {  
    try(FileInputStream input = new FileInputStream(new File("immagine.jpg"));  
        BufferedInputStream bufferedInput = new BufferedInputStream(input))  
    {  
        int data = bufferedInput.read();  
        while(data != -1){  
            System.out.print((char) data);  
            data = bufferedInput.read();  
        }}}}}
```

- risolve il problema delle “suppressed exceptions”
  - eccezioni possono essere sollevate nel blocco try, oppure nel blocco finally,
  - un'eccezione rilevata nella finally sopprimerebbe l'eccezione rilevata nel blocco try
- con il try with resources viene propagata l'eccezione rilevata nel blocco try



# HALF CLOSED SOCKETS

- `close()`: chiusura del socket in entrambe le direzioni
- half closure: chiusura del socket in una sola direzione
  - `shutdownInput()`
  - `shutdownOutput()`
- in molti protocolli: il client manda una richiesta al server e poi attende la risposta

```
try ( Socket connection = new Socket("www.somesite.com", 80)){  
    Writer out = new OutputStreamWriter(  
        connection.getOutputStream(), "8859_1");  
    out.write("GET / HTTP 1.0\r\n\r\n");  
    out.flush();  
    connection.shutdownOutput();  
    // read the response  
} catch (IOException ex) { ex.printStackTrace(); }
```
- scritture successive sollevano una `IOException`



# COSTRUZIONE SOCKET SENZA CONNESSIONE

- costruttore senza argomenti e connessione successiva

```
try {  
    Socket socket = new Socket();  
    // setta opzioni Socket, ad esempio timeout  
    SocketAddress = new InetSocketAddress ("time.nist.gov", 13);  
    socket.bind(connect(address));  
    // utilizza il socket  
} catch (IOException ex) {System.out.println(err); }
```

- scritture successive sollevano una IOException
- InetSocketAddress: costruttori

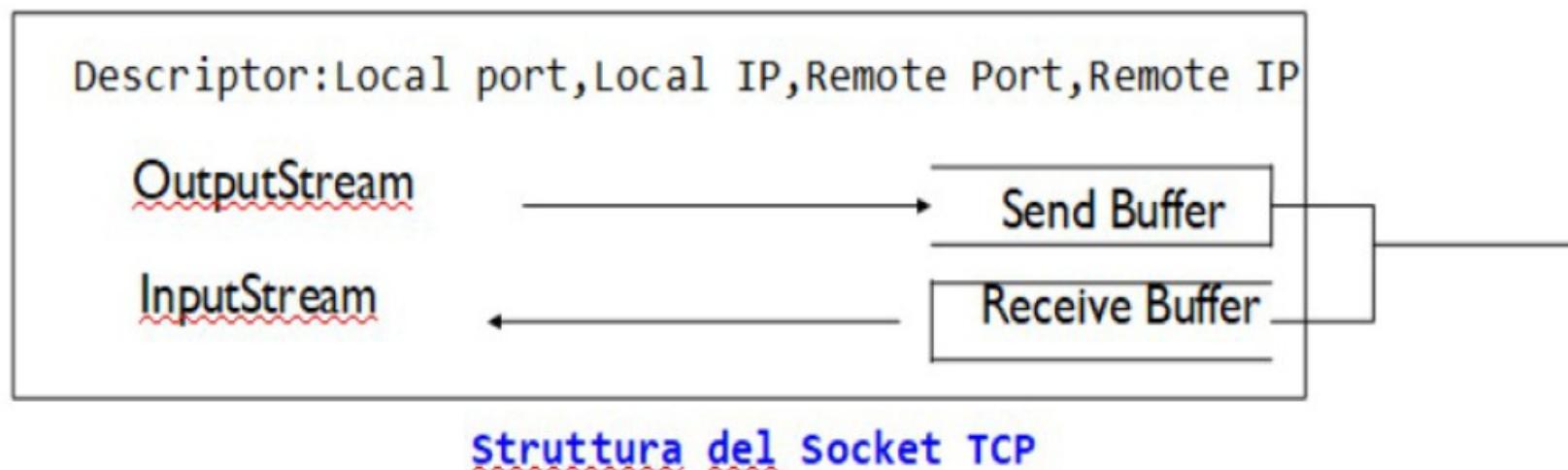
```
public InetSocketAddress (InetAddress address, int port);  
public InetSocketAddress(String host, int port);  
public InetSocketAddress (int port);
```



# REPERIRE INFORMAZIONI SU UN SOCKET

- metodi getter

```
public InetAddress getInetAddress() } indirizzo e porta  
public int getPort() host remoto  
public InetAddress getLocalAddress( } indirizzo e porta  
public int getLocalPort() host locale
```



# REPERIRE INFORMAZIONI SU UN SOCKET

```
import java.net.*;
import java.io.*;

public class SocketInfo {
    public static void main(String [] args)
    { for (String host: args) {
        try {
            Socket theSocket = new Socket (host, 80);
            System.out.println("Connected to "+theSocket.getInetAddress()
                +" on port"+ theSocket.getPort()+" from port "
                + theSocket.getLocalPort() + " of"
                + theSocket.getLocalAddress());
        } catch(UnknownHostException ex) {
            System.out.println("I cannot find"+host);}
        catch(SocketException ex) {
            System.out.println("Could not connect to"+host);}
        catch(IOException ex) { System.out.println(ex);}}}}
```

```
$ java SocketInfo www.repubblica.it www.google.com
Connected to www.repubblica.it/18.66.196.94
on port 80 from port 56261 of/192.168.1.146
Connected to www.google.com/142.250.180.164
on port 80 from port 56262 of/192.168.1.146
```



# RIASSUNTO

identificazione di un servizio con cui comunicare, occorre individuare:

- la **rete** all'interno della quale si trova l'host su cui è in esecuzione il processo
- l'**host** all'interno della rete
- il **processo** in esecuzione sull'host
- rete ed host: identificati da Internet Protocol, mediante indirizzi IP
- processo: identificato da una **porta**, rappresentata da un intero da 0 a 65535
- ogni comunicazione è quindi individuata dalla **seguente 5-upla**:
  - il protocollo (TCP o UDP)
  - l'indirizzo IP del computer locale (client `sky3.cm.deakin.edu.au`, `139.130.118.5`)
  - la porta locale esempio: `5101`
  - l'indirizzo del computer remoto (server `res.cm.deakin.edu.au` `139.130.118.102`),
  - la porta remota: `5100` {tcp, `139.130.118.102`, `5100`, `139.130.118.5`, `5101`}



# ASSIGNMENT 5

Il log file di un web server contiene un insieme di linee, con il seguente formato:

150.108.64.57 - - [15/Feb/2001:09:40:58 -0500] "GET / HTTP 1.0" 200 2511

in cui:

- 150.108.64.57 indica l'host remoto, in genere secondo la dotted quad form
- [data]
- "HTTP request"
- status
- bytes sent
- eventuale tipo del client "Mozilla/4.0....."
- scrivere un'applicazione Weblog che prende in input il nome del log file (che sarà fornito) e ne stampa ogni linea, in cui ogni indirizzo IP è sostituito con l'hostname
- sviluppare due versioni del programma, la prima single-threaded, la seconda invece utilizza un thread pool, in cui il task assegnato ad ogni thread riguarda la traduzione di un insieme di linee del file. Confrontare i tempi delle due versioni.



# Reti e Laboratorio III

## Modulo Laboratorio III

**AA. 2022-2023**

**docente: Laura Ricci**

**[laura.ricci@unipi.it](mailto:laura.ricci@unipi.it)**

## **Lezione 6**

### **Stream Sockets for servers**

### **Volatile, Atomic**

**20/10/2022**

# SOCKET LATO SERVER

- esistono due tipi di socket TCP, lato server:
  - **welcome (passive, listening) sockets:** utilizzati dal server per accettare le richieste di connessione
  - **connection (active) sockets:** connettono il server ad un particolare client e supportano lo streaming di byte tra di essi
- il client crea un active socket per richiedere la connessione
- il server accetta una richiesta di connessione sul welcome socket
  - crea **un proprio connection socket** che rappresenta il punto terminale della sua connessione con il client
  - la comunicazione vera e propria avviene mediante la **coppia di active socket** presenti nel client e nel server

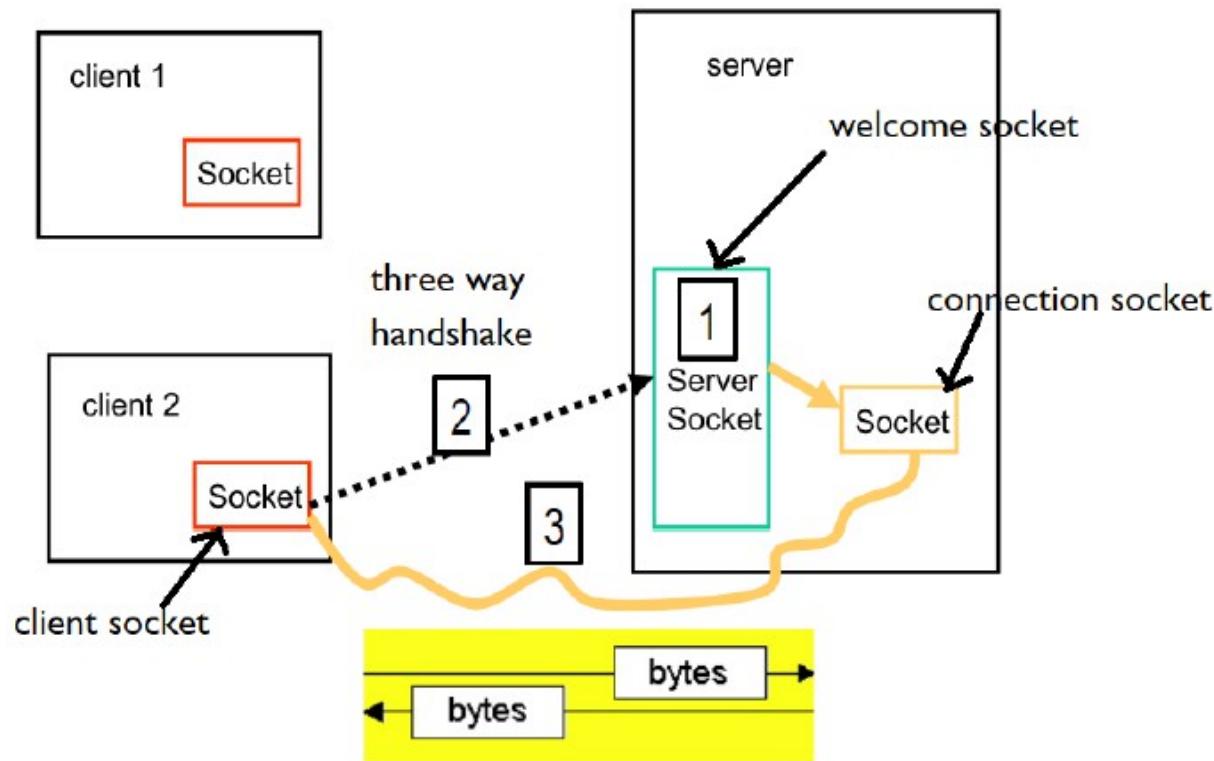


# SOCKET LATO SERVER

- il server pubblica un proprio servizio
  - gli associa un welcome socket, sulla porta remota PS, all'indirizzo IPS
  - usa un oggetto di tipo ServerSocket
- il client crea un Socket e lo connette all'endpoint IPS + PS
- la creazione del socket effettuata dal client produce in modo atomico la richiesta di connessione al server
  - three way handshake completamente gestito dal supporto
  - se la richiesta viene accettata,
    - il server crea un **socket dedicato** per l'interazione con quel client
    - tutti i messaggi spediti dal client vengono diretti **automaticamente** sul nuovo socket creato.

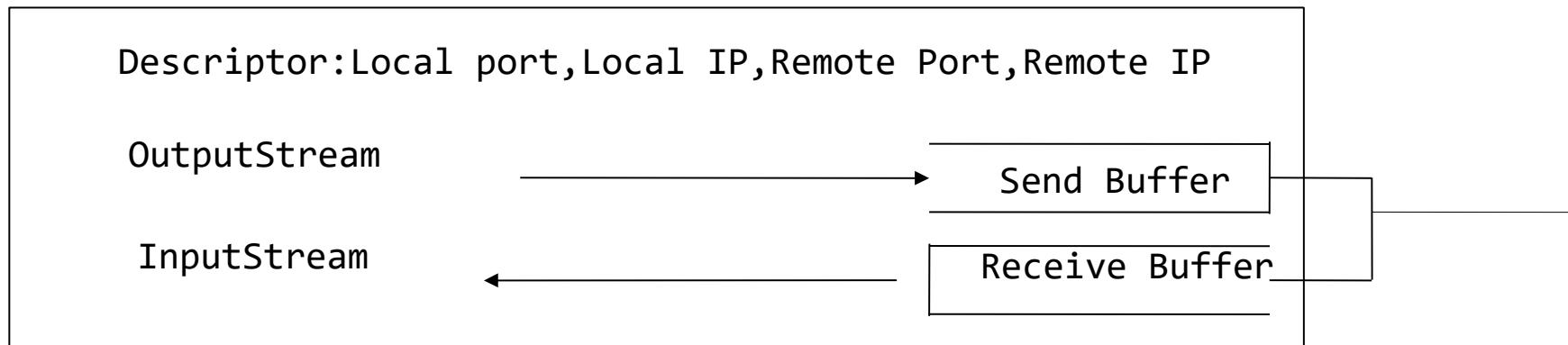


# SOCKET LATO SERVER



# STREAM BASED COMMUNICATION

- dopo che la richiesta di connessione viene accettata, client e server associano streams di bytes di input/output ai socket dedicati a quella connessione, poichè gli stream sono **unidirezionali**
  - a seconda del servizio può essere necessario un solo stream di output dal server verso il client, oppure una coppia di stream da/verso il client
- la comunicazione avviene mediante **lettura/scrittura di dati sullo stream**
- eventuale utilizzo di filtri associati agli stream



**Struttura del Socket TCP**



# JAVA STREAM SOCKET API: LATO SERVER

`java.net.ServerSocket`: costruttori

```
public ServerSocket(int port) throws BindException, IOException
```

```
public ServerSocket(int port, int length) throws BindException,  
IOException
```

- costruisce un listening socket, associandolo alla porta `port`.
- `length`: lunghezza della coda in cui vengono memorizzate le richieste di connessione.

se la coda è piena, ulteriori richieste di connessione sono rifiutate

```
public ServerSocket(int port, int length, InetAddress bindAddress)....
```

- permette di collegare il socket ad uno specifico indirizzo IP locale.
- utile per macchine dotate di più schede di rete, ad esempio un host con due indirizzi IP, uno visibile da Internet, l'altro visibile solo a livello di rete locale
- se voglio servire solo le richieste in arrivo dalla rete locale, associo il connection socket all'indirizzo IP locale



# JAVA STREAM SOCKET API: LATO SERVER

- accettare una nuova connessione dal **connection socket**

```
public Socket accept() throws IOException
```

metodo della classe **ServerSocket**.

- quando il processo server invoca il metodo `accept()`, pone il server in attesa di nuove connessioni.
- bloccante: se non ci sono richieste, il server si blocca (possibile utilizzo di time-outs)
- quando c'è almeno una richiesta, il processo si sblocca e costruisce un nuovo socket tramite cui avviene la comunicazione effettiva tra cliente e server



# PORT SCANNER LATO SERVER

- ricerca dei servizi attivi sull'host locale

```
import java.net.*;  
  
public class LocalPortScanner {  
  
    public static void main(String args[])  
    {for (int port= 1; port<= 1024; port++)  
        try {ServerSocket server = new ServerSocket(port);}  
        catch (BindException ex)  
            {System.out.println(port + "occupata");}  
  
        catch (Exception ex) {System.out.println(ex);}  
    } }
```



# CICLO DI VITA TIPICO DI UN SERVER

```
// instantiate the ServerSocket
ServerSocket servSock = new ServerSocket(port);
while (! done) // oppure while(true) {
    // accept the incoming connection
    Socket sock = servSock.accept();
    // ServerSocket is connected ... talk via sock
    InputStream in = sock.getInputStream();
    OutputStream out = sock.getOutputStream();
    //client and server communicate via in and out and do their work
    sock.close();
}
servSock.close();
```



# DAYTIME SERVER

```
import java.net.*;
import java.io.*;
import java.util.Date;
public class DayTimeServer {
    public final static int PORT = 1313;
    public static void main(String[] args) {
        try (ServerSocket server = new ServerSocket(PORT)) {
            while (true) {
                try (Socket connection = server.accept()) {
                    Writer out = new OutputStreamWriter(connection.getOutputStream());
                    Date now = new Date();
                    out.write(now.toString() +"\r\n");
                    out.flush();
                }
            }
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

porte 0-1023 privilegiate

si ferma qui ed aspetta, quando un client si connette restituisce un nuovo Socket

servizio della richiesta

inutile perchè si è usato il try with resources

try-with-resource: autoclose



# DAYTIME SERVER: CONNETTERSI CON TELNET

```
import java.net.*;  
import java.io.*;  
import java.util.Date;  
  
public class DayTimeServer {  
    public final static int PORT = 13;  
  
    public static void main(String[] args) {  
        try (ServerSocket server = new ServerSocket(PORT)) {  
            while (true) {  
                try (Socket connection = server.accept()) {  
                    Writer out = new OutputStreamWriter(connection.getOutputStream());  
                    Date now = new Date();  
                    out.write(now.toString() +"\r\n");  
                    out.flush();  
                    connection.close();  
                } catch (IOException ex) {}  
            } } catch (IOException ex) {System.err.println(ex);}}}
```

\$ telnet localhost 1333  
trying 127.0.0.1....  
connected to localhost  
San Oct 17 23:16:12 CEST 2021



# MULTITHREADED SERVER

- nello schema del lucido precedente, la fase “communicate and work” può essere eseguita in modo concorrente da più threads
- un thread per ogni client, gestisce le interazioni con quel particolare client
- il server può gestire le richieste in modo più efficiente
- tuttavia.....threads: anche se processi lightweigth ma tuttavia utilizzano risorse !
  - esempio: un thread che utilizza 1MB di RAM. 1000 thread simultanei possono causare problemi !
- Soluzioni alternative:
  - Thread Pooling
  - ServerSocketChannels di NIO



# A CAPITALIZER SERVICE: SERVER

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;
import java.util.concurrent.*;
public static void main(String[] args) throws Exception {
    try (ServerSocket listener = new ServerSocket(10000)) {
        System.out.println("The capitalization server is running...");
        ExecutorService pool = Executors.newFixedThreadPool(20);
        while (true) {
            pool.execute(new Capitalizer(listener.accept()));
        }
    }
}
```



# A CAPITALIZER SERVICE: SERVER

```
private static class Capitalizer implements Runnable {  
    private Socket socket;  
  
    Capitalizer(Socket socket) {  
        this.socket = socket; }  
  
    public void run() {  
        System.out.println("Connected: " + socket);  
        try (Scanner in = new Scanner(socket.getInputStream());  
             PrintWriter out = new PrintWriter(socket.getOutputStream(),  
                                              true))  
        { while (in.hasNextLine()) {  
                out.println(in.nextLine().toUpperCase()); }  
        } catch (Exception e) { System.out.println("Error:" + socket); }  
    }  
}
```



# A CAPITALIZER SERVICE: CLIENT

```
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;
public class CapitalizeClient {
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.err.println("Pass the server IP as the sole command line
                               argument");
        }
        Scanner scanner=null;
        Scanner in=null;
```



# A CAPITALIZER SERVICE: CLIENT

```
try (Socket socket = new Socket(args[0], 10000)) {  
    System.out.println("Enter lines of text then EXIT to quit");  
    scanner = new Scanner(System.in);  
    in = new Scanner(socket.getInputStream());  
    PrintWriter out = new PrintWriter(socket.getOutputStream(),  
                                      true);  
    boolean end=false;  
    while (!end) {  
        { String line= scanner.nextLine();  
          if (line.contentEquals("exit")) end=true;  
          out.println(line);  
          System.out.println(in.nextLine());}  
    }  
    finally {scanner.close(); in.close();}  
}  
}
```



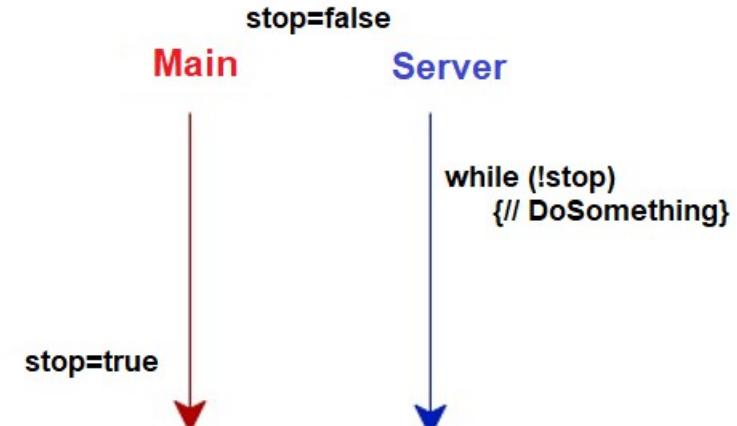
# ANCORA SUL MULTITHREADING

- Variabili volatile
- Variabili Atomic



# PERCHE' VOLATILE?

```
public class Server extends Thread  
{ boolean stop = false; int i;  
  
    public void run()  
    { while(! stop) {};  
  
        System.out.println("Server is stopped....");}  
  
    public void stopThread()  
    { stop = true;}}
```

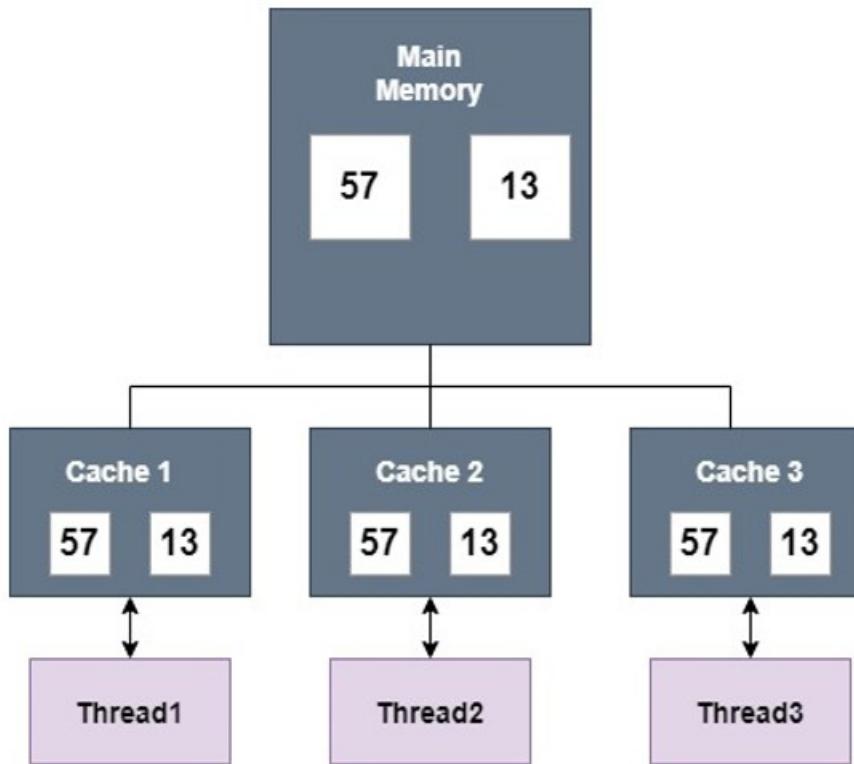


```
public class StoppingAThread  
{ public static void main(String args[]) throws InterruptedException  
{ Server myServer = new Server();  
myServer.start();  
  
System.out.println(Thread.currentThread().getName() + " is stopping Server thread");  
Thread.sleep(1000);  
myServer.stopThread();  
  
System.out.println(Thread.currentThread().getName() + " is finished now"); }}
```

il programma non termina!



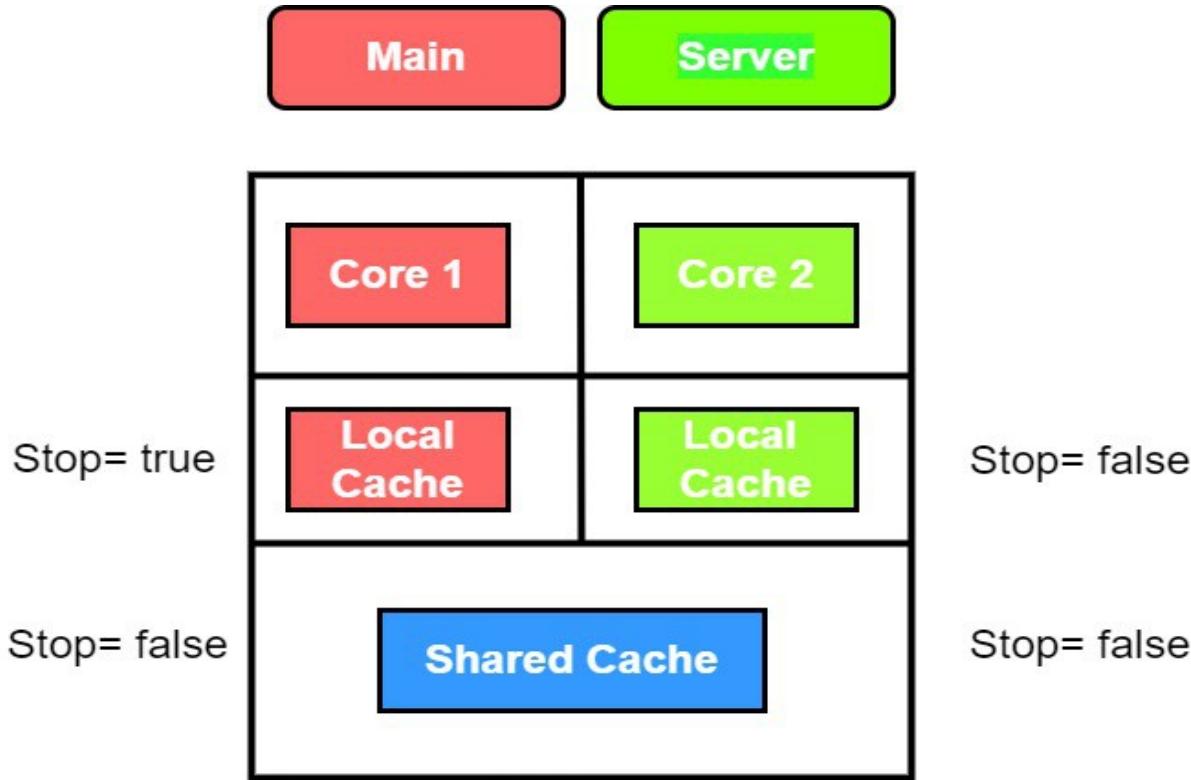
# IL PROBLEMA DELLA VISIBILITÀ



architettura di riferimento, utile per capire il problema della visibilità



# IL PROBLEMA DELLA VISIBILITÀ

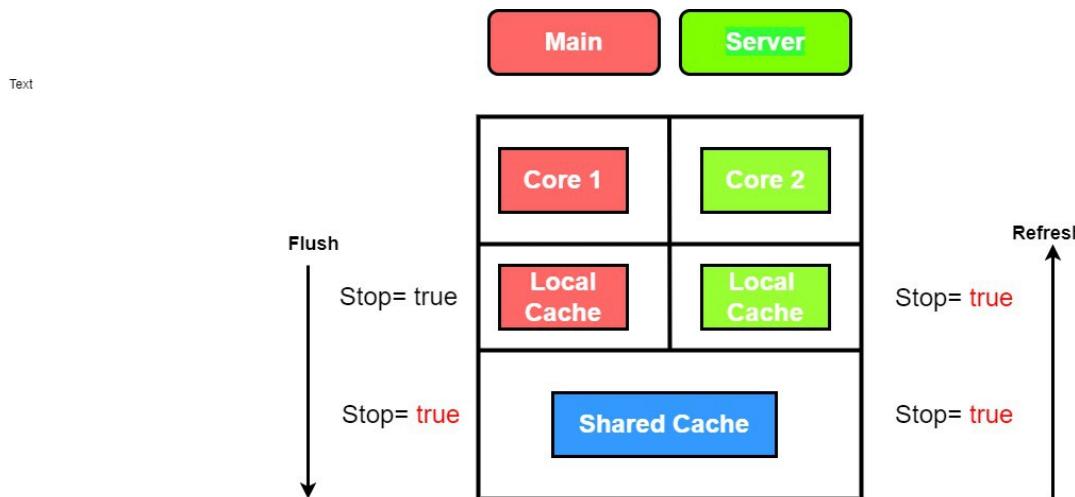


- quando il Main aggiorna Stop, è possibile che la modifica non sia riportata nella memoria condivisa
- il Main aggiorna la variabile Stop nella propria cache, ma la modifica non viene riportata nella memoria condivisa



# IL MODIFICATORE VOLATILE

- il problema riguarda la “**visibilità**” della modifica, non la sincronizzazione: read e write di un booleano sono atomiche
- modifichiamo la dichiarazione con la keyword **volatile**  
`volatile boolean stop = false`
  - l'aggiornamento ad una variabile **volatile** è sempre effettuato nella main memory
    - flush della cache
  - il valore della variabile **volatile** è sempre letto dalla memoria



# VOLATILE: VISIBILITÀ DI SCRITTURE

- tutte le scritture su una variabile volatile sono riportate direttamente nella memoria condivisa
- inoltre, tutte le variabili visibili dal thread che sta eseguendo la modifica vengono anche sincronizzate sulla memoria condivisa
- esempio:

```
this.nonVolatileVarA = 34;  
this.nonVolatileVarB = new String("Text");  
this.volatileVarC     = 300;
```

- quando viene eseguita la terza istruzione, sulla variabile volatileC, i valori delle due variabili non-volatile vengono sincronizzati in memoria condivisa



# VOLATILE: VISIBILITA' DI LETTURE

- quando viene letto il valore di una variabile volatile, viene garantito che tale valore venga letto direttamente dalla memoria condivisa
- inoltre, viene fatto il refresh di tutte le variabili visibili dal thread che sta eseguendo la lettura
- esempio:

```
c = other.volatileVarC;  
b = other.nonVolatileB;  
a = other.nonVolatileA;
```

- la prima istruzione è la lettura di una variabile volatile. Quando questa variabile viene letta dalla memoria, viene effettuato il refresh anche delle due altre variabili

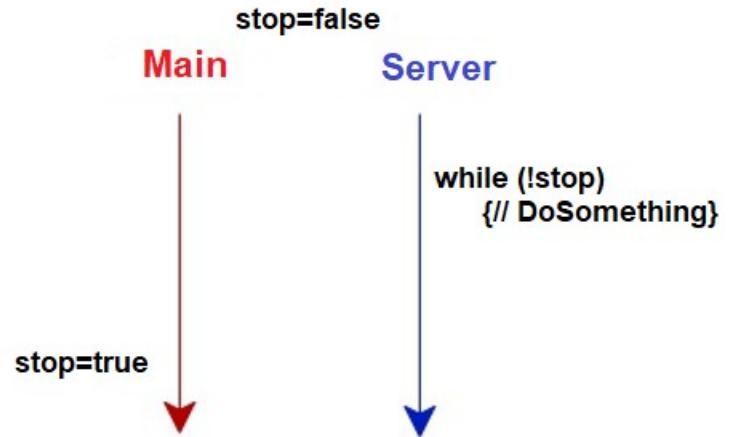


# UNA SOLUZIONE ALTERNATIVA

```
public class Server extends Thread  
  
{ Boolean stop = false; int i;  
  
    public void run()  
  
    { synchronized(stop) {};  
        while(! stop)  
            {synchronized(stop) {}};  
  
        System.out.println("Server is stopped....");}  
  
    public synchronized void stopThread(){ stop = true; } }
```

```
public class StoppingAThread  
  
{....}
```

sincronizzarsi sulla variabile stop ha lo stesso  
effetto di usare il modificatore volatile  
la variabile deve essere definita Boolean, per poter  
acquisire la lock

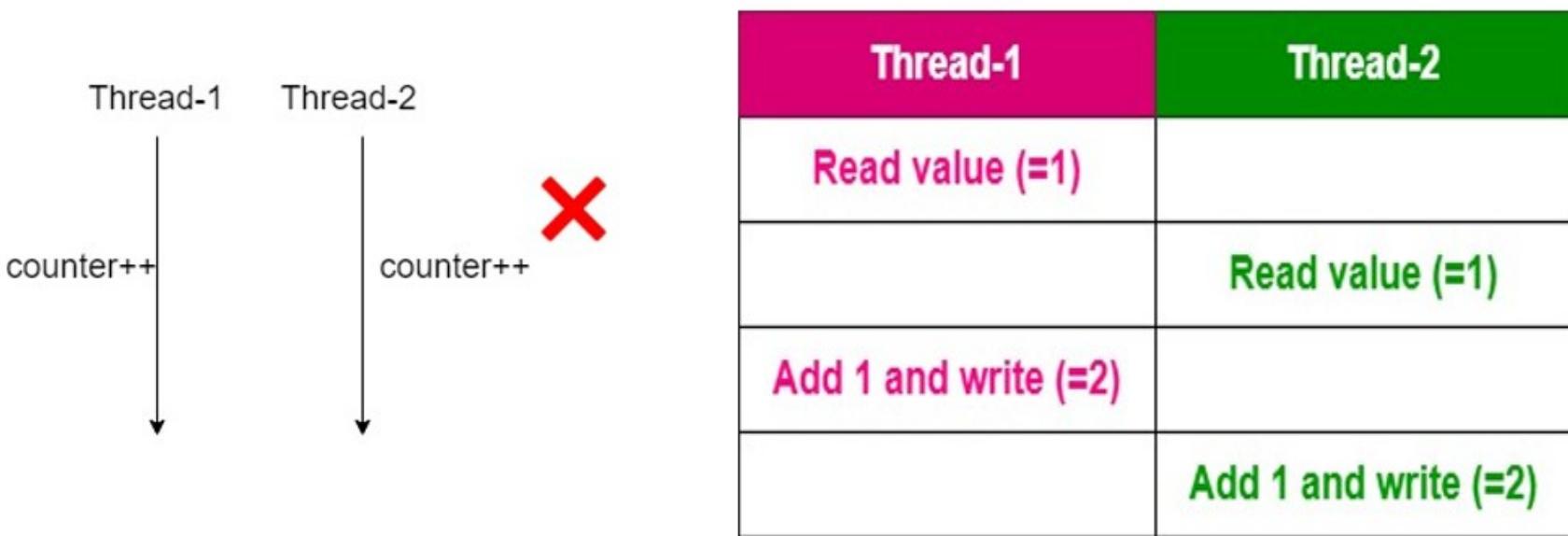


# SINCRONIZZAZIONE: VISIBILITÀ

- blocchi e metodi sincronizzati forniscono garanzia di visibilità simile a quella offerta dal modificatore volatile
- quando un thread entra in un metodo o blocco sincronizzato, viene effettuato un refresh di tutte le variabili visibili dal thread
- quando un thread esce da un blocco sincronizzato, tutte le variabili visibili dal thread vengono scritte in memoria
- monitor garantisce sia sincronizzazione che visibilità
- quando usare volatile?
  - quando la variabile condivisa è di tipo semplice
  - per acquisire la lock occorrerebbe fare il cast al corrispondente oggetto
  - tipico del pattern “termina l'esecuzione di un thread”



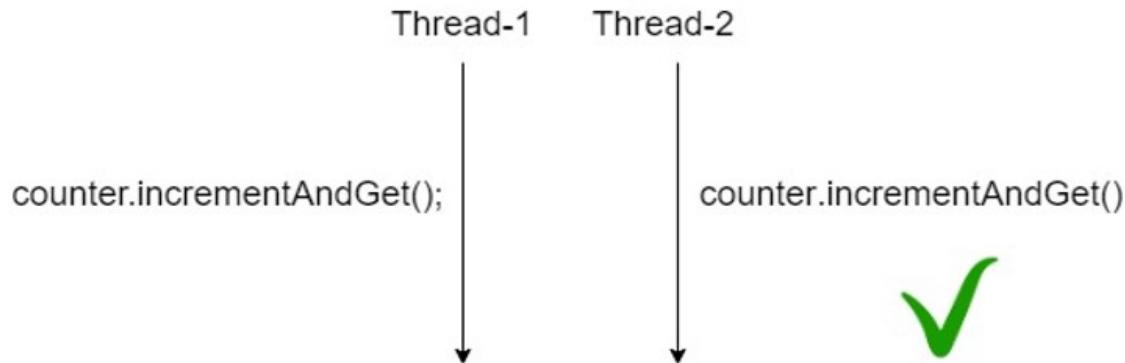
# SINCRONIZZAZIONE SU VARIABILI



- l'incremento di una variabile (volatile o meno) non è atomico
- se più thread provano ad incrementare una variabile in modo concorrente, un aggiornamento può andare perduto (anche se la variabile è volatile)
- ovviamente il problema può essere risolto con le lock
- soluzione alternativa: usare le variabili Atomic



# ATOMIC VARIABLES



```
AtomicInteger value = new AtomicInteger(1);
```

- operazioni atomiche che non richiedono sincronizzazioni esplicite o lock: è la JVM che garantisce la atomicità
  - incrementAndGet(): atomically increments by one
  - decrementAndGet(): atomically decrements by one
  - compareAndSet(int expectedValue, int newValue)
- molte altre classi
  - AtomicLong
  - AtomicBoolean



# ATOMIC VARIABLES: UN ESEMPIO

```
import java.util.concurrent.*; import java.util.concurrent.atomic.*;
public class AtomicIntegerExample {
public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(2);
    AtomicInteger atomicInt = new AtomicInteger();
    for(int i = 0; i < 10; i++){
        CounterRunnable runnableTask = new CounterRunnable(atomicInt);
        executor.submit(runnableTask);
    }
    executor.shutdown(); }

class CounterRunnable implements Runnable {
    AtomicInteger atomicInt;
    CounterRunnable(AtomicInteger atomicInt){this.atomicInt = atomicInt;}
    @Override
    public void run() {
        System.out.println("Counter- " + atomicInt.incrementAndGet());}}
```



# JAVA.UTIL.CONCURRENT.ATOMIC



# ASSIGNMENT 6: DUNGEON ADVENTURES

- sviluppare un'applicazione client server in cui il server gestisce le partite giocate in un semplice gioco, “Dungeon adventures” basato su una semplice interfaccia testuale
- ad ogni giocatore viene assegnato, ad inizio del gioco, un livello X di salute e una quantità Y di una pozione, X e Y generati casualmente
- ogni giocatore combatte con un mostro diverso. Anche al mostro assegnato a un giocatore viene associato, all'inizio del gioco un livello Z di salute generato casualmente



# ASSIGNMENT 6: DUNGEON ADVENTURES

- il gioco si svolge in round, ad ogni round un giocatore può
  - *combattere con il mostro*: il combattimento si conclude decrementando il livello di salute del mostro e del giocatore. Se  $LG$  è il livello di salute attuale del giocatore e  $MG$  quello del mostro, tale livello viene decrementato di un valore casuale  $X$ , con  $0 \leq X \leq LG$ . Analogamente, per il mostro si genera un valore casuale  $K$ , con  $0 \leq K \leq MG$ .
  - *bere una parte della pozione*, la salute del giocatore viene incrementata di un valore proporzionale alla quantità di pozione bevuta, che è un valore generato casualmente
  - *uscire dal gioco*. In questo caso la partita viene considerata persa per il giocatore
- il combattimento si conclude quando il giocatore o il mostro o entrambi hanno un valore di salute pari a 0.
- se il giocatore ha vinto o pareggiato, può chiedere di giocare nuovamente, se invece ha perso deve uscire dal gioco.



# ASSIGNMENT 6: DUNGEON ADVENTURES

- sviluppare una applicazione client server che implementi Dungeon adventures
  - il server riceve richieste di gioco da parte dei client e gestisce ogni connessione in un diverso thread
  - ogni thread riceve comandi dal client li esegue. Nel caso del comando “combattere”, simula il comportamento del mostro assegnato al client
  - dopo aver eseguito ogni comando ne comunica al client l'esito
  - comunica al client l'eventuale terminazione del del gioco, insieme con l'esito
- il client si connette con il server
  - chiede iterativamente all'utente il comando da eseguire e lo invia al server. I comandi sono i seguenti 1: combatti, 2: bevi pozione, 3: esci del gioco
  - attende un messaggio che segnala l'esito del comando
  - nel caso di gioco concluso vittoriosamente, chiede all'utente se intende continuare a giocare e lo comunica al server



# Reti e Laboratorio III

## Modulo Laboratorio III

**AA. 2022-2023**

**docente: Laura Ricci**

**[laura.ricci@unipi.it](mailto:laura.ricci@unipi.it)**

## **Lezione 7**

### **Serializzazione:**

# **JSON e Java native serialization**

**27/10/2022**

# SCRIVERE/LEGGERE OGGETTI DA STREAM

- gli oggetti esistono in memoria fino a che la JVM è in esecuzione:
  - per la loro persistenza al di fuori della JVM, occorre
    - creare una rappresentazione dell'oggetto indipendente dalla JVM
    - usando meccanismi di **serializzazione**
- ogni oggetto è caratterizzato da uno stato e da un comportamento
  - comportamento: specificato dai metodi della classe
  - stato: “vive” con l’istanza dell’oggetto
  - la serializzazione effettua il **flattening** dello **stato dell’oggetto**
  - la deserializzazione ricostruisce lo stato dell’oggetto

1 Object on the heap

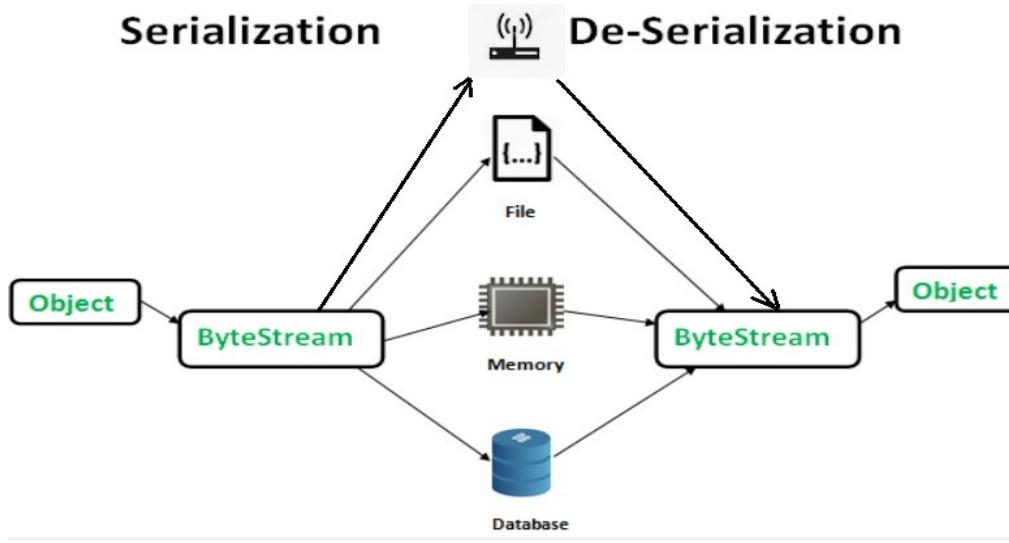


2 Object serialized



# PERSISTANZA ED INVIO DI OGGETTI

- l'oggetto serializzato può quindi essere scritto su un qualsiasi **stream di output**



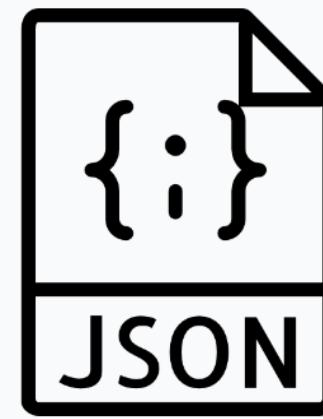
- come useremo la serializzazione in questo corso?
  - per inviare oggetti su uno stream che rappresenta una connessione TCP
  - per generare pacchetti UDP, si scrive l'oggetto serializzato su uno stream di byte e poi si genera un pacchetto UDP

# SERIALIZZAZIONE: INTEROPERABILITÀ

- caratteristica auspicabile di un formato di serializzazione
  - non vincolare chi scrive e chi legge ad usare lo stesso linguaggio
- la portabilità può limitare le potenzialità della rappresentazione:
  - una rappresentazione che corrisponde all'intersezione di tutti i vari linguaggi
- formati per la serializzazione dei dati che consentono l'interoperabilità tra linguaggi/macchine diverse
  - XML
  - JSON-JavaScript Object Notation
- JSON: formato nativo di Javascript, ha il vantaggio di essere espresso con una sintassi molto semplice e facilmente riproducibile

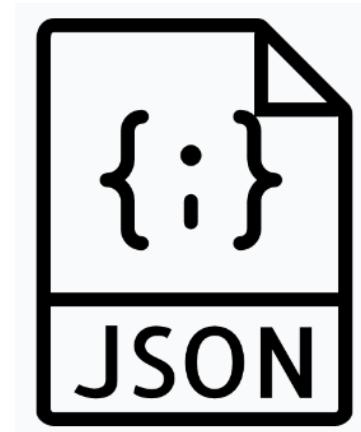
# JAVASCRIPT OBJECT NOTATION (JSON)

- formato lightweight per l'interscambio di dati, indipendente dalla piattaforma poichè è testo, scritto secondo la notazione JSON
  - non dipende dal linguaggio di programmazione
  - “self describing”, semplice da capire e facilmente parsabile
- basato su 2 strutture:
  - coppie (chiave: valore)
  - liste ordinate di valori
- una risorsa JSON ha una struttura ad albero
  - composizione ricorsiva di coppie e liste



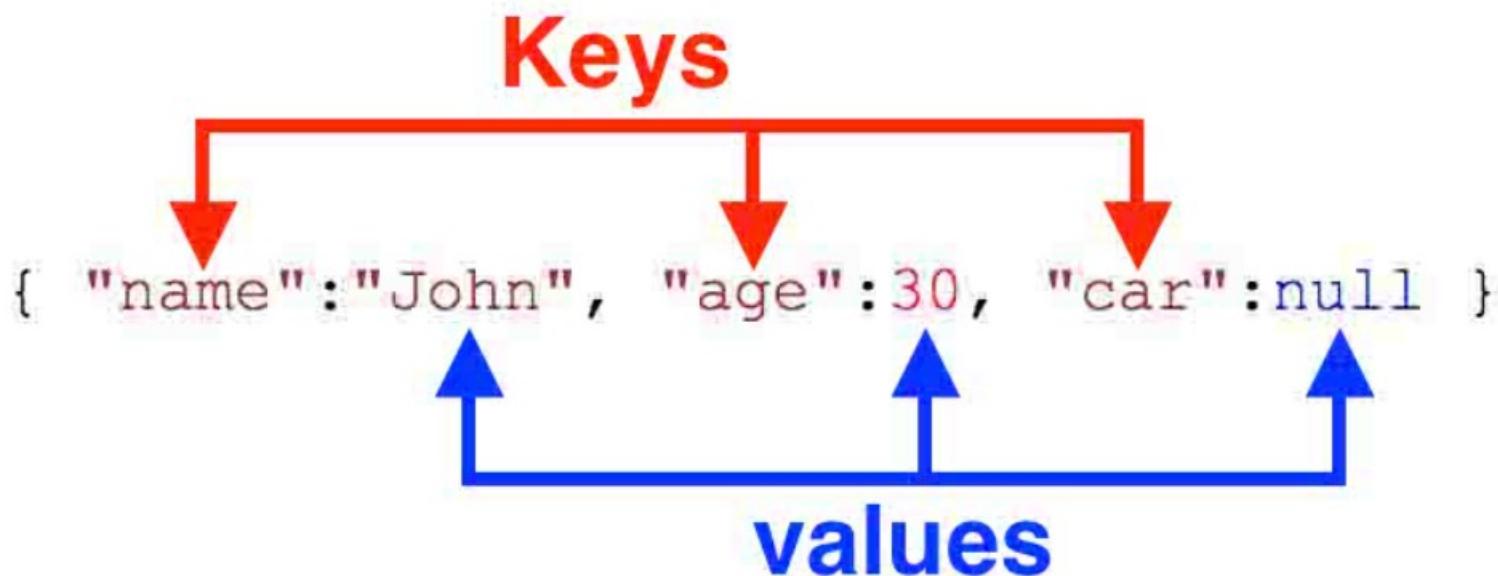
# JAVASCRIPT OBJECT NOTATION

- coppie (chiave: valore)
  - le chiavi devono esser stringhe { "name": "John" }
- i tipi di dato ammissibili per i valori sono:
  - String
  - Number (int o float)
  - object (JSON object, la struttura può essere ricorsiva)
  - Array
  - Boolean
  - null



# JSON OBJECT

- una serie non ordinata di coppie (*nome, valore*)
- delimitato da parentesi graffe
- le coppie sono separate da virgole



# JSON ARRAY

- una raccolta ordinata di valori  
["Ford", "BMW", "Fiat"]
- delimitato da parentesi quadre e i valori sono separati da virgola.
  - un valore può essere di tipo string, un numero, un booleano, un oggetto JSON o un array.
  - queste strutture possono essere annidate.
- mapping diretto con [array](#), [list](#), [vector](#), [di JAVA](#) etc.



# JSON: STRUTTURA RICORSIVA

```
JSON Object → {  
    "company": "mycompany",  
    "companycontacts": { ← Object Inside Object  
        "phone": "123-123-1234",  
        "email": "myemail@domain.com"  
    },  
    "employees": [ ← JSON Array  
        {  
            "id": 101,  
            "name": "John",  
            "contacts": [  
                "email1@employee1.com",  
                "email2@employee1.com"  
            ]  
        },  
        {  
            "id": 102, ← Number Value  
            "name": "William",  
            "contacts": null ← Null Value  
        }  
    ]  
}
```

String Value

Object Inside Object

JSON Array

Array Inside Array

Number Value

Null Value

# JSON E XML: CONFRONTO

## JSON Example

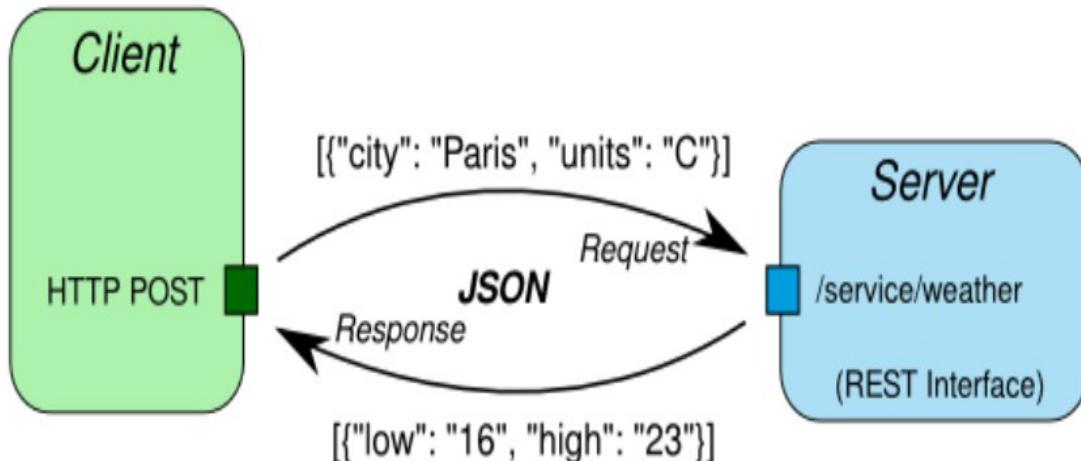
```
{"employees": [  
    {"firstName": "John", "lastName": "Doe"},  
    {"firstName": "Anna", "lastName": "Smith"},  
    {"firstName": "Peter", "lastName": "Jones"}  
]}
```

## XML Example

```
<employees>  
    <employee>  
        <firstName>John</firstName> <lastName>Doe</lastName>  
    </employee>  
    <employee>  
        <firstName>Anna</firstName> <lastName>Smith</lastName>  
    </employee>  
    <employee>  
        <firstName>Peter</firstName> <lastName>Jones</lastName>  
    </employee>  
</employees>
```

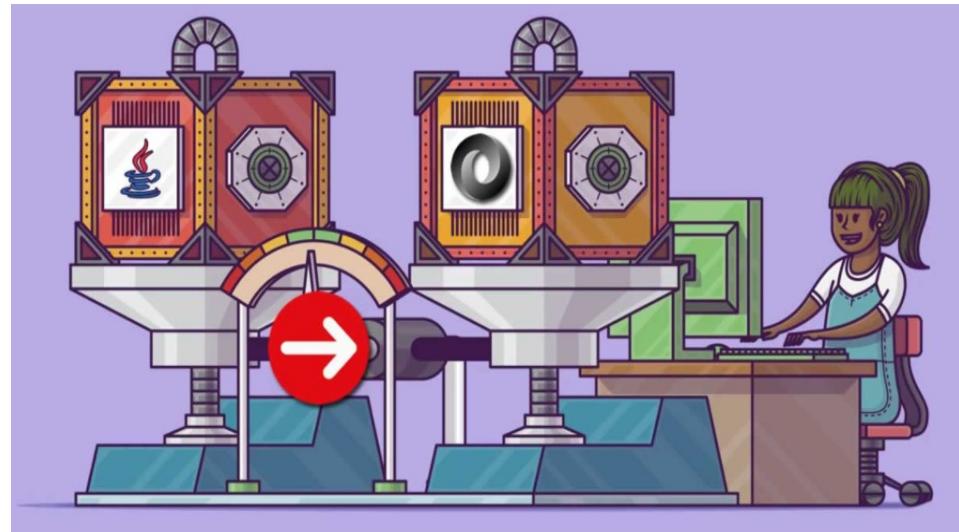


# JSON/REST/HTTP



- client e server interagiscono mediante interfaccia REST
- JSON è in genere il formato dei dati scambiati

- cosa accade se la applicazione è scritta in JAVA?
- necessaria trasformazione JAVA/JSON e viceversa



# DA JAVA A JSON

JSON

```
=====
```

```
{  
    "id":1,  
    "name": SiAm,  
    "color": Cream ,  
    "breed": Siamese  
}
```

JSON string is understood by any program because it's

INTEROPERABLE – program and platform independent

Java Obj

```
=====
```

```
1L  
SiAm  
Cream  
Siamese
```

Representation of ' cat obj'

While it's clear to us that our cat object has: 1L,SiAm, Cream, Siamese

only our java application will understand what these things are.

Our JSON string is understood by every application

Quali librerie per la traduzione?

- **GSON**
- **JACKSON**
- **JSON-Simple**
  - leggera e semplice, ma... scarsa documentazione
  - FastJSON
  - ...



# GSON: GOOGLE GSON

- libreria per serializzare/deserializzare oggetti Java in/da JSON
  - toJson() e fromJson() semplici metodi per la serializzazione e la deserializzazione
  - serializzazione semplice, deserializzazione richiede reflection
  - supporto per JAVA generics ed oggetti arbitrariamente complessi
  - possibile personalizzare la serializzazione
- scaricare JAR ed inserirlo come libreria esterna nel progetto
  - scaricare GSON
  - importare la libreria in Eclipse
  - tasto destro sul nome del progetto → JAVA Build Path → Add libraries  
→ User Library selezionare la libreria scaricata



# SERIALIZZAZIONE/DESERIALIZZAZIONE CON GSON

- GSON fornisce il supporto per trasformare oggetti JSON in oggetti JAVA e viceversa
  - una classe JAVA con la stessa struttura dell'oggetto JSON
- consideriamo il seguente oggetto JSON e la corrispondente classe JAVA

```
{ "name": "Alice",
  "age" : 45
}
```

```
class Person
{ String name;
  int age; }
```

- metodi base offerti da GSON per il passaggio da JAVA a JSON sono
  - *serializzazione*: dato un oggetto JAVA, restituisce la rappresentazione JSON dell'oggetto  
`toJson(Object src)`
  - *deserializzazione*: da una stringa in formato JSON ad oggetto JAVA  
`fromJson(String json, Class<T> classOfT)`  
`fromJson(JsonElement json, java.lang.reflect.Type typeOfT)`

# GSON: GOOGLE GSON

```
import com.google.gson.Gson;

public class GSONJava {

    public static void main(String args[]) { // Serialization

        Gson gson = new Gson();

        System.out.println(gson.toJson(1));                                // ==> 1
        System.out.println(gson.toJson("abcd"));                            // ==> "abcd"
        int[] values = { 1 };
        System.out.println(gson.toJson(values));                           // ==> [1]

        // Deserialization

        int one = gson.fromJson("1", int.class);
        System.out.println(one);                                         // ==> 1
        Long oneL = gson.fromJson("1", Long.class);                     // ==> 1
        System.out.println(oneL);

        Boolean f = gson.fromJson("false", Boolean.class);   // ==> false
        System.out.println(f);

        String str = gson.fromJson("\\"abc\\\"", String.class); // ==> abc
        System.out.println(str);}}
```



# SERIALIZZAZIONE DI OGGETTI SEMPLICI

```
import com.google.gson.Gson;  
  
public class Person  
  
    { String name;  
        int age;  
  
        Person(String name, int age)  
        { this.name = name;  
            this.age = age; }  
  
    }  
  
public class ToJSON  
  
    public static void main(String[] args)  
    {  
  
        Person p = new Person("Alice", 59);  
        Gson gson = new Gson();  
        String json = gson.toJson(p);  
  
        System.out.println(json);  
    }  
}
```

serializzazione

```
Gson gson = new Gson();  
String json = gson.toJson(p);
```

```
System.out.println(json);
```

```
$java ToJSON
```

```
{"name": "Alice", "age": 59}
```



# SERIALIZZAZIONE: FORMATTARE L'OUTPUT

```
import com.google.gson.Gson;  
  
public class Person  
  
    { String name;  
        int age;  
        Person(String name, int age)  
            { this.name = name;  
                this.age = age; }  
    }  
  
public class ToJSON  
  
    public static void main(String[] args)  
    {  
        Person p = new Person("Alice", 59);  
        Gson gson = new GsonBuilder()  
                    .setPrettyPrinting()  
                    .create();  
        String json = gson.toJson(p);  
        System.out.println(json);  
    }
```

```
$java ToJSON  
{  
    "name": "Alice",  
    "age": 59  
}
```



# SERIALIZZARE OGGETTI COMPOSTI

```
import java.util.*;  
  
public class RestaurantWithMenu {  
  
    String name;  
  
    List<RestaurantMenuItem> menu;  
  
    public RestaurantWithMenu (String name, List<RestaurantMenuItem> menu )  
  
    {this.name=name;  
  
     this.menu= menu;  
  
    }  
  
import java.util.*;  
  
public class RestaurantMenuItem {  
  
    String description;  
  
    float price;  
  
    public RestaurantMenuItem (String description, float price)  
  
    {this.description=description;  
  
     this.price= price;      }  
  
    public String toString() {return description+price;}}  
 
```



# SERIALIZZARE OGGETTI COMPOSTI

```
import java.util.*;  
  
import com.google.gson.*;  
  
public class Restaurants {  
  
    public static void main (String args[])  
  
    { List<RestaurantMenuItem> menu = new ArrayList<>();  
  
        menu.add(new RestaurantMenuItem("Spaghetti", 9.99f));  
  
        menu.add(new RestaurantMenuItem("Steak", 14.99f));  
  
        menu.add(new RestaurantMenuItem("Salad", 6.99f));  
  
        RestaurantWithMenu restaurant =  
  
                new RestaurantWithMenu("AllWhatYouCanEat", menu);  
  
        Gson gson = new GsonBuilder()  
  
                .setPrettyPrinting()  
  
                .create();  
  
        String restaurantJson= gson.toJson(restaurant);  
  
        System.out.println(restaurantJson);
```



# SERIALIZZAZIONE DELL'OGGETTO

```
{  
  "name": "AllWhatYouCanEat",  
  "menu": [  
    {  
      "description": "Spaghetti",  
      "price": 9.99  
    },  
    {  
      "description": "Steak",  
      "price": 14.99  
    },  
    {  
      "description": "Salad",  
      "price": 6.99  
    }  
  ]  
}
```



# SERIALIZZARE OGGETTI COMPOSTI

```
import java.util.*;  
  
import com.google.gson.Gson; import com.google.gson.GsonBuilder;  
  
enum Degree_Type { TRIENNALE, MAGISTRALE}  
  
public class Student {  
  
    private String firstName;  
  
    private String lastName;  
  
    private int studentID;  
  
    private String email;  
  
    private List<String> courses;  
  
    private Degree_Type Dg;  
  
    public Student(String FName, String LName, int SID, String email,  
                  List<String> Clist, Degree_Type DG )  
  
    {this.lastName=LName; this.lastName=LName; this.studentID=SID;  
     this.email= email; this.courses=Clist; this.Dg=DG;};  
  
    public String toString()  
  
    { return "name:"+firstName+ " surname:"+lastName+ " ID:"+studentID+ "  
             email:"+email+ " corsi:"+courses+ " Degree:"+Dg; }  
  
    // Metodi getter e setter
```

# SERIALIZZARE COMPOSIZIONE DI OGGETTI

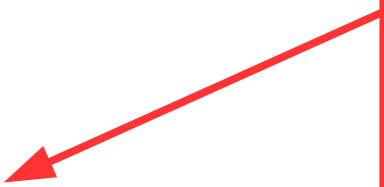
```
public static void main (String args[])
{
List <String> ComputerScienceCourses = Arrays.asList("Reti", "Architetture");
List <String> MathCourses = Arrays.asList("Analisi", "Statistica");
// Instantiating students
Student max = new Student("Mario", "Rossi", 1254, "mario.rossi@uni1.it",
ComputerScienceCourses, Degree_Type.TRIENNALE);
Student amy = new Student("Anna", "Bianchi", 1328, "anna.bianchi@uni1.it",
MathCourses, Degree_Type.MAGISTRALE);
// Instantiating Gson
Gson gson = new GsonBuilder()
.setPrettyPrinting()
.create();
// Converting JAVA to JSON
String marioJson = gson.toJson(mario);
String annaJson = gson.toJson(anna);
System.out.println(marioJson);
System.out.println(annaJson);}}
```

```
$java Student
{
  "lastName": "Rossi",
  "studentID": 1254,
  "email": "mario.rossi@uni1.it",
  "courses": [
    "Reti",
    "Architetture"
  ],
  "Dg": "TRIENNALE"
}
{
  "lastName": "Bianchi",
  "studentID": 1328,
  "email": "anna.bianchi@uni1.it",
  "courses": [
    "Analisi",
    "Statistica"
  ],
  "Dg": "MAGISTRALE"
}
```



# DESERIALIZZARE STRUTTURE JSON RICORSIVE

```
{  
  "name": "AllWhatYouCanEat",  
  "menu": [  
    {  
      "description": "Spaghetti",  
      "price": 9.99  
    },  
    {  
      "description": "Steak",  
      "price": 14.99  
    },  
    {  
      "description": "Salad",  
      "price": 6.99  
    }  
  ] }
```



creiamo il file `restaurant.json`  
in cui memorizziamo la struttura JSON  
a fianco  
~~nella slide successiva vedremo come~~  
deserializzare questa struttura

# DESERIALIZZARE STRUTTURE JSON RICORSIVE

```
import com.google.gson.*; import java.io.*; import java.util.*;

public class GSONComplexObject {

public static void main(String[] args) {

File input = new File("restaurant.json");

try {

JsonElement fileElement = JsonParser.parseReader(new FileReader(input));

JsonObject fileObject = fileElement.getAsJsonObject();

//extracting basic fields

String identifier = fileObject.get("name").getAsString();

System.out.println("name is="+identifier);

JSONArray jsonArrayOfItems =fileObject.get("menu").getAsJSONArray();

List <RestaurantMenuItem> menuitems = new ArrayList <RestaurantMenuItem>();
```



# DESERIALIZZAZIONE COMPOSIZIONE DI OGGETTI

```
for (JsonElement menuElement: jsonArrayOfItems) {  
    //Get the JsonObject  
    JsonObject itemJsonObject = menuElement.getAsJsonObject();  
    String desc= itemJsonObject.get("description").getAsString();  
    float price = itemJsonObject.get("price").getAsFloat();  
    RestaurantMenuItem restaurantel = new RestaurantMenuItem(desc, price);  
    menuitems.add(restaurantel);  
}  
  
System.out.println("Items are"+menuitems);  
}  
  
catch (FileNotFoundException e) {e.printStackTrace();}  
catch (Exception e) {e.printStackTrace();}}
```

Stampa

name is>AllWhatYouCanEat

Items are[Spaghetti 9.99, Steak 14.99, Salad 6.99]



# DESERIALIZZAZIONE CON REFLECTION

- nell'esempio precedente la deserializzazione avviene accedendo ai singoli campi dell'oggetto JSON
- è possibile deserializzare l'intera struttura JSON trasformandola in un solo passo nel corrispondente oggetto JAVA?
- la deserializzazione in un oggetto composto richiede in generale informazioni aggiuntive
- occorre indicare, a run time, il tipo (la classe) utilizzata per la deserializzazione
- uso del meccanismo delle Reflection
  - capacità di analizzare ed interagire a run time con le classi
  - in particolare utilizzo del tipo Type e della funzione getType per determinare a run time il tipo di una classe

# DESERIALIZZAZIONE CON REFLECTION

```
import com.google.gson.*;  
import java.lang.reflect.*;  
import com.google.gson.reflect.*;  
  
public class RestaurantRefelection {  
  
    public static void main(String[] args) {  
  
        try {  
  
            String JsonRestaurant = "{\"name\":\"AllWhatYouCanEat\", \"menu\":"  
                + "[{\"description\":\"Spaghetti\", \"price\":9.99}, "  
                + "{\"description\":\"Steak\", \"price\":14.99}, "  
                + "{\"description\":\"Salad\", \"price\":6.99}]}" ;  
  
            Reflection  
            Gson gson = new Gson();  
  
            Type restaurantType = new TypeToken<RestaurantWithMenu>() {}.getType();  
            RestaurantWithMenu rm=gson.fromJson(JsonRestaurant, restaurantType);  
  
            System.out.println(rm);  
        }  
  
        catch (Exception e) {e.printStackTrace();}  
    }  
}
```

# INTERAZIONE DI RETE IN JSON

- JSON è un formato interoperabile utilizzato soprattutto per scambiare dati in rete
  - nel caso del progetto sia il client che il server saranno implementati in JAVA, per cui potrebbe essere utilizzata anche la serializzazione nativa di JAVA.
  - ma è possibile considerare anche un client/server JAVA che riceve dati JSON generati da una applicazione implementata con un linguaggio diverso
- due possibili scenari
  - il client/server invia al server/client un oggetto JSON che rappresenta una singola entità
    - esempio: il client invia ad un servizio social i dati del proprio profilo
  - il client/server invia al server/client un oggetto JSON che contiene la rappresentazione di uno stream di entità
    - esempio: un server invia al client tutti i post pubblicati sul suo profili social

# CLIENT JSON: INVIO SINGOLA ENTITA'

```
import java.util.*; import java.net.*; import java.io.*; import com.google.gson.*;  
  
public class Restaurants {  
  
    public static void main (String args[])  
    { if (args.length!=2) return;  
  
        String host= args[0]; int port = Integer.parseInt(args[1]); DataOutputStream os;  
        try (Socket s= new Socket(host, port)){  
            os= new DataOutputStream(s.getOutputStream());      try with resources  
  
            List<RestaurantMenuItem> menu = new ArrayList<>();  
  
            menu.add(new RestaurantMenuItem("Spaghetti", 9.99f));  
  
            menu.add(new RestaurantMenuItem("Steak", 14.99f));  
  
            menu.add(new RestaurantMenuItem("Salad", 6.99f));  
  
            RestaurantWithMenu restaurant = new  
                RestaurantWithMenu("AllWhatYouCanEat", menu);  
  
            Gson gson = new Gson();  
  
            String restaurantJson= gson.toJson(restaurant);  
            os.writeUTF(restaurantJson);  
        } catch(Exception e) {};}  
    }  
}
```

try (Socket s= new Socket(host, port)){

os= new DataOutputStream(s.getOutputStream()); try with resources

invio oggetto JSON sullo stream (una stringa)

# SERVER JSON: RICEZIONE DI UNA SINGOLA ENTITA'

```
import java.net.*; import java.io.*; import com.google.gson.*; import java.lang.reflect.*;

import com.google.gson.reflect.*;

public class ServerRestaurant {

public static void main (String args[])

{ if (args.length!=1) return;

int port = Integer.parseInt(args[0]);

try (ServerSocket s= new ServerSocket(port)) {

    DataInputStream is= new DataInputStream(s.accept().getInputStream());

    System.out.println("accettato");

    String json= is.readUTF();

    Gson gson = new Gson();

    Type restaurantType =new TypeToken<RestaurantWithMenu>() {}.getType();

    RestaurantWithMenu rm=gson.fromJson(json, restaurantType);

    System.out.println(rm);

}

catch (Exception e) {}
```



# GSON STREAMING API

- streaming: utile supporto quando si invia uno stream di oggetti JSON
- immaginiamo di avere un file JSON di 1.5 G che contiene un insieme di documenti, con i relativi metadati
  - un unico oggetto JSON contenente tutti i documenti?
  - caricare tutto l'oggetto e deserializzarlo con i metodi visti è improponibile, perchè il file avrebbe grosse dimensioni
- GSON streaming offre metodi il caricamento incrementale di parti dell'oggetto
- utile
  - quando l'oggetto ha dimensione troppo grossa
  - quando non si dispone dell'intero oggetto da deserializzare, perchè ad esempio l'oggetto viene inviato in streaming su una connessione di rete
- metodi: JsonReader, JsonWriter

# JSON STREAMING API: JSONWRITER

```
import com.google.gson.stream.JsonWriter; import java.io.FileWriter; import java.io.IOException;

public class GsonStreamWriter {

    public static void main(String... args){

        JsonWriter writer;

        try { writer = new JsonWriter(new FileWriter("result.json"));

            writer.beginObject(); // {

            writer.name("name").value("Steve"); // "name": "Steve"

            writer.name("surname").value("Jobs"); // "surname": "Job"

            writer.name("birthyear").value(1955); // "birthyear": 2016

            writer.name("skills"); // "skills": [

            writer.beginArray(); // [

            writer.value("JAVA"); // "JAVA"

            writer.value("Python"); // "Python"

            writer.value("Rust"); // "Rust"

            writer.endArray(); // ]

            writer.endObject(); // }

            writer.close();

        } catch (IOException e) { System.err.print(e.getMessage());}}}
```



# JSON STREAMING API: JSONREADER

```
import com.google.gson.stream.JsonReader; import java.io.FileNotFoundException;
import java.io.FileReader; import java.io.IOException;
public class GSONStreamReader {
    public static void main(String... args){
        JsonReader reader;
        try {
            reader = new JsonReader(new FileReader("result.json"));
            reader.beginObject();
            while (reader.hasNext()){
                String name = reader.nextName();
                if ("name".equals(name)){
                    System.out.println(reader.nextString());
                } else if ("surname".equals(name)){
                    System.out.println(reader.nextString());
                } else if ("birthyear".equals(name)){
                    System.out.println(reader.nextString());
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



# JSON STREAMING API: JSONREADER

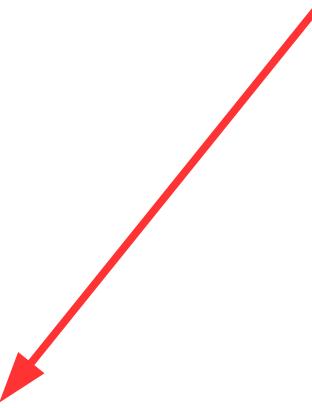
```
    } else if ("skills".equals(name))  
    { reader.beginArray();  
        while (reader.hasNext()) {  
            System.out.println("\t" + reader.nextString());}  
        reader.endArray();  
    } else {  
        reader.skipValue();  
    }  
    reader.endObject();  
    reader.close();  
} catch (FileNotFoundException e) { System.err.print(e.getMessage());}  
} catch (IOException e) { System.err.print(e.getMessage());}}
```



# GSON STREAMING API: JSONREADER

```
Steve  
Jobs  
1955  
JAVA  
Python  
Rust
```

Stampa prodotta dal programma



# SERIALIZZAZIONE JAVA: HOW TO DO

- **Serializable Interface**
  - per rendere un oggetto “persistente”, l'oggetto deve implementare l'interfaccia **Serializable**
  - marker interface: nessun metodo, solo informazione su un oggetto per il compilatore e la JVM
  - controllo limitato sul meccanismo di linearizzazione dei dati
  - tutti i tipi di dato primitivi sono serializzabili
  - gli oggetti, se implementano **Serializable**, sono serializzabili
    - a parte alcuni oggetti....(vedi slide successive)
- **Externalizable Interface**
  - estende **Serializable**
  - consente creare un proprio protocollo di serializzazione
    - ottimizzare la rappresentazione serializzata dell'oggetto
    - implementazione metodi **readExternal** e **writeExternal**

# SERIALIZZAZIONE JAVA: HOW TO DO

```
import java.io.Serializable;  
import java.util.Date;  
import java.util.Calendar;  
public class PersistentTime implements Serializable  
{ private static final long serialVersionUID = 1;  
    private Date time;  
    public PersistentTime()  
    {time = Calendar.getInstance().getTime(); }  
    public Date getTime()  
    {return time; } }
```

in rosso le parti relative alla serializzazione

**Regola #1:** per serializzare un oggetto persistente la classe di cui l'oggetto è istanza deve implementare l'interfaccia Serializable oppure ereditare l'implementazione dalla sua gerarchia di classi



# SERIALIZZAZIONE JAVA: HOW TO DO

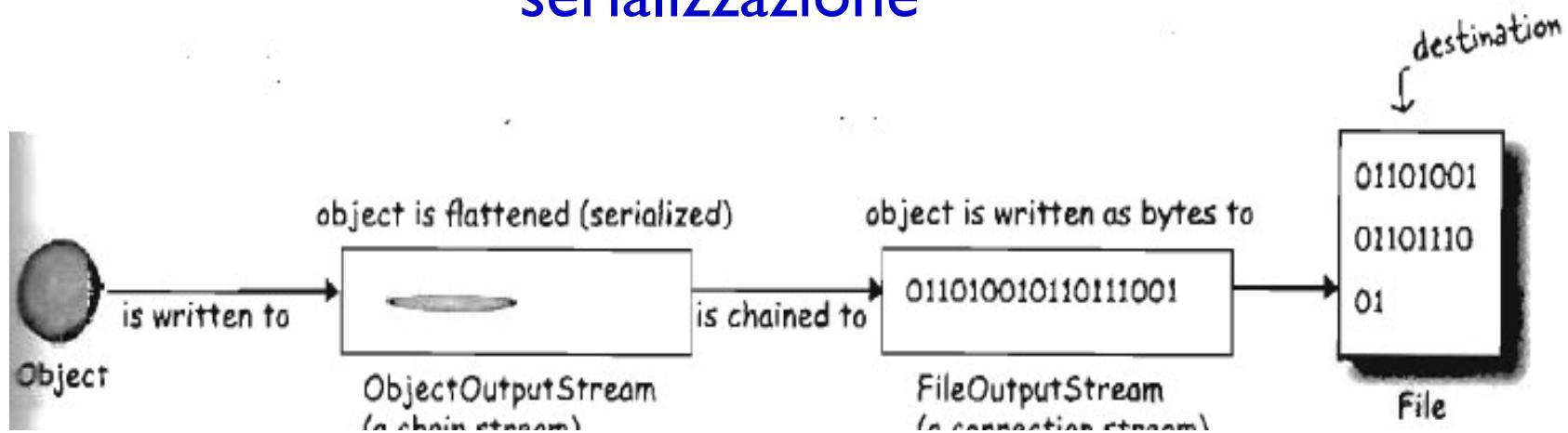
```
import java.io.*;  
  
public class FlattenTime  
{public static void main(String [] args)  
{String filename = "time.ser";  
 if(args.length > 0) { filename = args[0]; }  
 PersistentTime time = new PersistentTime();  
 try{  
     FileOutputStream fos = new FileOutputStream(filename);  
     ObjectOutputStream out = new ObjectOutputStream(fos);  
     { out.writeObject(time);}  
     catch(IOException ex) {ex.printStackTrace();  
 }}
```

- la serializzazione vera e propria è gestita dalla classe `ObjectOutputStream`
- tale stream deve essere concatenato con uno stream di bytes, che può essere un `FileOutputStream`, uno stream di bytes associato ad un socket, uno stream di byte generato in memoria,...

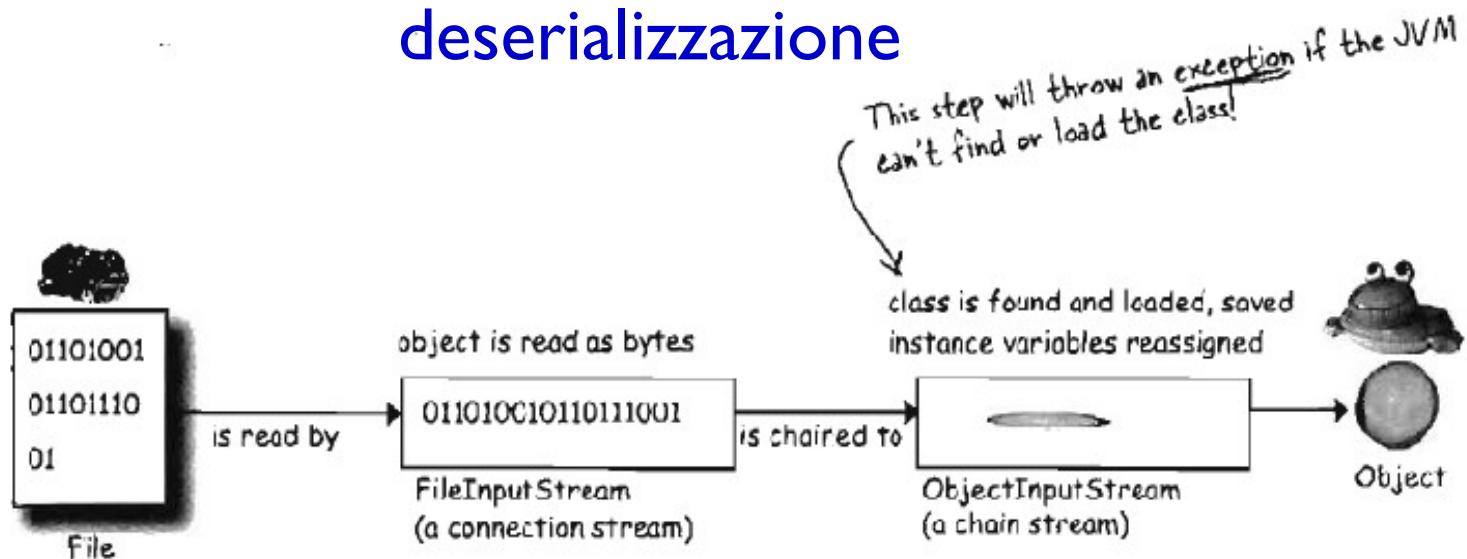


# SERIALIZZAZIONE E DESERIALIZZAZIONE

## serializzazione



## deserializzazione



# DESERIALIZZAZIONE

```
public class InflateTime
{
    public static void main(String [] args)
    {
        String filename = "time.ser";
        if(args.length > 0)
            {filename = args[0]; }

        PersistentTime time = null; FileInputStream fis = null;
        ObjectInputStream in = null;

        Try {
            FileInputStream fis = new FileInputStream(filename);
            ObjectInputStream in = new ObjectInputStream(fis);
            time = (PersistentTime)in.readObject();
        }

        catch(IOException ex)
        {
            ex.printStackTrace();
        }
        catch(ClassNotFoundException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

in rosso le parti relative alla **deserializzazione**



# DESERIALIZZAZIONE

```
// print out restored time  
System.out.println("Flattened time: " + time.getTime());  
System.out.println();  
// print out the current time  
System.out.println("Current time: "+  
                    Calendar.getInstance().getTime());}  
}
```

Output ottenuto:

Flattened time: Mon Mar 12 19:11:55 CET 2012

Current time: Mon Mar 12 19:16:24 CET 2012

**ClassNotFoundException**: l'applicazione tenta di caricare una classe, ma non trova nessuna definizione di una classe con quel nome



# DESERIALIZZAZIONE

- il metodo `readObject()` legge la sequenza di bytes memorizzati in precedenza e crea un oggetto che è l'esatta replica di quello originale
  - `readObject` può leggere qualsiasi tipo di oggetto, è necessario effettuare un `cast` al tipo corretto dell'oggetto
- la JVM determina, mediante informazione memorizzata nell'oggetto serializzato, il tipo della classe dell'oggetto e tenta di caricare quella classe o una classe compatibile
- se non la trova viene sollevata una `ClassNotFoundException` ed il processo di deserializzazione viene abortito
- altrimenti, viene creato un nuovo oggetto sullo heap
  - lo stato di tutti gli oggetti serializzati viene ricostruito cercando i valori nello stream, senza invocare il costruttore (uso di Reflection)
  - si percorre l'albero delle superclassi fino alla prima superclasse non-serializzabile. Per quella classe viene invocato il costruttore



# COSA NON E' SERIALIZZABILE?

- oggetti contenenti riferimenti specifici alla JVM o al SO (JAVA native class)
  - Thread, OutputStream, Socket, File, non possono essere ricreati, perché contengono riferimenti specifici al particolare ambiente di esecuzione
- le variabili marcate come transient
  - ad esempio variabili che non devono essere scritte per questioni di privacy, es. numero carta di credito
- le variabili statiche: sono associate alla classe e non alla specifica istanza dell'oggetto che si sta serializzando
  - lette dalla classe in fase di deserializzazione
- tutti i componenti di un oggetto devono essere serializzabili: se ne esiste uno non serializzabile e non transient si solleva una notSerializableException
  - regola #2: per rendere un oggetto persistente occorre marcare tutti i campi che non sono serializzabili come transient

# ASSIGNMENT 7: GESTIONE CONTI CORRENTI

- considerare un file contenente i conti correnti di una banca, il file è di grosse dimensioni
- i conti correnti sono memorizzati nel file in formato JSON
- ogni conto corrente contiene il nome del correntista ed una lista di movimenti.
- per ogni movimento vengono registrati la data e la causale del movimento e l'insieme delle causali possibili è fissato: Bonifico, Accredito, Bollettino, F24, PagoBancomat.
- leggere poi il file e calcolare, per ogni possibile causale, quanti movimenti hanno quella causale.
- progettare un'applicazione che attiva un insieme di thread.
  - uno di essi legge dal file gli oggetti JSON “conto corrente” e li passa, uno per volta, ai thread presenti in un thread pool.
  - la lettura dal file deve essere fatta utilizzando l'API GSON per lo streaming
  - utilizzare il file di grande pubblicato sulla pagina del corso



# Reti e Laboratorio III

## Modulo Laboratorio III

**AA. 2022-2023**

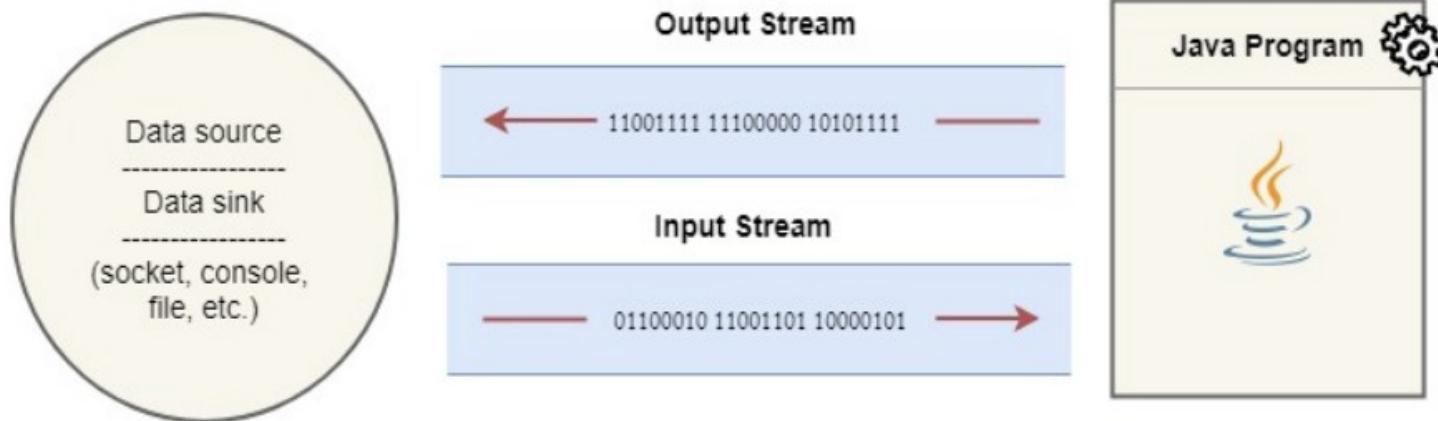
**docente: Laura Ricci**

**[laura.ricci@unipi.it](mailto:laura.ricci@unipi.it)**

**Lezione 8**  
**JAVA NIO:**  
**BUFFERS AND CHANNELS**

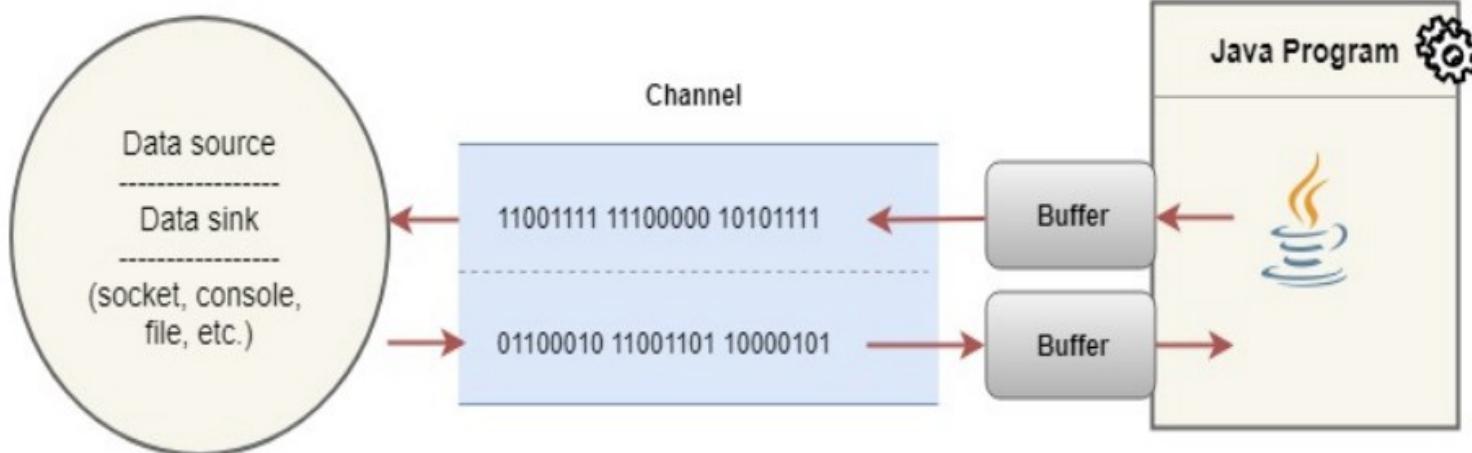
**03/11/2022**

# JAVA STREAM ORIENTED IO



- i dati sono scritti/letti su/da uno stream
- stream sono **unidirezionali** e **bloccanti**
- byte sono scritti/letti sullo stream un byte alla volta, ma è possibile una bufferizzazione dei dati scritti/letti su/dallo stream
  - `BufferedInputStream/OutputStream`: un buffer allocato nello heap della JVM da cui la JVM preleva i dati e poi li passa alla applicazione. Gestito dalla JVM
  - un array di byte: a carico del programmatore, allocato sullo heap

# JAVA NIO CHANNELS



- i dati sono trasferiti sul dispositivo mediante un **canale** e vengono scritti/letti in un **buffer**
  - il buffer è una interfaccia tra il programma e il canale
  - Il programma opera sul buffer, non sul canale
- i canali sono bidirezionali e possono essere non bloccanti

# CHANNEL E STREAM: CONFRONTO

- Channel sono bidirezionali
  - lo stesso Channel può leggere dal dispositivo e scrivere sul dispositivo
  - più vicino alla implementazione reale del sistema operativo.
- tutti i dati gestiti tramite oggetti di tipo Buffer: non si scrive/legge direttamente su un canale, ma si passa da un buffer
- possono essere bloccanti o meno:
  - non bloccanti: utili soprattutto per comunicazioni in cui i dati arrivano in modo incrementale
    - tipiche dei collegamenti di rete
  - minore importanza per letture da file, FileChannel sono bloccanti

- vantaggi
  - definizione di primitive “più vicine” al livello del sistema operativo, aumento di performance
  - in generale: migliori prestazioni, in molti casi, ma da valutare
- svantaggi
  - primitive a più basso livello di astrazione
    - perdita di semplicità ed eleganza rispetto allo stream-based I/O
    - maggior difficoltà nella messa a punto del programma
  - ma anche primitive espressive, ad esempio per il multiplexing dei canali
    - adatto per lo sviluppo di applicazioni che devono gestire un alto numero di connessioni di rete.
  - prestazioni dipendenti dalla piattaforma su cui si eseguono le applicazioni

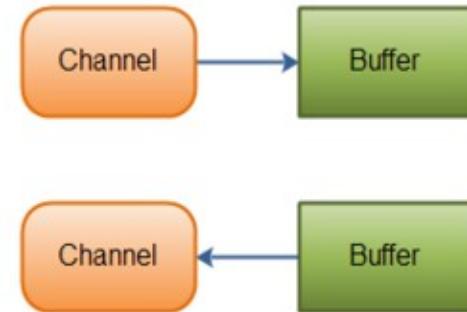
# JAVA NIO E NIO.2

- NIO (JAVA 1.4)
  - Buffers
  - Channels
  - Selectors
- NIO.2 (JAVA 1.7)
  - new File System API
  - asynchronous I/O
  - update
- NIO.2 implementato in alcuni package contenuti nel package NIO
- ci focalizzeremo soprattutto su NIO

# NIO: COSTRUTTI BASE

- **Canali e Buffers**

- IO standard è basato su stream di byte o di caratteri, con filtri
- NIO: tutti i dati da e verso dispositivi devono passare da un **canale**
  - simile ad uno stream in JAVA.IO
  - tutti i dati inviati a o letti da un canale devono essere memorizzati in **un buffer**



- **Selector** (introdotti in una prossima lezione)

- oggetto in grado di monitorare un insieme di canali
- intercetta **eventi** provenienti da diversi canali: dati arrivati, apertura di una connessione,...
- fornisce la possibilità di monitare più canali con un unico thread

# NIO BUFFERS E CHANNELS

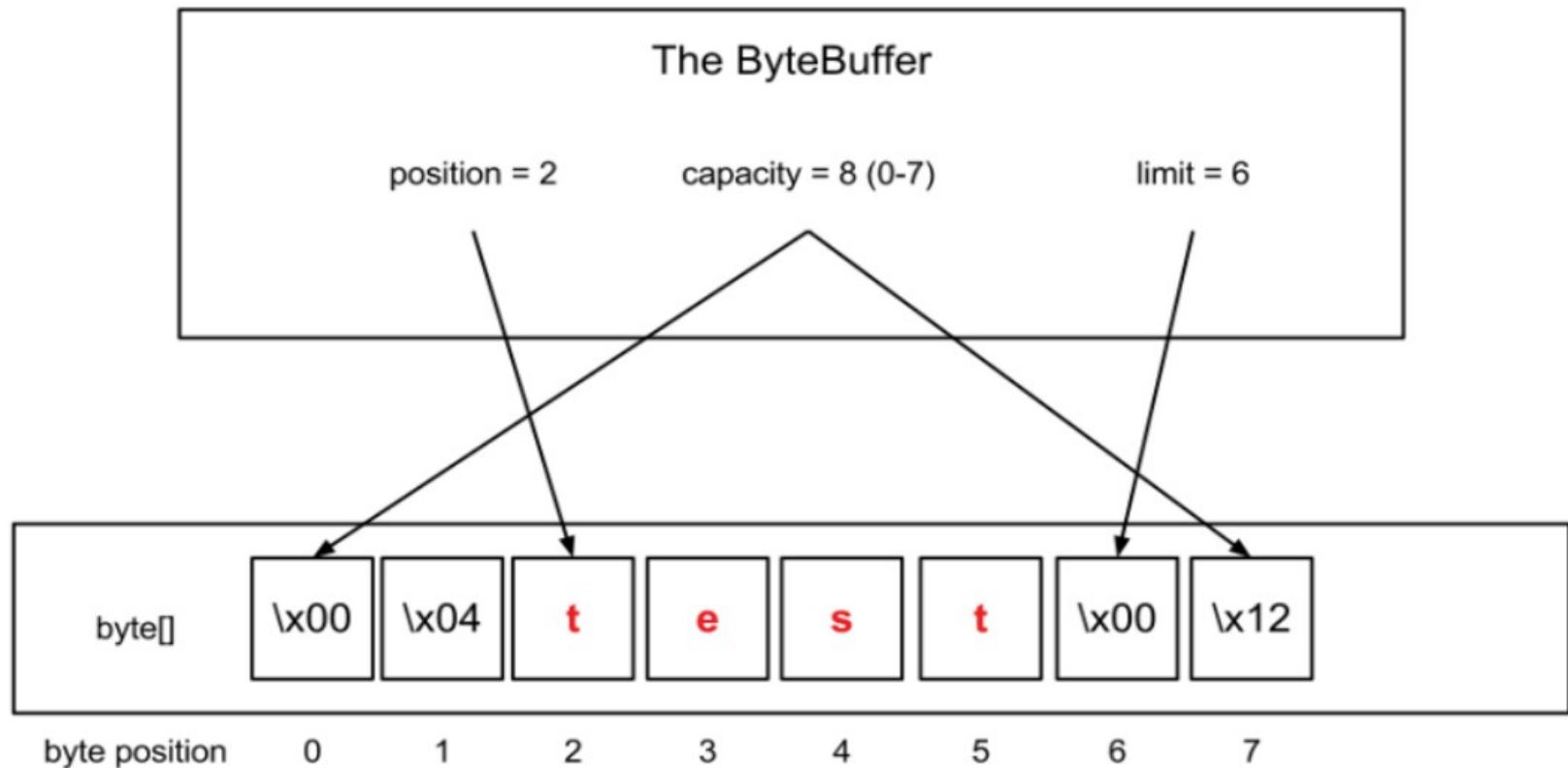
- Buffer

- implementati nella classe `java.nio.Buffer`
- contengono dati appena letti o che devono essere scritti su un Channel
  - interfaccia verso il sistema operativo
- array + puntatori per tenere traccia di read e write fatte dal programma e dal sistema operativo sul buffer
- non thread-safe

- Channel

- collega da/verso i dispositivi esterni, è **bidirezionale**
- a differenza degli stream, non si scrive/legge mai direttamente da un canale
- interazione con i canali
  - trasferimento dati dal canale nel buffer, il programma accede al buffer
  - il programma scrive nel buffer, il contenuto del buffer viene trasferito nel canale

# BUFFER: BYTEBUFFER

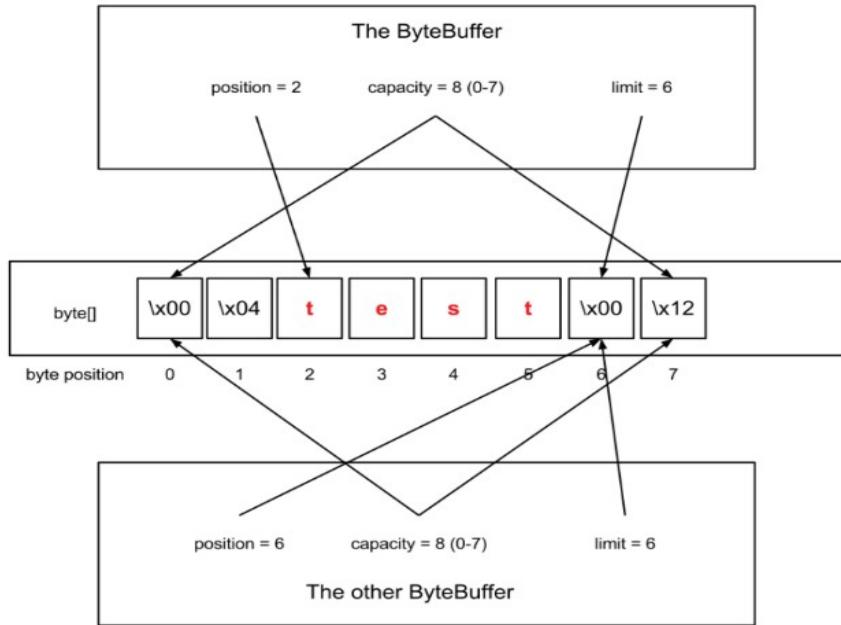
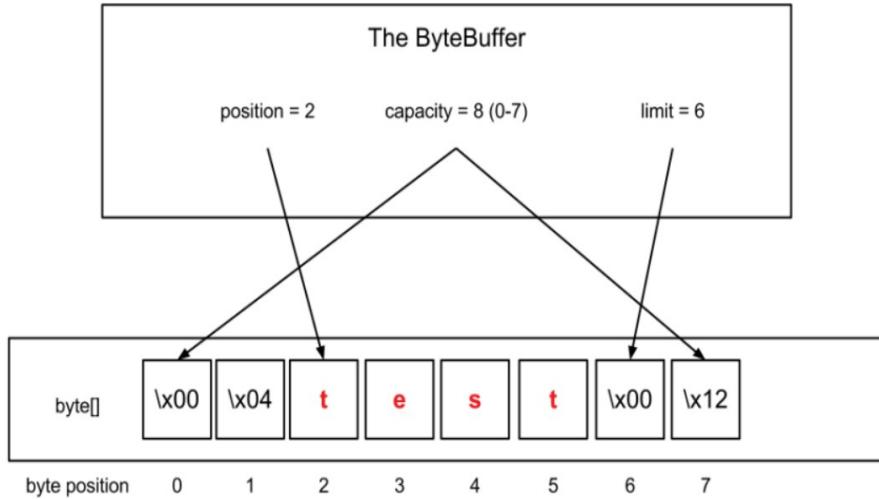


un oggetto di tipo Buffer è composto da

- uno spazio di memorizzazione: byte buffer
- un insieme di variabili di stato

un ByteBuffer “backed” da un byte array

# BUFFER: BYTEBUFFER



- supponiamo di eseguire il seguente codice

```
ByteBuffer other = bb.duplicate();
other.position(bb.position() + 4);
```

- otteniamo due diversi ByteBuffer che si riferiscono al solito bytearray, ma il loro contenuto è diverso

# BUFFER: LE VARIABILI DI STATO

- Capacity
  - massimo numero di elementi del Buffer
  - definita al momento della creazione del Buffer, non può essere modificata
  - `java.nio.BufferOverflowException`, se si tenta di leggere/scrivere in/da una posizione > Capacity
- Limit
  - indica il limite della porzione del Buffer che può essere letta/scritta
    - per le scritture `limit = capacity`
    - per le letture delimita la porzione di Buffer che contiene dati significativi
  - aggiornato implicitamente dalla operazioni sul buffer effettuate dal programma o dal canale

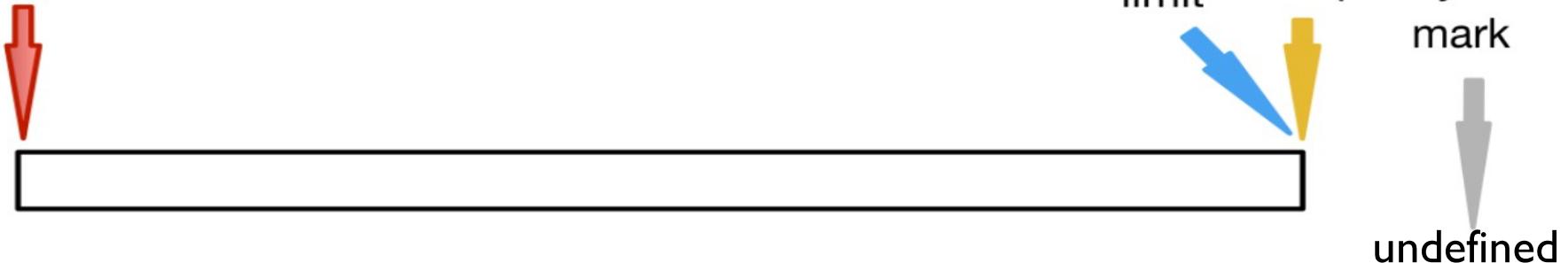
# LE VARIABILI DI STATO

- Position
  - come un file pointer per un file ad accesso sequenziale
  - posizione in cui bisogna scrivere o da cui bisogna leggere
  - aggiornata implicitamente dalla operazioni di lettura/scrittura sul buffer effettuate dal programma o dal canale
- Mark
  - memorizza il puntatore alla posizione corrente
  - il puntatore può quindi essere resettato a quella posizione per rivisitarla
  - inizialmente è undefined
  - se si resetta un mark undefined: `java.nio.InvalidMarkException`
- valgono sempre le seguenti relazioni

$$0 \leq \text{mark} \leq \text{position} \leq \text{limit} \leq \text{capacity}$$

# SCRIVERE DATI NEL BUFFER

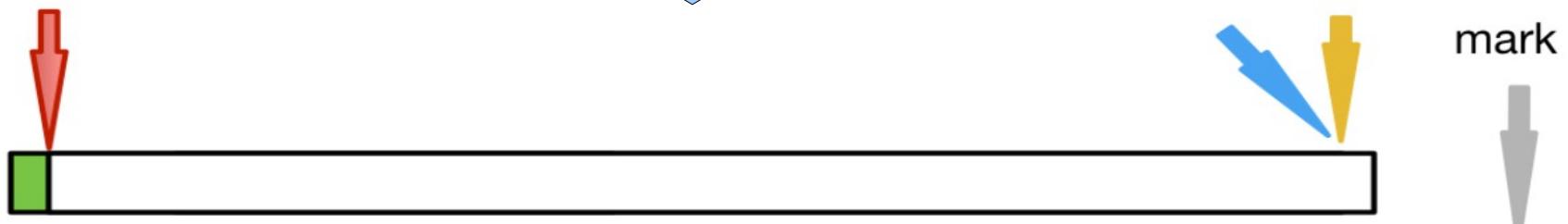
Position



stato iniziale del Buffer: Limit=Capacity-1, Position=0,  
Mark=undefined

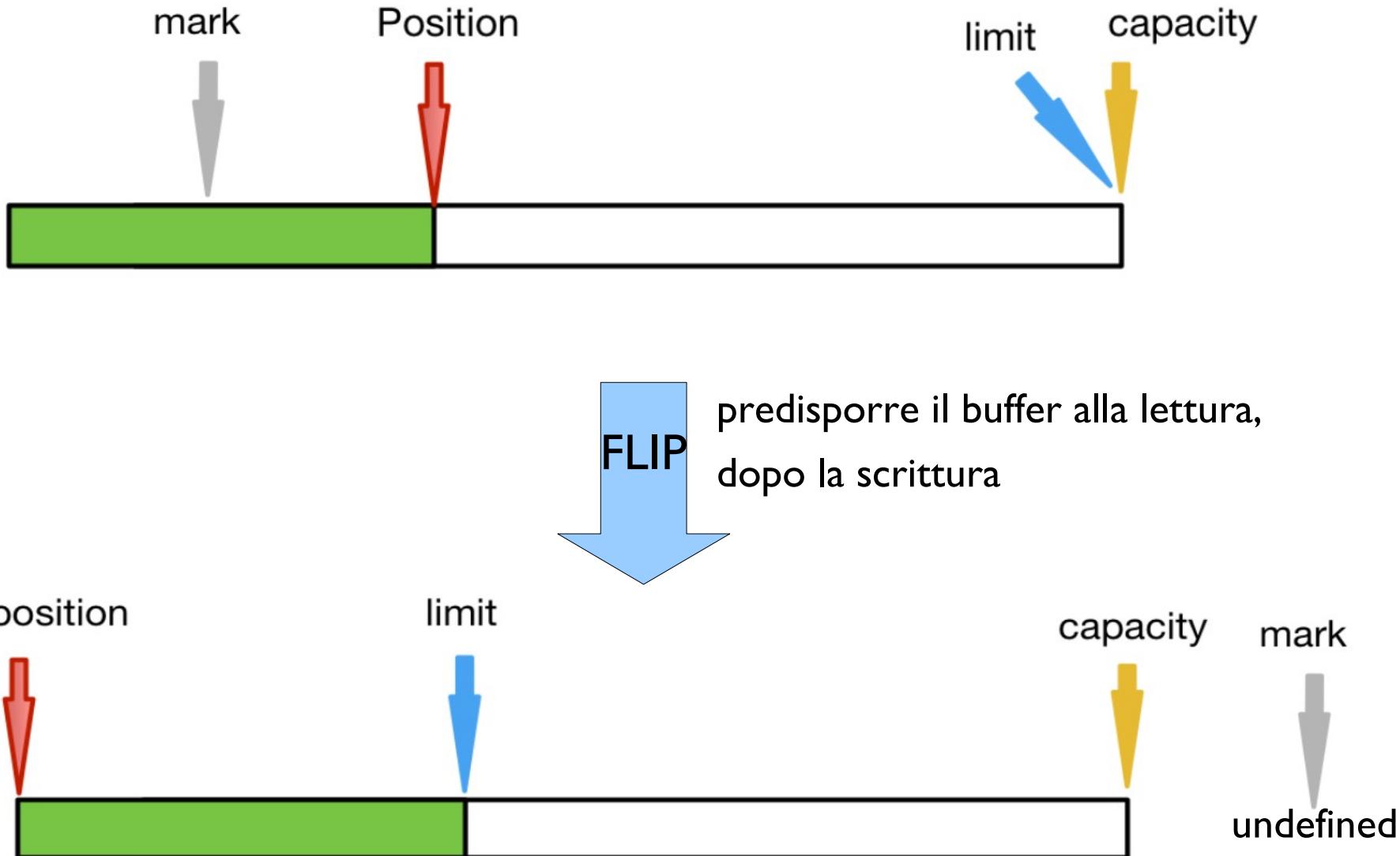
Position++

SCRITTURA

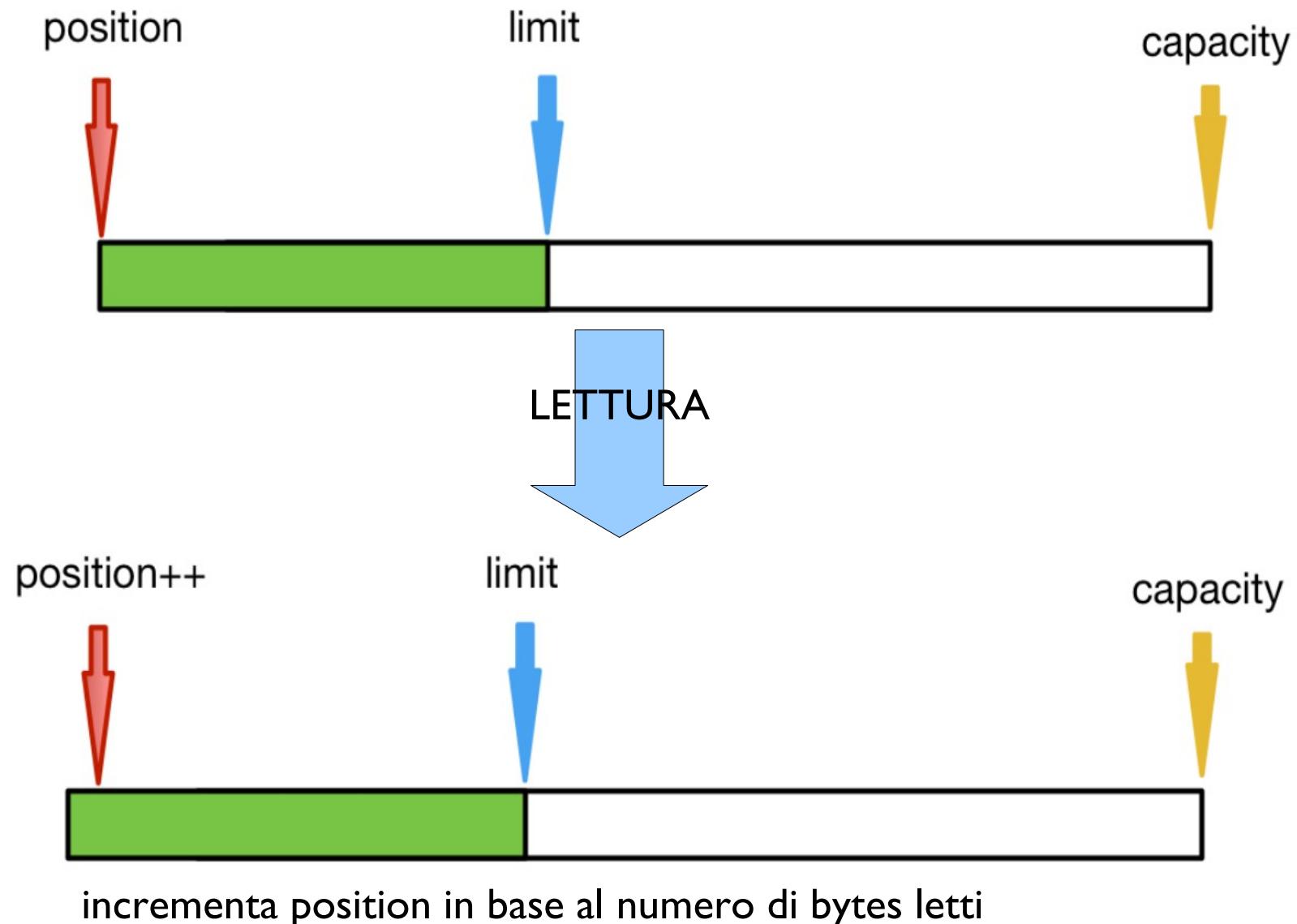


incrementa Position in base al numero di bytes scritti

# OPERAZIONI SUL BUFFER: FLIPPING

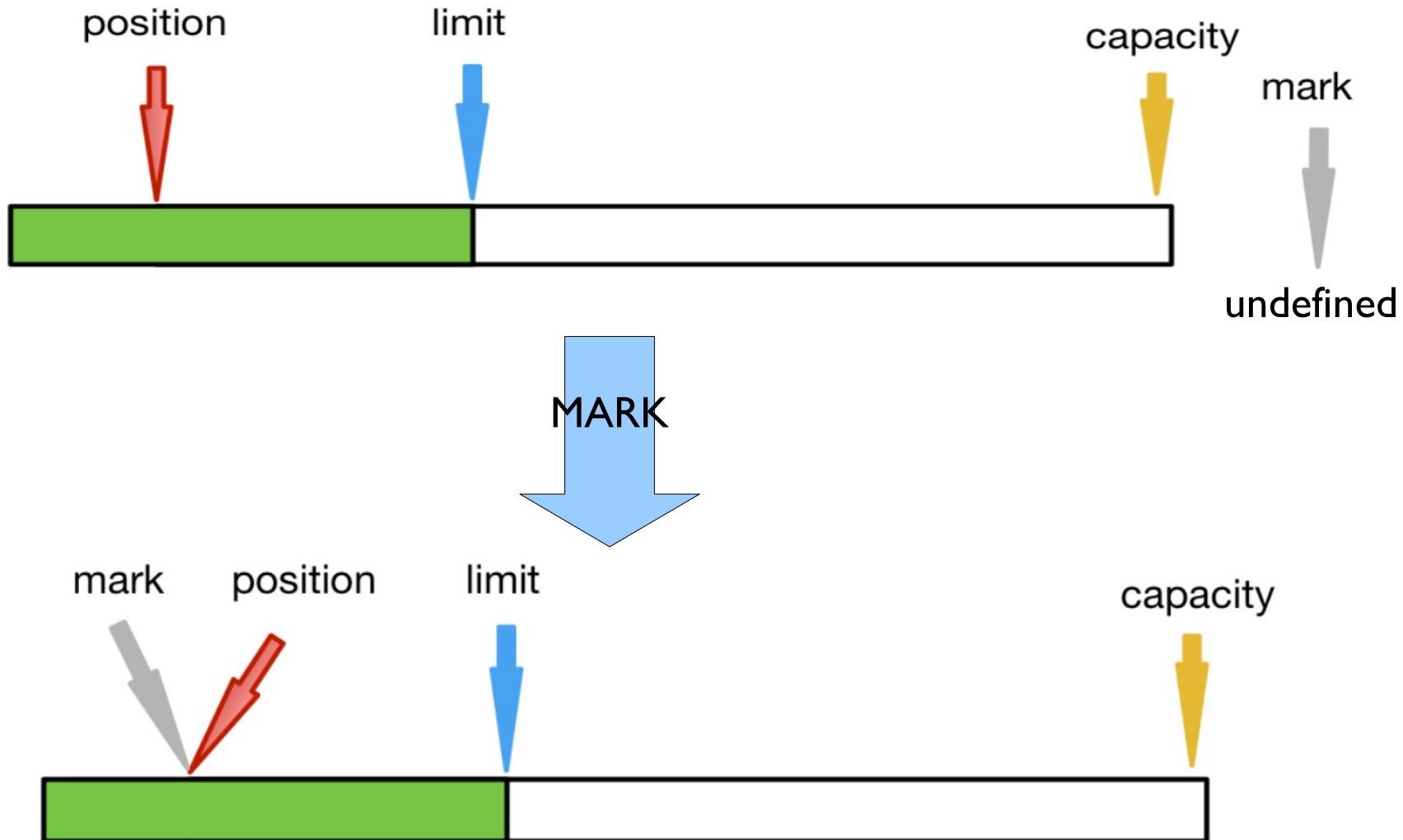


# LETTURA DAL BUFFER



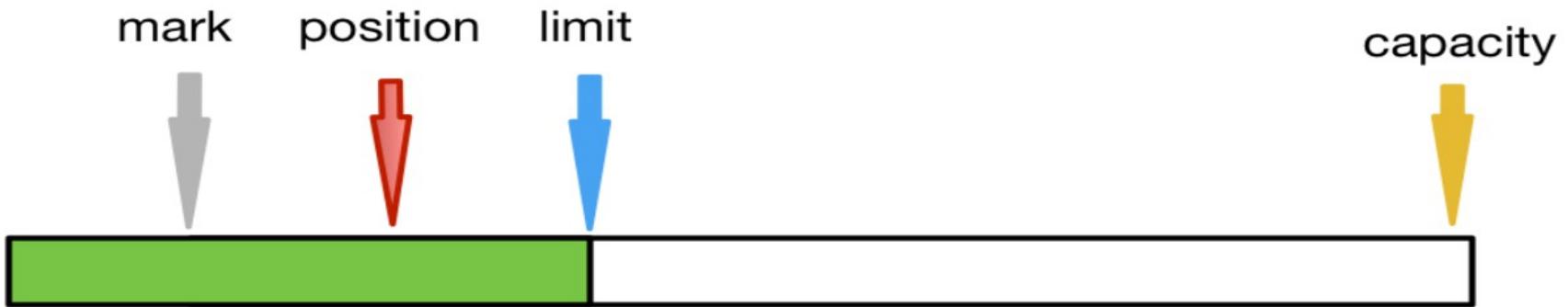
incrementa position in base al numero di bytes letti

# OPERAZIONI SUL BUFFER: MARK

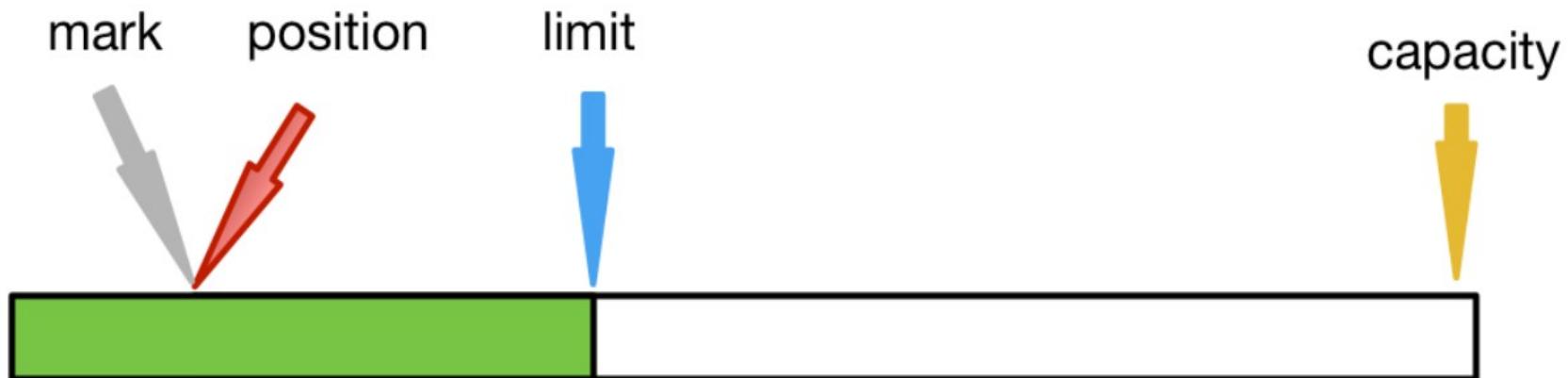


ricorda la Position corrente, per poi eventualmente riportare il puntatore a questa posizione

# OPERAZIONI SUL BUFFER: RESET

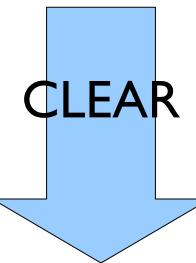
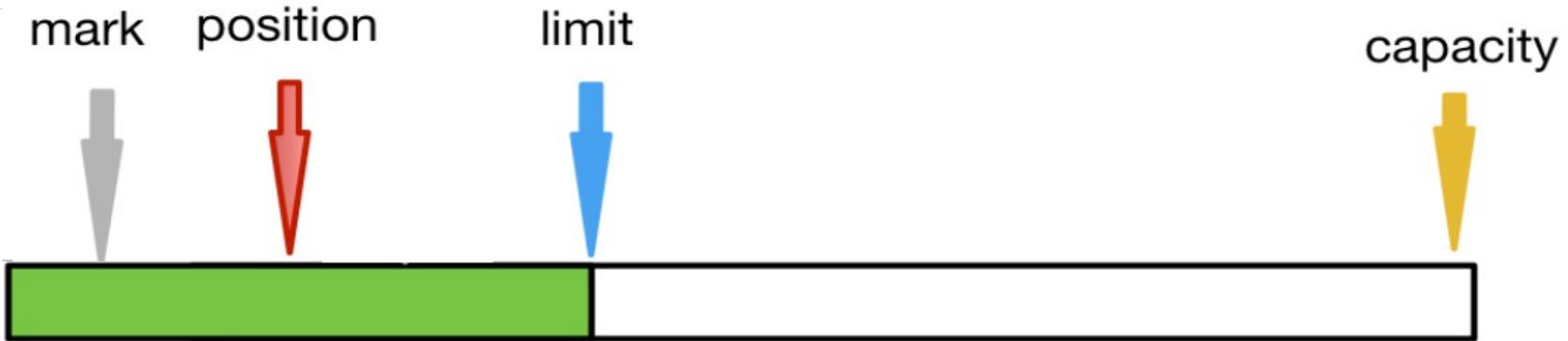


RESET

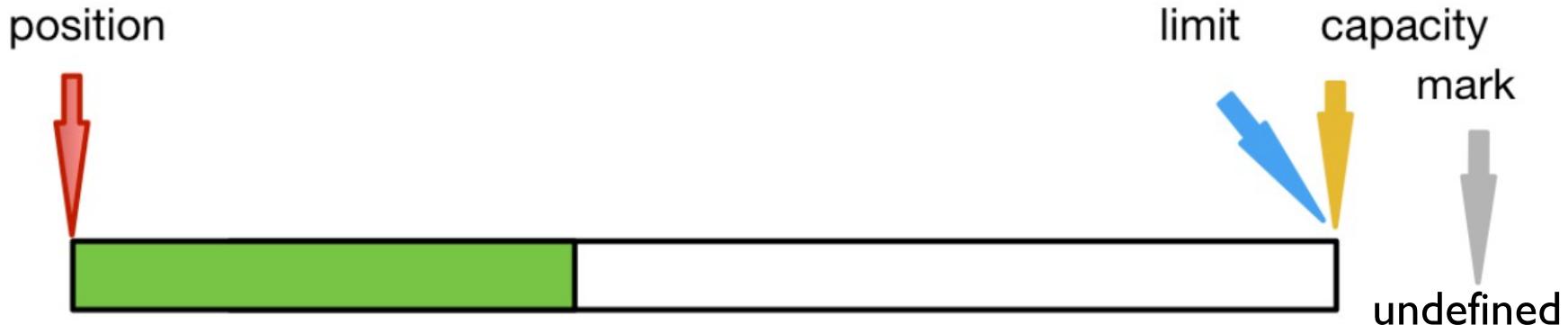


resetta position alla posizione precedentemente memorizzata in mark

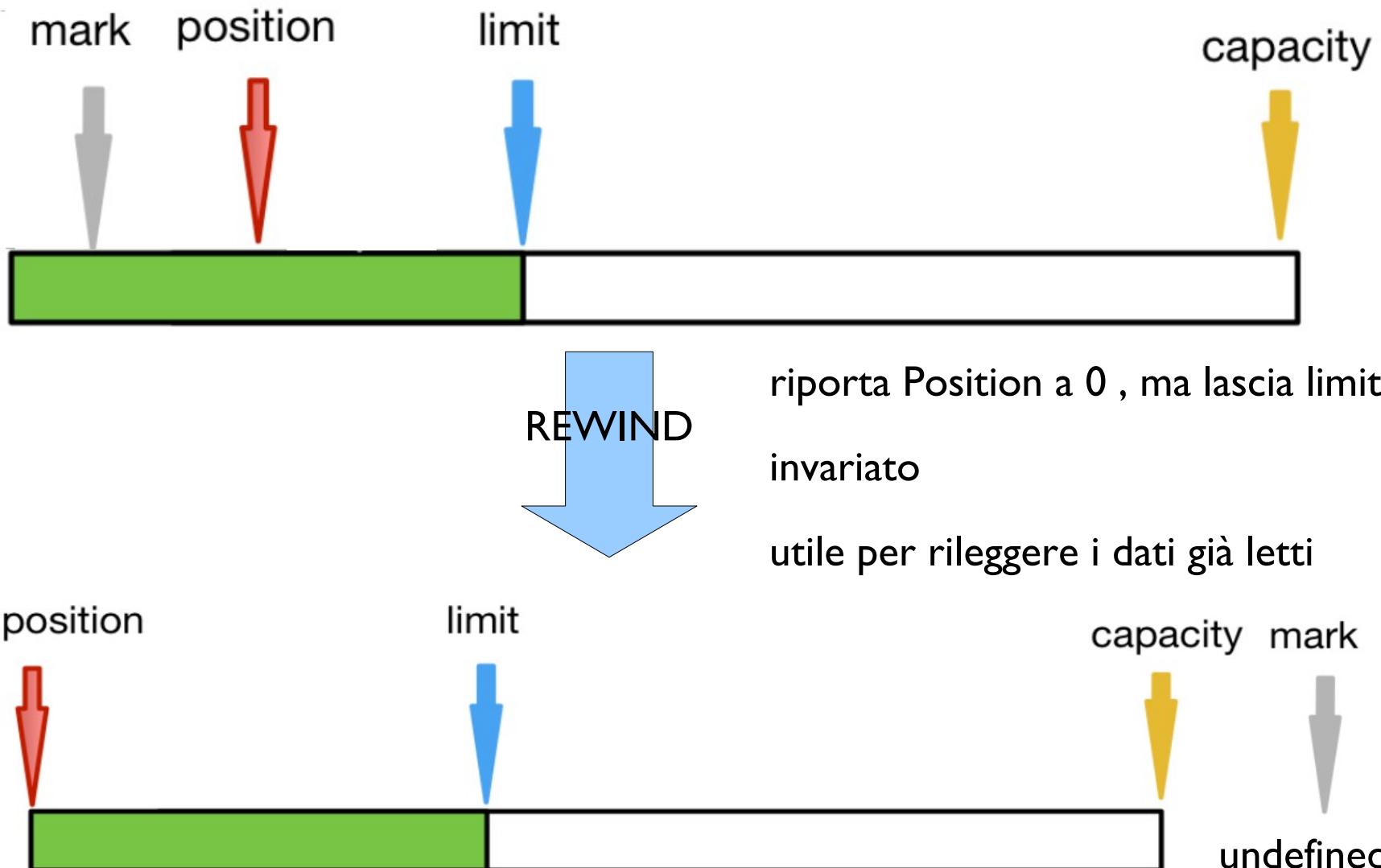
## §OPERAZIONI SUL BUFFER: CLEARING



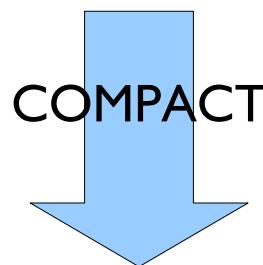
per ritornare in modalità scrittura  
non elimina i dati dal buffer resetta  
i contatori



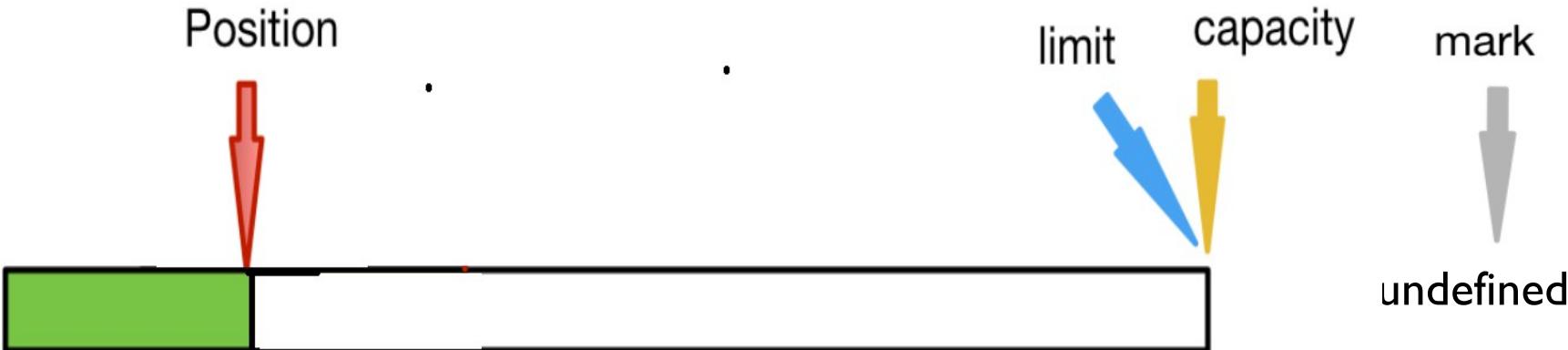
# OPERAZIONI SUL BUFFER: REWINDING



# OPERAZIONE SUL BUFFER: COMPACTING



utile se il contenuto del buffer non è stato completamente letto e si inizia una nuova scrittura  
i bytes non ancora letti vengono copiati all'inizio del buffer



# ALTRI METODI UTILI



- **Remaining()**
  - restituisce il numero di elementi nel buffer compresi tra position e limit
- **HasRemaining()**
  - restituisce true se `remaining()` è maggiore di 0

# ANALIZZARE LE VARIABILI DI STATO

```
import java.nio.*;
public class Buffers {
    public static void main (String args[])
    {ByteBuffer byteBuffer1 = ByteBuffer.allocate(10);
        System.out.println(byteBuffer1);
        // java.nio.HeapByteBuffer[pos=0 lim=10 cap=10]
        byteBuffer1.putChar('a');
        System.out.println(byteBuffer1);
        // java.nio.HeapByteBuffer[pos=2 lim=10 cap=10]
        byteBuffer1.putInt(1);
        System.out.println(byteBuffer1);
        // java.nio.HeapByteBuffer[pos=6 lim=10 cap=10]
        byteBuffer1.flip();
        System.out.println(byteBuffer1);
        // java.nio.HeapByteBuffer[pos=0 lim=6 cap=10]
```



# ANALIZZARE LE VARIABILI DI STATO

```
System.out.println(byteBuffer1.getChar());  
System.out.println(byteBuffer1);  
// a  
// java.nio.HeapByteBuffer[pos=2 lim=6 cap=10]  
byteBuffer1.compact();  
System.out.println(byteBuffer1);  
// java.nio.HeapByteBuffer[pos=4 lim=10 cap=10]  
byteBuffer1.putInt(2);  
System.out.println(byteBuffer1);  
// java.nio.HeapByteBuffer[pos=8 lim=10 cap=10]  
byteBuffer1.flip();  
// java.nio.HeapByteBuffer[pos=0 lim=8 cap=10]  
System.out.println(byteBuffer1.getInt());  
System.out.println(byteBuffer1.getInt()); System.out.println(byteBuffer1);  
// 1  
// 2  
// java.nio.HeapByteBuffer[pos=8 lim=8 cap=10]
```

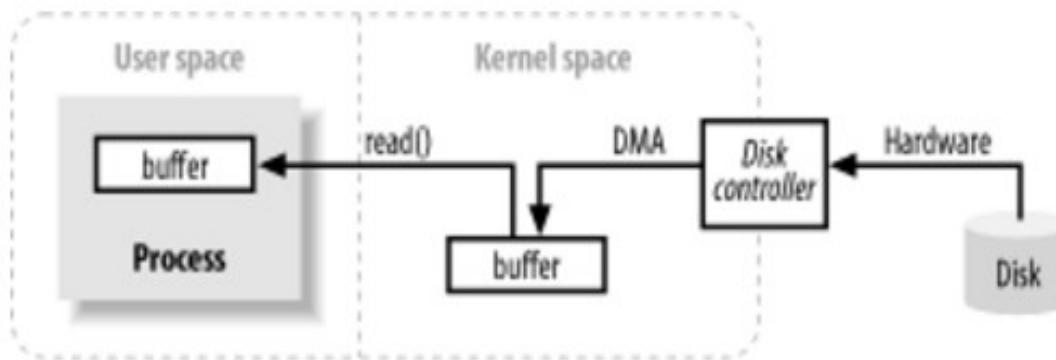


# ANALIZZARE LE VARIABILI DI STATO

```
byteBuffer1.rewind();
// rewind prepara a rileggere i dati che sono nel buffer, ovvero resetta
// position a 0 e non modifica limit
// java.nio.HeapByteBuffer[pos=0 lim=8 cap=10]
System.out.println(byteBuffer1.getInt());
// 1
byteBuffer1.mark();
System.out.println(byteBuffer1.getInt());
// 2
System.out.println(byteBuffer1);
//position:8;limit:8;capacity:10
byteBuffer1.reset();
System.out.println(byteBuffer1);
//position:4;limit:8;capacity:10
byteBuffer1.clear();
System.out.println(byteBuffer1);
//position:0;limit:10;capacity:10]]>
```

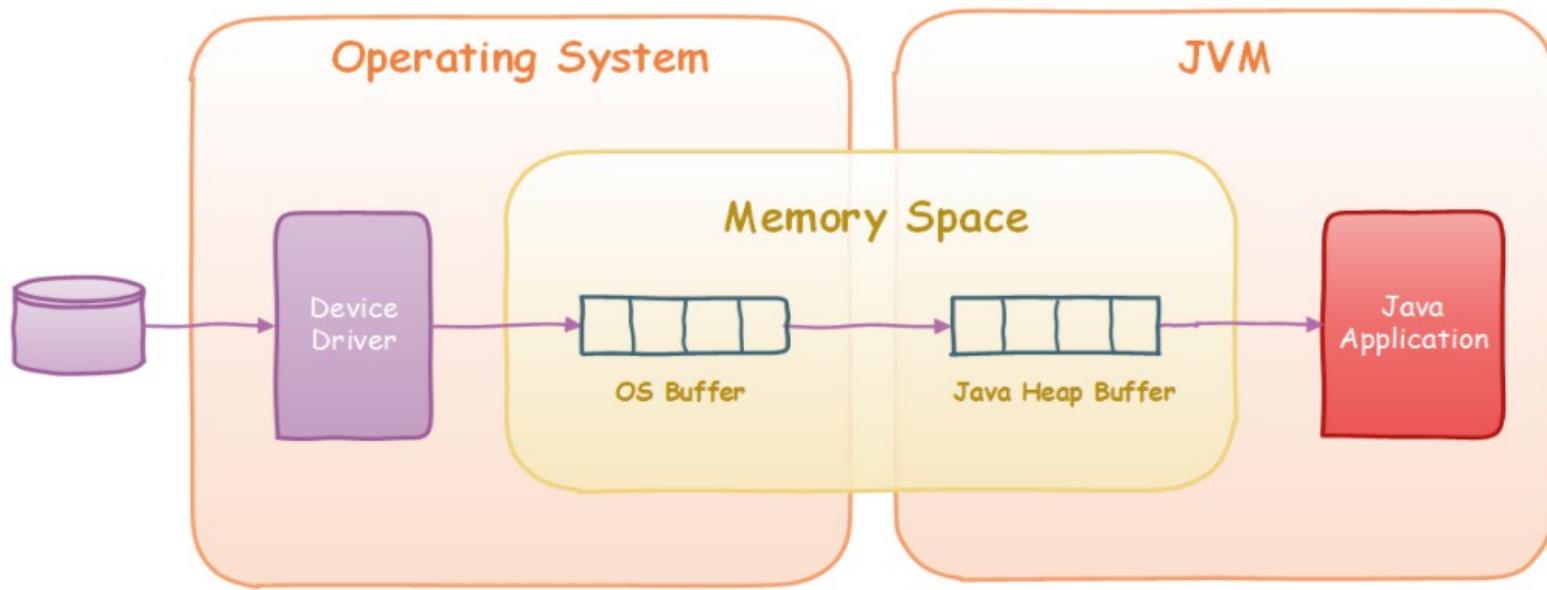


# INTERAZIONE JVM/SISTEMA OPERATIVO



- la JVM esegue una `read()` da stream o canale e provoca una system call (native code)
- il kernel invia un comando al disk controller
- il disk controller, via DMA (senza controllo della CPU) scrive direttamente un blocco di dati nel kernel space
- i dati sono copiati dal kernel space nello user space (all'interno della JVM).
- si può ottimizzare questo processo?
- la gestione ottimizzata di questi buffer può comportare un notevole miglioramento della performance dei programmi!

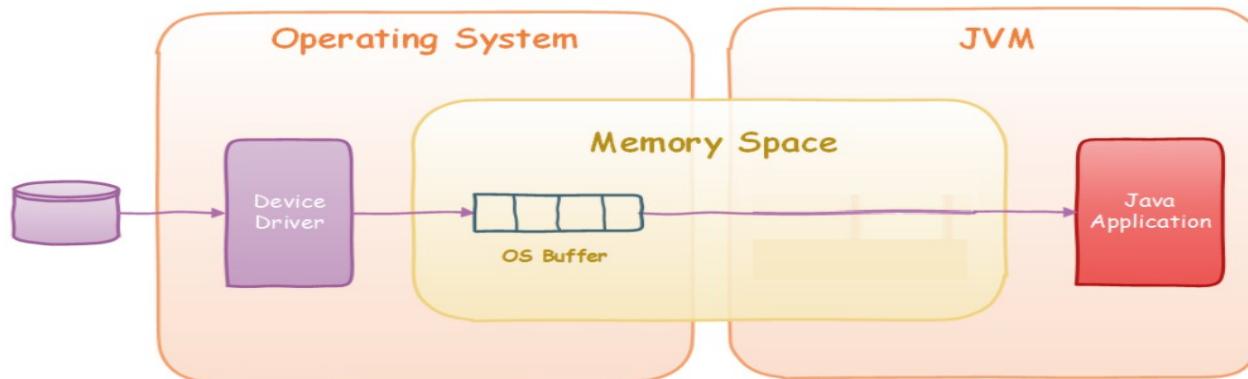
# NON DIRECT BUFFERS: CREAZIONE



```
ByteBuffer buf = ByteBuffer.allocate(10);
```

- crea sullo heap un oggetto Buffer
- doppia copia dei dati
  - nel buffer del kernel
  - nel buffer sullo heap della JVM

# DIRECT BUFFER: CREAZIONE



```
ByteBuffer buffer = ByteBuffer.allocateDirect( 1024 );
```

- trasferire dati tra il programma ed il sistema operativo, mediante accesso diretto alla kernel memory da parte della JVM
- evita copia dei dati da/in un buffer intermedio prima/dopo l'invocazione del sistema operativo
- vantaggi: migliore performance
- svantaggi
  - maggiore costo di allocazione/deallocazione
  - il buffer non è allocato sullo heap. Garbage collector non può recuperare memoria

# CANALI: SCRITTURA

- se il canale è utilizzato solo in output, possiamo crearlo partendo da un FileOutputStream, usando classi “ponte” tra stream e channels

```
FileOutputStream fout = new FileOutputStream( "example.txt" );  
FileChannel fc = fout.getChannel();
```

- creazione del Buffer interfaccia sul canale

```
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
```

- scrittura del messaggio nel Buffer

```
for (int i=0; i<message.length; ++i) {  
    buffer.put( message[i] ); }
```

- scrittura sul canale

```
buffer.flip();  
fc.write( buffer );
```

- notare che occorre predisporre il Buffer in lettura, dopo che i dati sono stati trasferiti



# CANALI: LETTURA

- se il canale è utilizzato solo in input, possiamo crearlo partendo da un `FileInputStream`, usando classi “ponte” tra stream e channels

```
FileInputStream fin = new FileInputStream( "example.txt" );  
FileChannel fc = fin.getChannel();
```

- creazione di un `ByteBuffer`

```
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
```

- lettura dal canale al Buffer

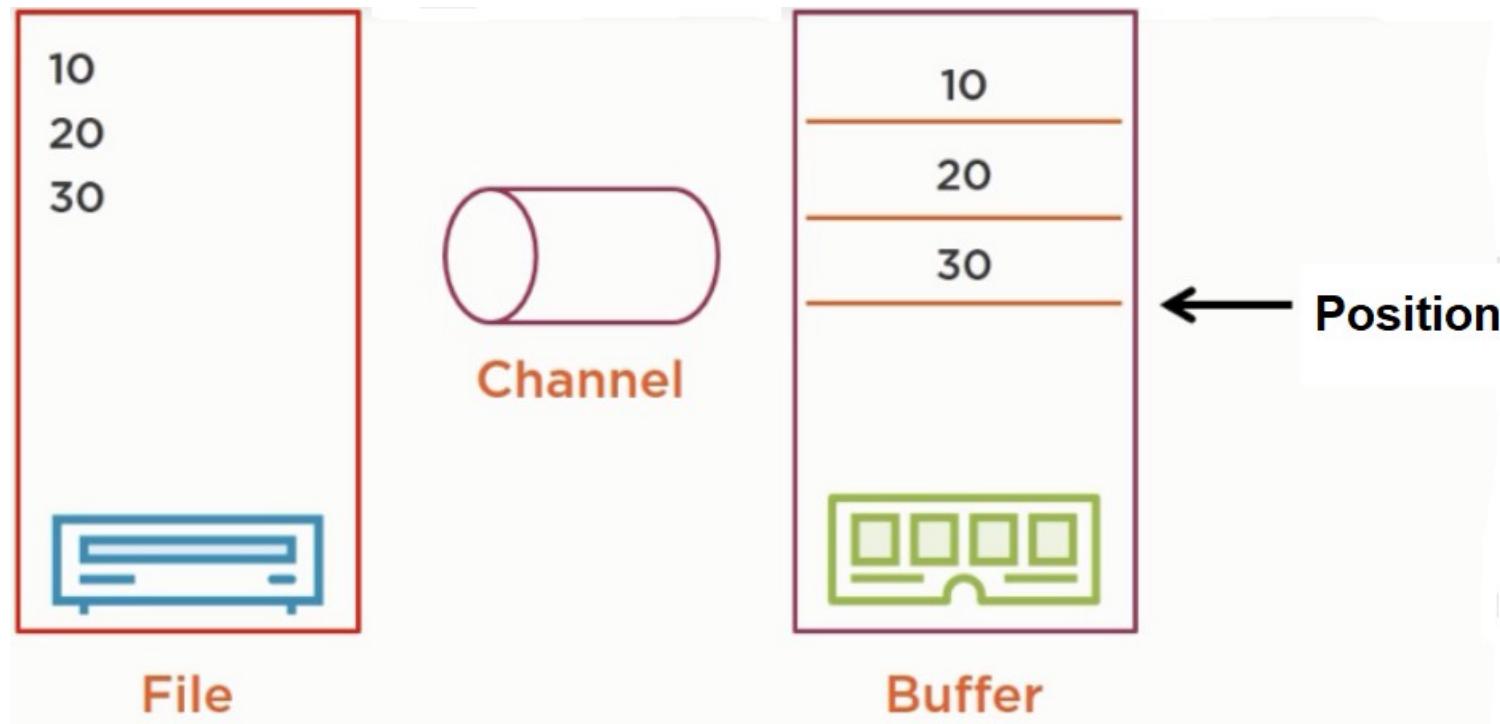
```
fc.read( buffer );
```

- non si specifica quanti byte il sistema operativo deve leggere nel Buffer

- quando la read termina ci saranno alcuni byte nel canale, ma quanti?

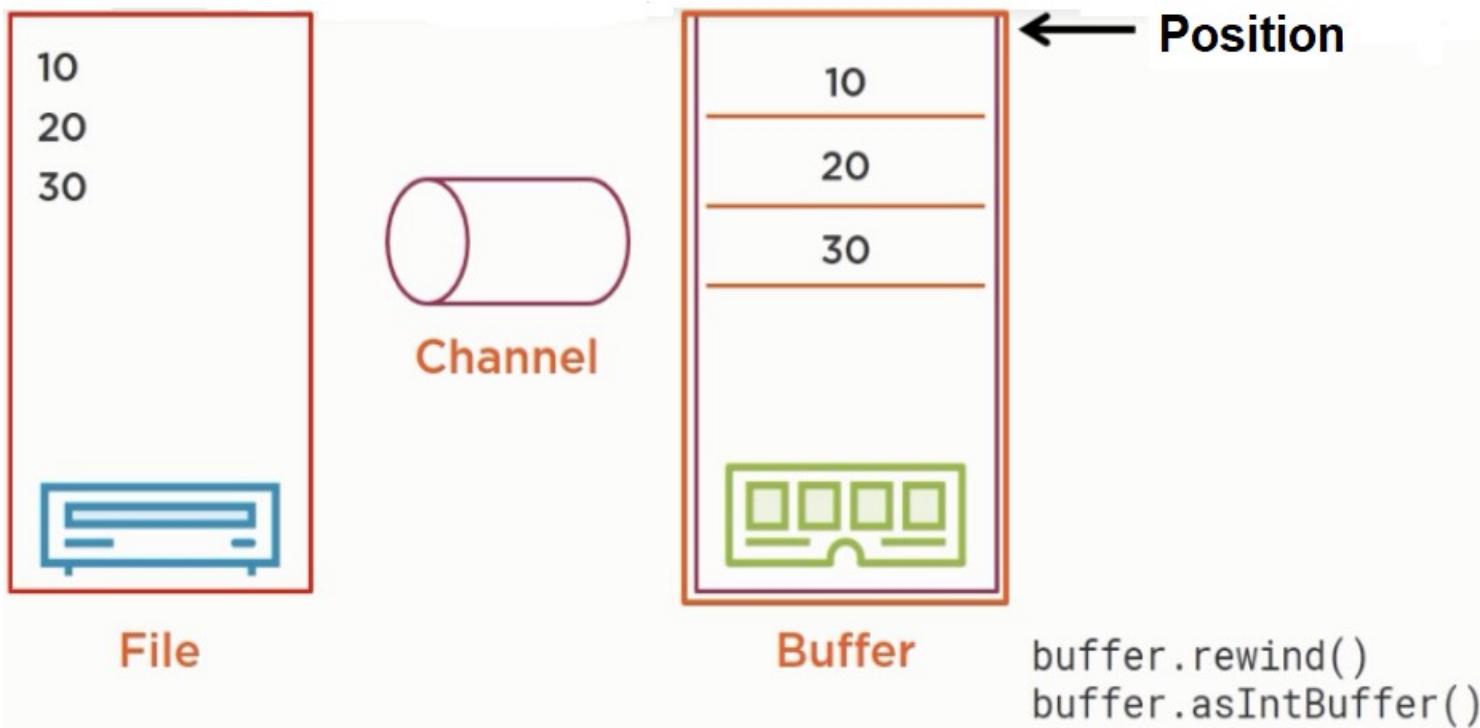
- necessarie delle variabili interne all'oggetto Buffer che mantengano lo stato del Buffer, ad esempio: quale parte del buffer è significativa?

# LEGGERE DAL CANALE



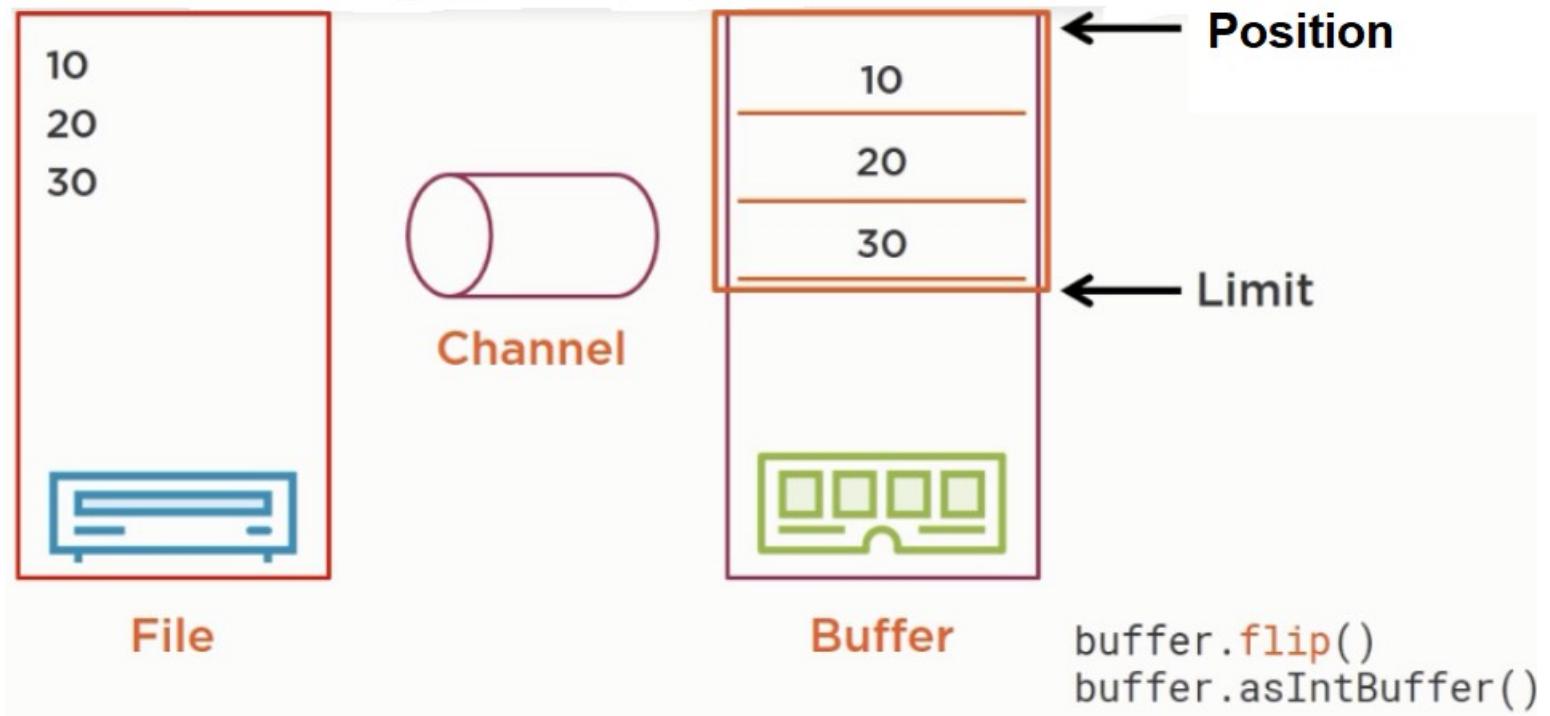
- lettura di 3 interi dal File al Buffer tramite il Channel
- dopo la lettura dal file, Position indica l'ultima posizione letta dal file
- ora il programma deve leggere i dati dal Buffer

# LEGGERE DAL CANALE



- una possibilità (errata): usare la `rewind()` che riporta il cursore all'inizio del Buffer
- quanti interi posso leggere nel Buffer?
- occorre tenere traccia di dove si trovava Position prima della `rewind()`
- Nota: `asIntBuffer()` interpreta i byte del Buffer come interi

# L'OPERAZIONE GIUSTA E' LA FLIP!



- la `flip()` setta Limit alla Position corrente
  - viene memorizzata quale è la parte significativa del Buffer
- quindi si comporta come la `rewind()`, riporta Position a 0

- connessi a descrittori di file/socket gestiti dal Sistema Operativo
- l'API per i Channel utilizza molte interfacce JAVA
  - le implementazioni utilizzano principalmente codice nativo
- una interfaccia, Channel che è radice di una gerarchia di interfacce

FileChannel: legge/scrive dati su un File

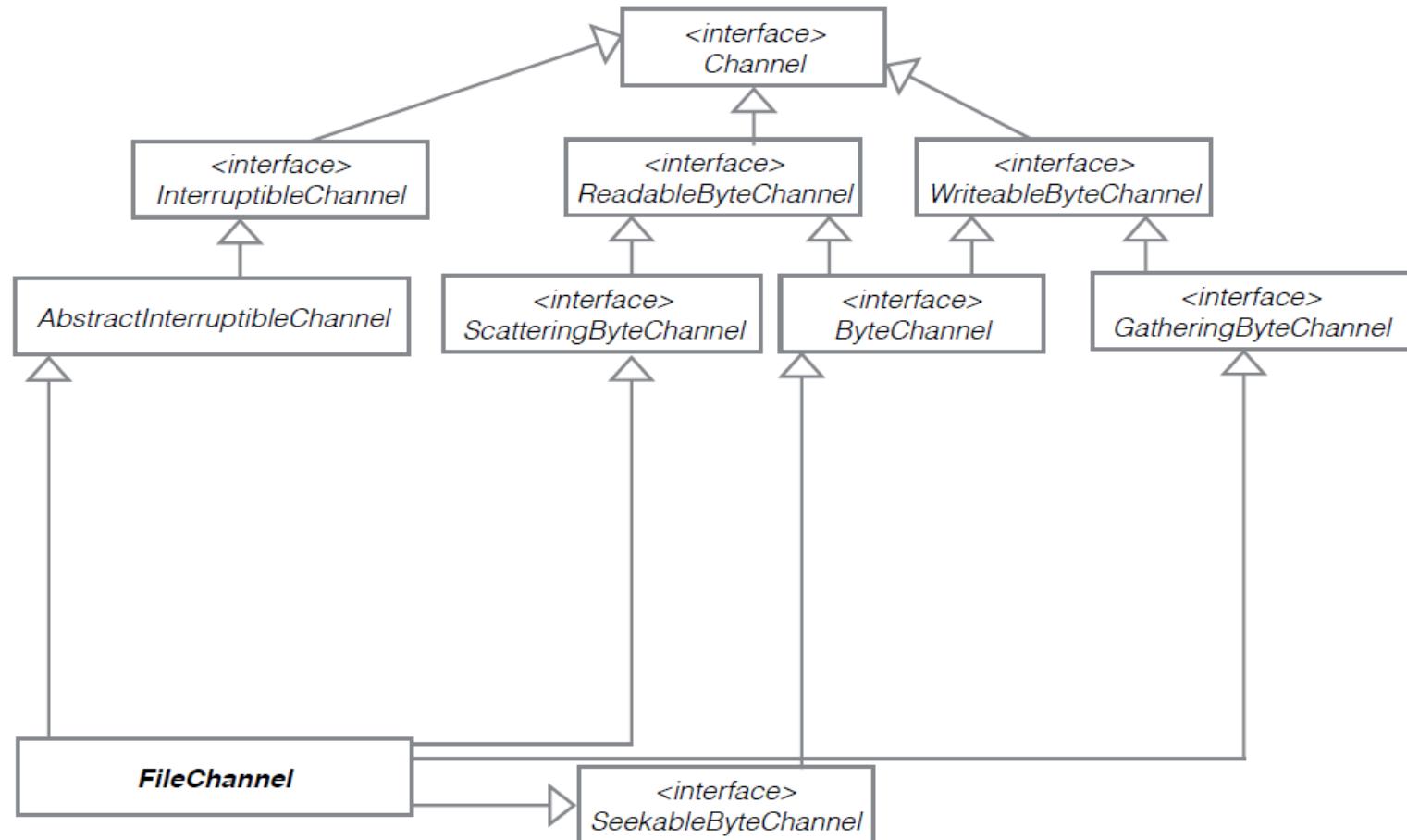
DatagramChannel: legge/scrive dati sulla rete via UDP

SocketChannel: legge/scrive dati sulla rete via TCP

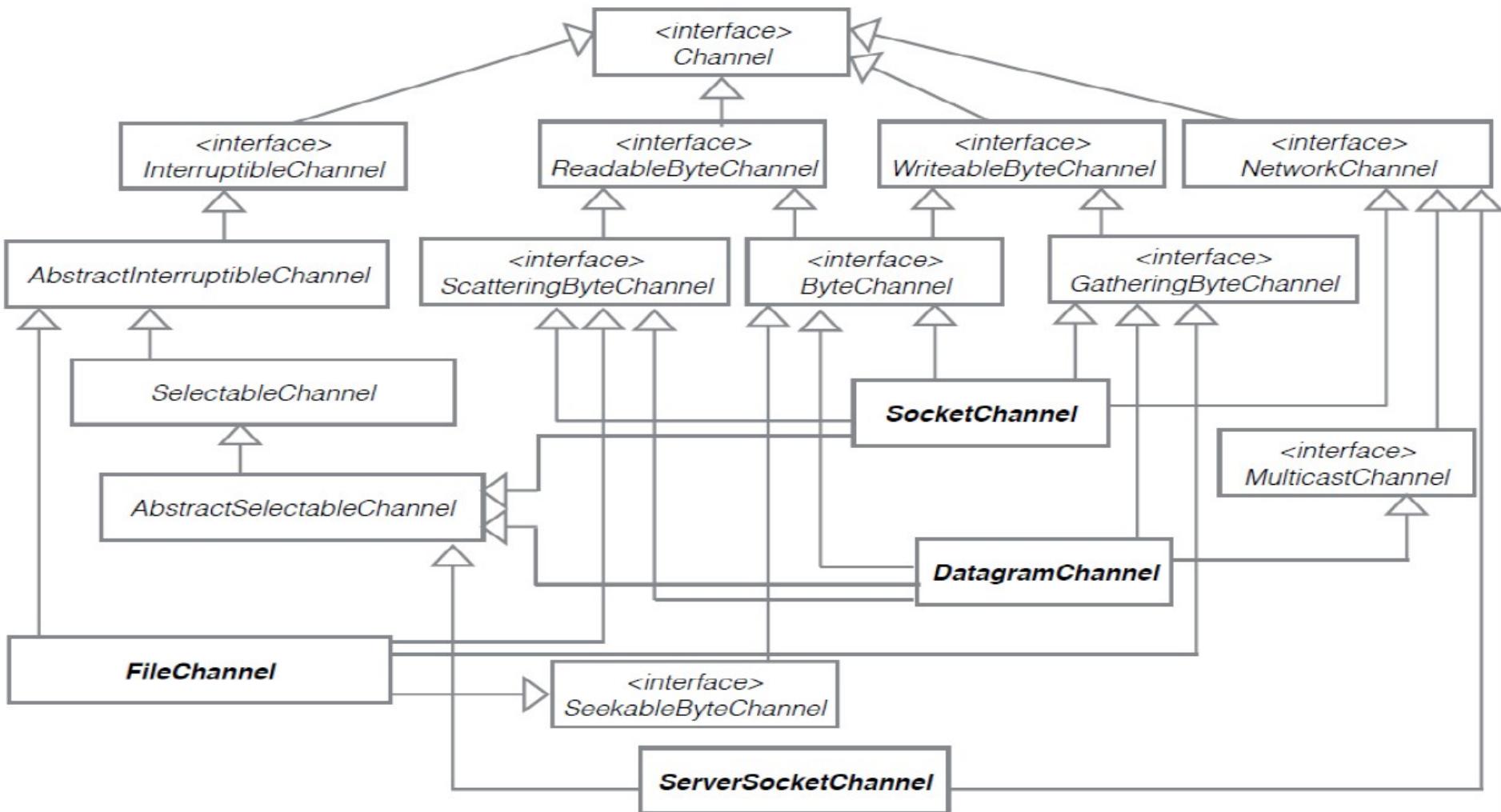
ServerSocketChannel: attende richieste di connessioni TCP e crea un  
SocketChannel per ogni connessione creata.

- gli ultimi tre possono essere **non bloccanti** (vedi prossime lezioni)
- è possibile un “trasferimento diretto” da Channel a Channel se almeno uno dei due è un Channel

# FILECHANNEL: GERARCHIA DI INTERFACCE



# CHANNEL: CLASSI ED INTERFACCE



# FILE CHANNELS

- oggetti di tipo FileChannel possono essere creati direttamente utilizzando FileChannel.open (di JAVA.NIO.2), dichiarando il tipo di accesso al channel (READ/WRITE o entrambi)

```
File fileEx = new File(inFileExemple);  
FileChannel in=FileChannel.open(fileEx.toPath(),StandardOpenOption.READ)
```

- FileChannel API è a basso livello: solo metodi per leggere e scrivere bytes
  - lettura e scrittura richiedono come parametro un ByteBuffer
- bloccanti e thread safe
  - più thread possono lavorare in modo consistente sullo stesso channel
  - alcune operazioni possono essere eseguite in parallelo (esempio: read), altre vengono automaticamente serializzate
    - ad esempio le operazioni che cambiano la dimensione del file o il puntatore sul file vengono eseguite in mutua esclusione
  - operazioni non bloccanti possibili su SocketChannel

# COPIARE FILE CON NIO

```
import java.nio.ByteBuffer;
import java.nio.channels.ReadableByteChannel;
import java.nio.channels.WritableByteChannel;
import java.nio.channels.Channels;
import java.io.*;

public class ChannelCopy
{ public static void main (String [] argv) throws IOException
{ ReadableByteChannel source =
    Channels.newChannel(new FileInputStream("in.txt"));
WritableByteChannel dest =
    Channels.newChannel (new FileOutputStream("out.txt"));
channelCopy1 (source, dest);
source.close();
dest.close();
}
```



# COPIARE FILE CON NIO

```
private static void channelCopy1 (ReadableByteChannel src,
                                WritableByteChannel dest) throws IOException
{ ByteBuffer buffer = ByteBuffer.allocateDirect (16 * 1024);
  while (src.read (buffer) != -1) {
    // prepararsi a leggere i byte che sono stati inseriti nel buffer
    buffer.flip();
    // scrittura nel file destinazione; può essere bloccante
    dest.write (buffer);
    // non è detto che tutti i byte siano trasferiti, dipende da quanti
    // bytes la write ha scaricato sul file di output
    // compatta i bytes rimanenti all'inizio del buffer
    // se il buffer è stato completamente scaricato, si comporta come clear()
    buffer.compact(); }

  // quando si raggiunge l'EOF, è possibile che alcuni byte debbano essere ancora
  // scritti nel file di output
  buffer.flip();
  while (buffer.hasRemaining()) { dest.write (buffer); }}
```



# COPIARE FILE CON NIO

`read()`

- può non riempire l'intero buffer, limit indica la porzione di buffer riempita dai dati letti dal canale
- restituisce -1 quando i dati sono finiti

`flip()`

- converte il buffer da modalità scrittura a modalità lettura

`write()`

- preleva alcun dati dal buffer e li scarica sul canale. Non necessariamente scrive tutti i dati presenti nel Buffer sul canale

`hasRemaining()`

- verifica se esistono elementi nel buffer nelle posizioni comprese tra position e limit

# COPIARE FILE CON NIO

```
private static void channelCopy2 (ReadableByteChannel src,
                                 WritableByteChannel dest)      throws IOException
{
    ByteBuffer buffer = ByteBuffer.allocateDirect (16 * 1024);
    while (src.read (buffer) != -1) {
        // prepararsi a leggere i byte inseriti nel buffer dalla lettura
        // del file
        buffer.flip();
        // riflettere sul perchè del while
        // una singola lettura potrebbe non aver scaricato tutti i dati
        while (buffer.hasRemaining()) {
            dest.write (buffer);
        }
        // a questo punto tutti i dati sono stati letti e scaricati sul file
        // preparare il buffer all'inserimento dei dati provenienti
        // dal file
        buffer.clear();
    }
}
```



# TRASFERIMENTO DIRETTO TRA CANALI

- due metodi
  - `FileChannel.transferTo()`
  - `FileChannel.transferFrom()`
- ad esempio

```
RandomAccessFile fromFile = new RandomAccessFile("fromFile.txt", "rw");
FileChannel      fromChannel = fromFile.getChannel();

RandomAccessFile toFile = new RandomAccessFile("toFile.txt", "rw");
FileChannel      toChannel = toFile.getChannel();

long position = 0;
long count     = fromChannel.size();
toChannel.transferFrom(fromChannel, position, count);
```

# ASSIGNMENT 8: VALUTAZIONE IO BUFFER

- scopo dell'assignment è dare una valutazione delle prestazioni di diverse strategie di bufferizzazione di I/O offerte da JAVA
- scrivere un programma che copi un file di input in un file di output, utilizzando le seguenti modalità alternative di bufferizzazione, valutando il tempo impiegato per la copia del file in ognuna delle seguenti strategie:
  - FileChannel con buffer indiretti
  - FileChannel con buffer diretti
  - FileChannel utilizzando l'operazione transferTo()
  - Buffered Stream di I/O
  - stream letto in un byte-array gestito dal programmatore
- confrontare le prestazioni delle diverse soluzioni, variando la dimensione del file (da qualche kbyte fino ad almeno una decina di Megabyte) e la dimensione del buffer
- riportare i risultati ottenuti nel sorgente, in un commento

# **Reti e Laboratorio III**

## **Modulo Laboratorio III**

### **AA. 2022-2023**

**docente: Laura Ricci**

**[laura.ricci@unipi.it](mailto:laura.ricci@unipi.it)**

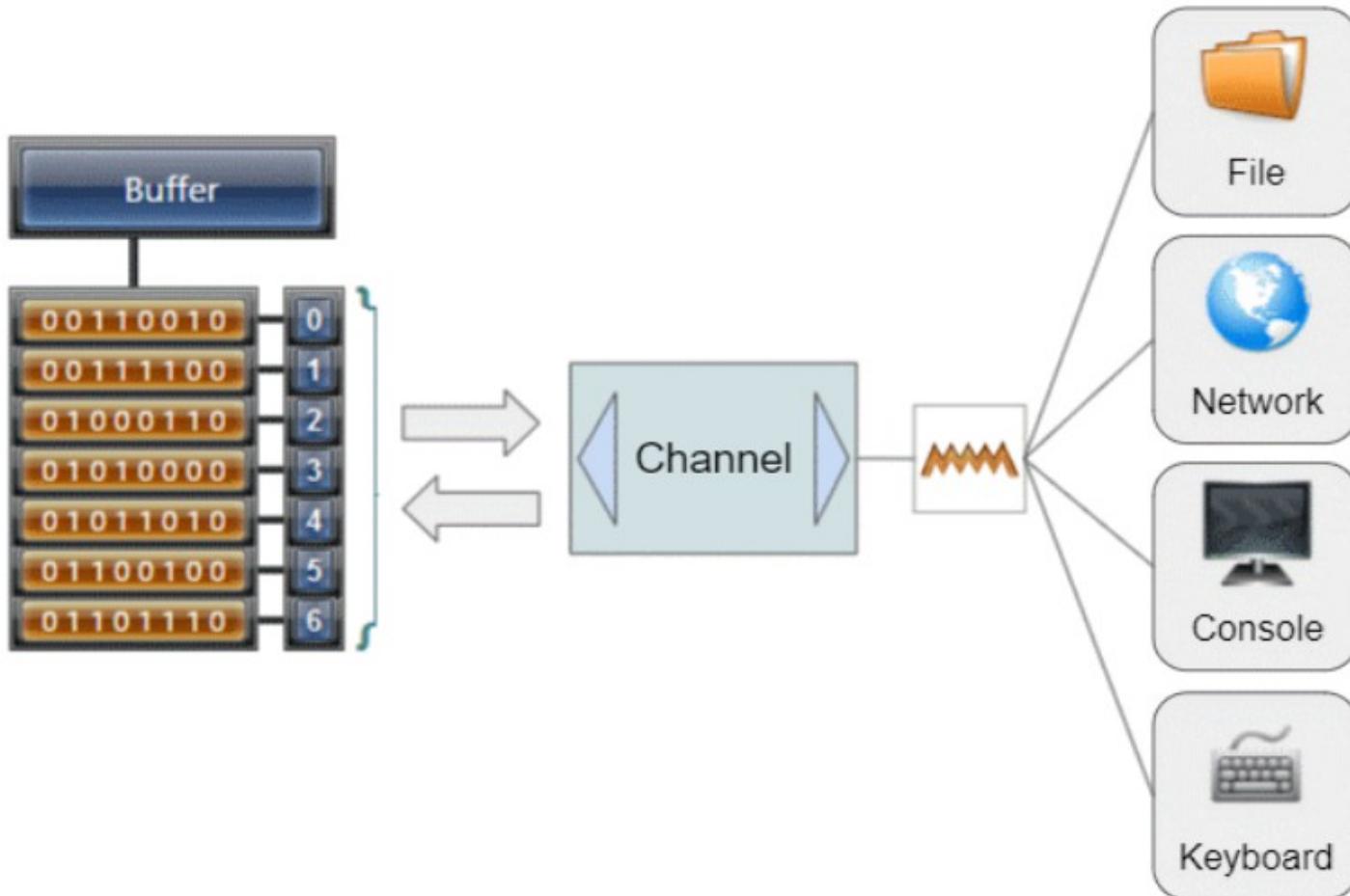
### **Lezione 9**

### **JAVA NIO:**

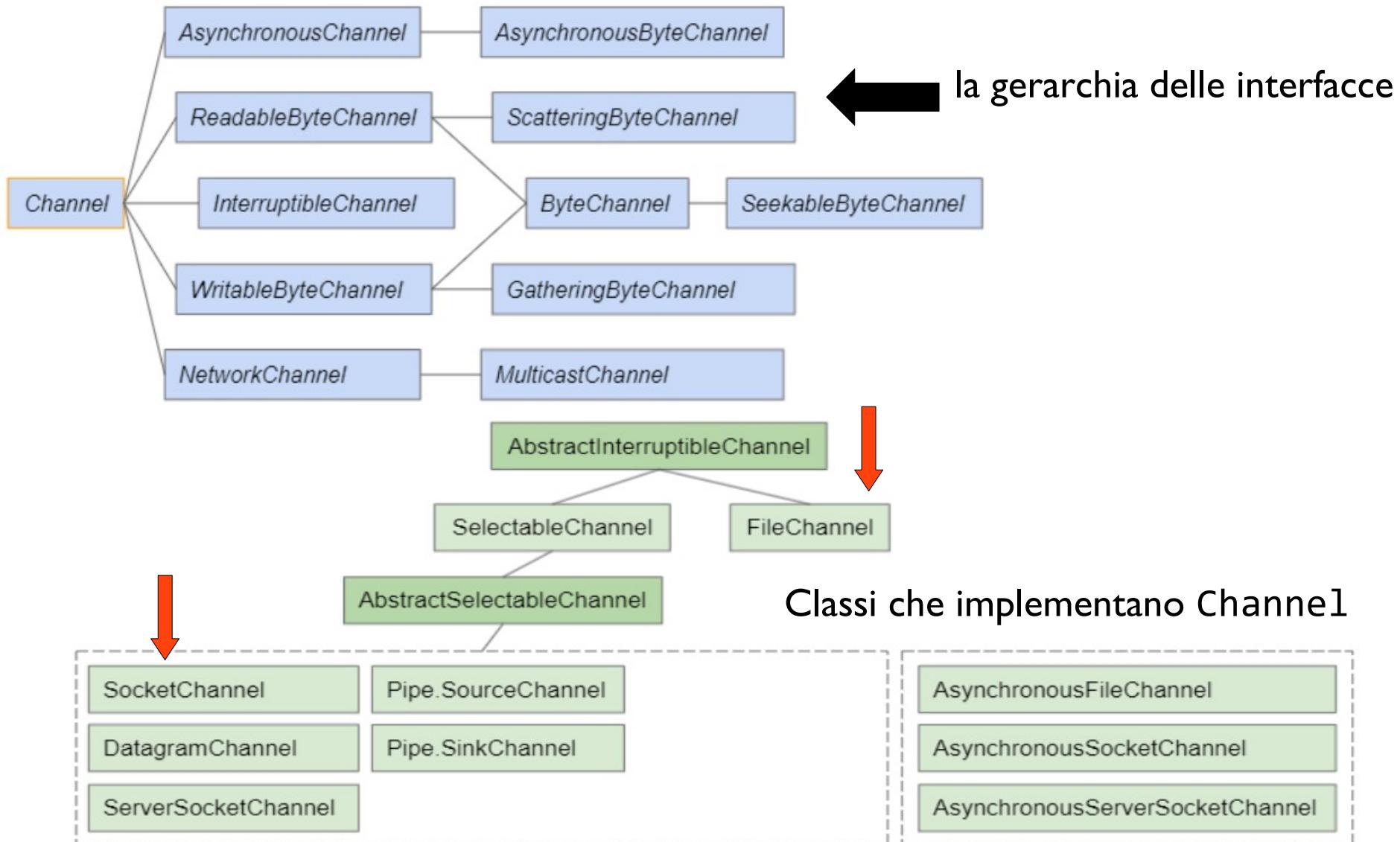
# **CHANNEL MULTIPLEXING**

**17/11/2022**

# JAVA NIO: CHANNEL



# JAVA NIO.CHANNEL



# JAVA NIO: OBIETTIVI

- fast buffered binary e character I/O

*“provide new features and improved performance in the areas of buffer management, scalable network and file I/O, character-set support, and regular-expression matching”*

- “non blocking mode” e multiplexing

*“production-quality web and application servers that scale well to thousands of open connections and can easily take advantage of multiple processors”*

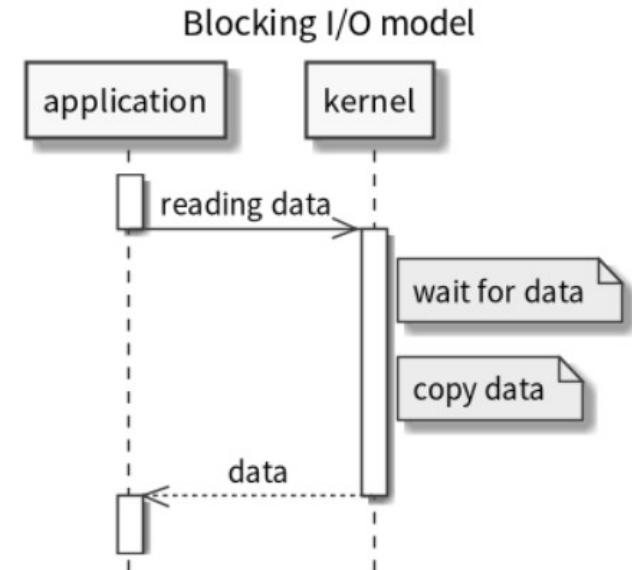
in questa lezione:

- non blocking channels associati a socket
- multiplexing: Selector



# IL MODELLO BLOCKING IO

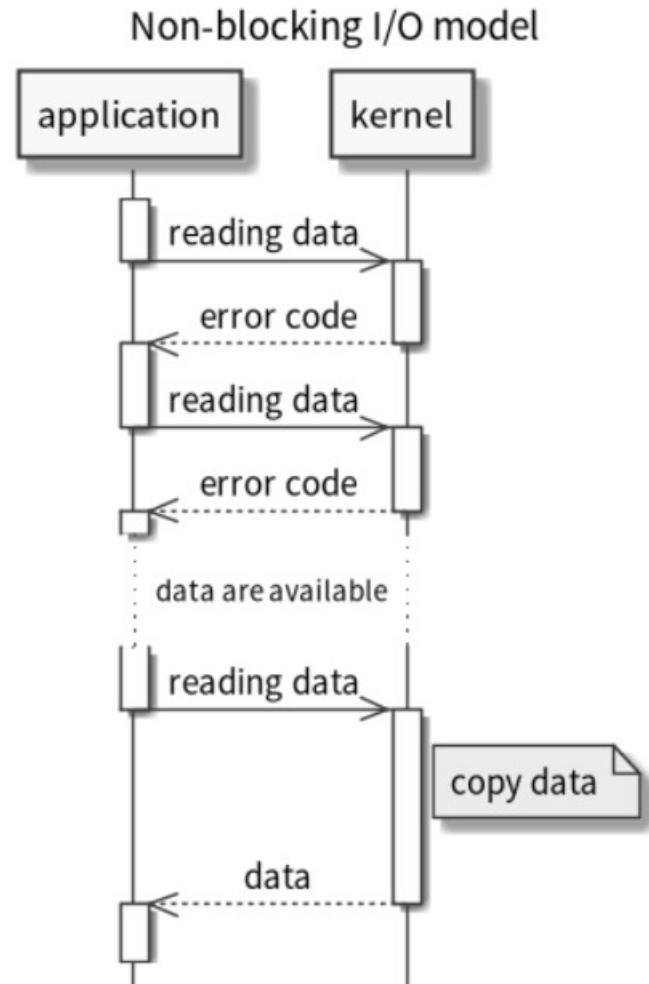
- operazioni bloccanti **su stream**: l'applicazione esegue una chiamata di sistema e si blocca fino a che tutti i dati sono ricevuti nel kernel, e copiati dal kernel space alla memoria della applicazione
- **read( )**
  - si blocca fino a quando non è stato letto un byte, un vettore di byte, un intero,...



- **accept( )**
  - si blocca fino a che non viene stabilita una nuova connessione
- **write(byte [ ] buffer)**
  - si blocca fino a che tutto il contenuto del buffer è stato copiato sulla periferica di I/O

# IL MODELLO NON-BLOCKING IO

- la chiamata di sistema restituisce il controllo alla applicazione prima che l'operazione richiesta sia stata “pienamente soddisfatta”.
- scenari possibili
  - restituiti i dati disponibili, o una parte di essi
  - operazione I/O non possibile: codice errore o valore null
- per completare l'operazione
  - effettuare system-call ripetute, fino a che l'operazione può essere effettuata
  - possibile con canali associati a socket



# SOCKET CHANNEL

- non blocking I/O è possibile per channel associati ai socket
- un channel associato ad un socket TCP “combina” un socket con un canale di comunicazione bidirezionale
  - scrive e legge da un socket TCP
  - estende la classe AbstractSelectableChannel e da questa mutua la capacità di passare dalla modalità bloccante a quella **non bloccante**
  - in modalità bloccante funzionamento simile a quello degli stream socket, ma con interfaccia basata su buffers
- classi SocketChannel, SocketServerChannel
- ognuno di essi associato ad un oggetto Socket della libreria java.net
  - il socket può essere reperito mediante il metodo socket(), applicato al channel

# SERVER SOCKET CHANNEL

- ad ogni ServerSocketChannel è associato un oggetto ServerSocket
  - **blocking**: come ServerSocket, ma con interfaccia buffer-based
  - **non blocking**: permette multiplexing di canali

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
ServerSocket socket = serverSocketChannel.socket();
socket.bind(new InetSocketAddress(9999));
serverSocketChannel.configureBlocking(false);
while(true){
    SocketChannel socketChannel = serverSocketChannel.accept();
    if(socketChannel != null){
        //do something with socketChannel...
    } else //do something useful... }
```

- notare l'uso di oggetti di tipo InetSocketAddress

# SOCKET CHANNEL

- associati ad un oggetto di tipo Socket
- creazione di un SocketChannel
  - implicita: creato se si accetta una connessione su un ServerSocketChannel.
  - esplicita, **lato client**, quando si apre una connessione verso un server, mediante una operazione di connect()

```
SocketChannel socketChannel = SocketChannel.open();
socketChannel.connect (new InetSocketAddress("www.google.it", 80));
```

InetSocketAddress può essere specificato direttamente nella open, in questo caso viene effettuata implicitamente la connect

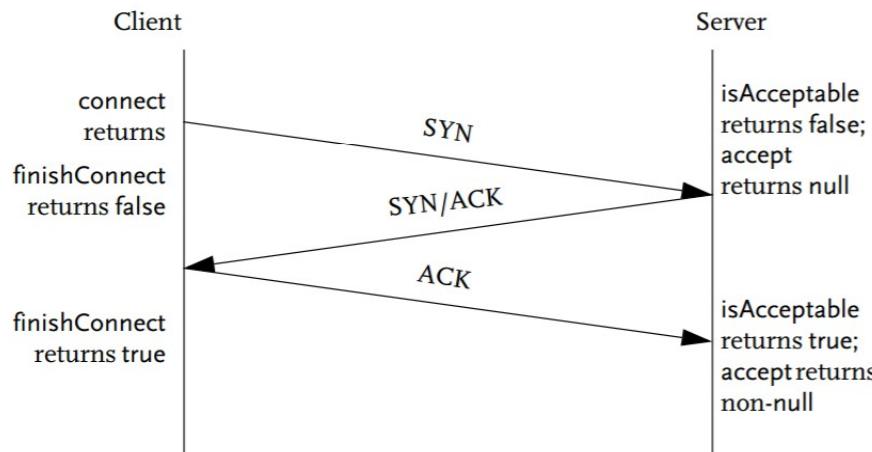
- modalità blocking/non blocking:

```
SocketChannel.configureBlocking(false);
```

- non blocking, lato client, significativa ad esempio nel caso in cui un'applicazione deve gestire l'interazione con l'utente, mediante GUI, e l'apertura del socket

# NON BLOCKING CONNECT

- può restituire il controllo al chiamante prima che venga stabilita la connessione.



**isAcceptable**

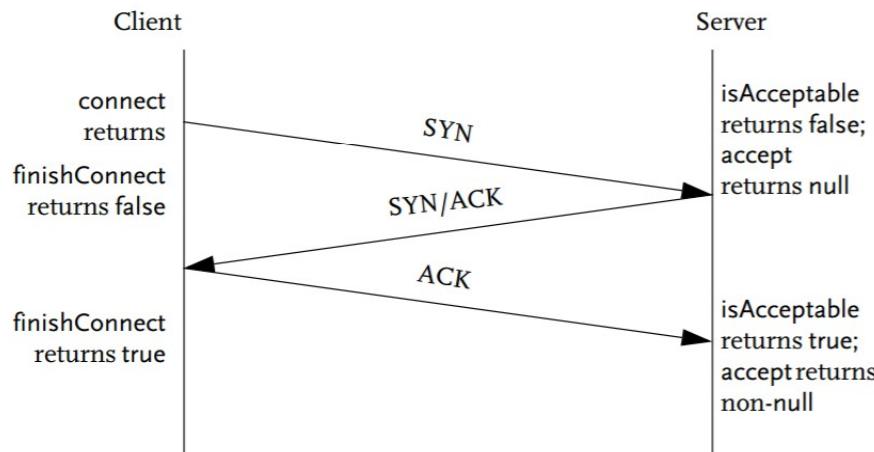
restituisce vero quando la connessione può essere accettata

- `finishConnect()` per controllare la terminazione della operazione.

```
socketChannel.configureBlocking(false);  
socketChannel.connect(new InetSocketAddress("www.google.it", 80));  
while(! socketChannel.finishConnect() ){  
    //wait, or do something else...    }  
}
```

# NON BLOCKING CONNECT

- può restituire il controllo al chiamante prima che venga stabilita la connessione.



**isAcceptable**

restituisce vero quando la connessione può essere accettata

- se l'ultima fase del three way handshake non è completo quando il client effettua la read, la read restituerà 0 valori nel buffer
- se si toglie

```
while(! socketChannel.finishConnect() )  
    { //wait, or do something else... }
```

viene sollevata **java.nio.channels.NotYetConnectedException**

# BLOCKING E NON BLOCKING: RIASSUNTO

- Accept
  - blocking: si blocca finchè non arriva una richiesta di connessione
  - non blocking: controlla se c'è una richiesta da accettare e ritorna comunque
- Write
  - blocking si blocca finchè la scrittura dei dati nel buffer non è completata
  - non blocking tenta di scrivere i dati nella socket, ritorna immediatamente, anche se i dati non sono stati completamente scritti
- Read:
  - blocking: si blocca in attesa di byte da leggere
  - non-blocking: ritorna immediatamente e restituisce il numero di byte letti (anche 0)

criteri per la valutazione delle prestazioni di un server:

- **scalability**: capacità di servire un alto numero di client che inviano richieste concorrentemente
- **acceptance latency**: tempo tra l'accettazione di una richiesta da parte di un client e la successiva
- **reply latency**: tempo richiesto per elaborare una richiesta ed inviare la relativa risposta
- **efficiency**: utilizzo delle risorse utilizzate sul server (RAM, numero di threads, utilizzo della CPU)

# UN SINGOLO THREAD

Un solo thread per tutti client:

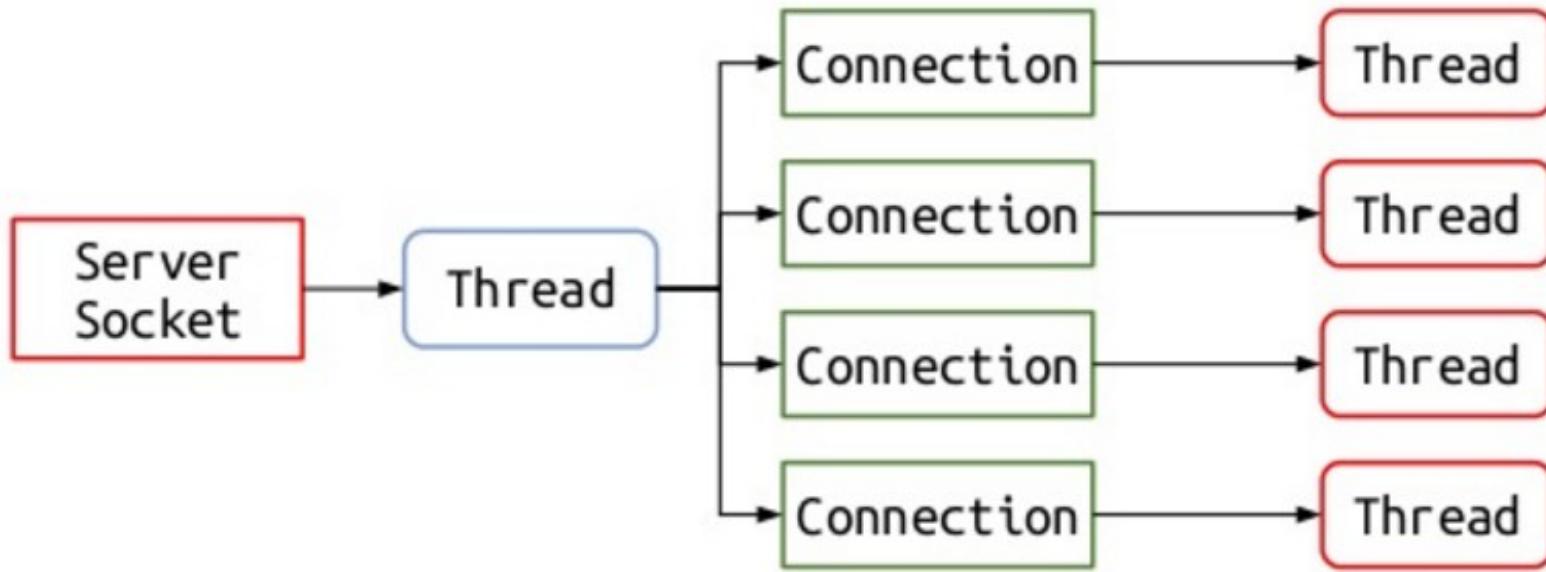
- **scalabilità:** nulla, in ogni istante, solo un client viene servito
- **accept latency:** alta, il “prossimo” cliente deve attendere fino a che il primo cliente termina la connessione
- **reply latency bassa:** tutte le risorse a disposizione di un singolo client
- **efficiency:** buona, il server utilizza esattamente le risorse necessarie per il servizio dell'utente.
- adatto quando il tempo di servizio di un singolo utente è garantito rimanere rimanga basso



# UN THREAD PER OGNI CONNESSIONE

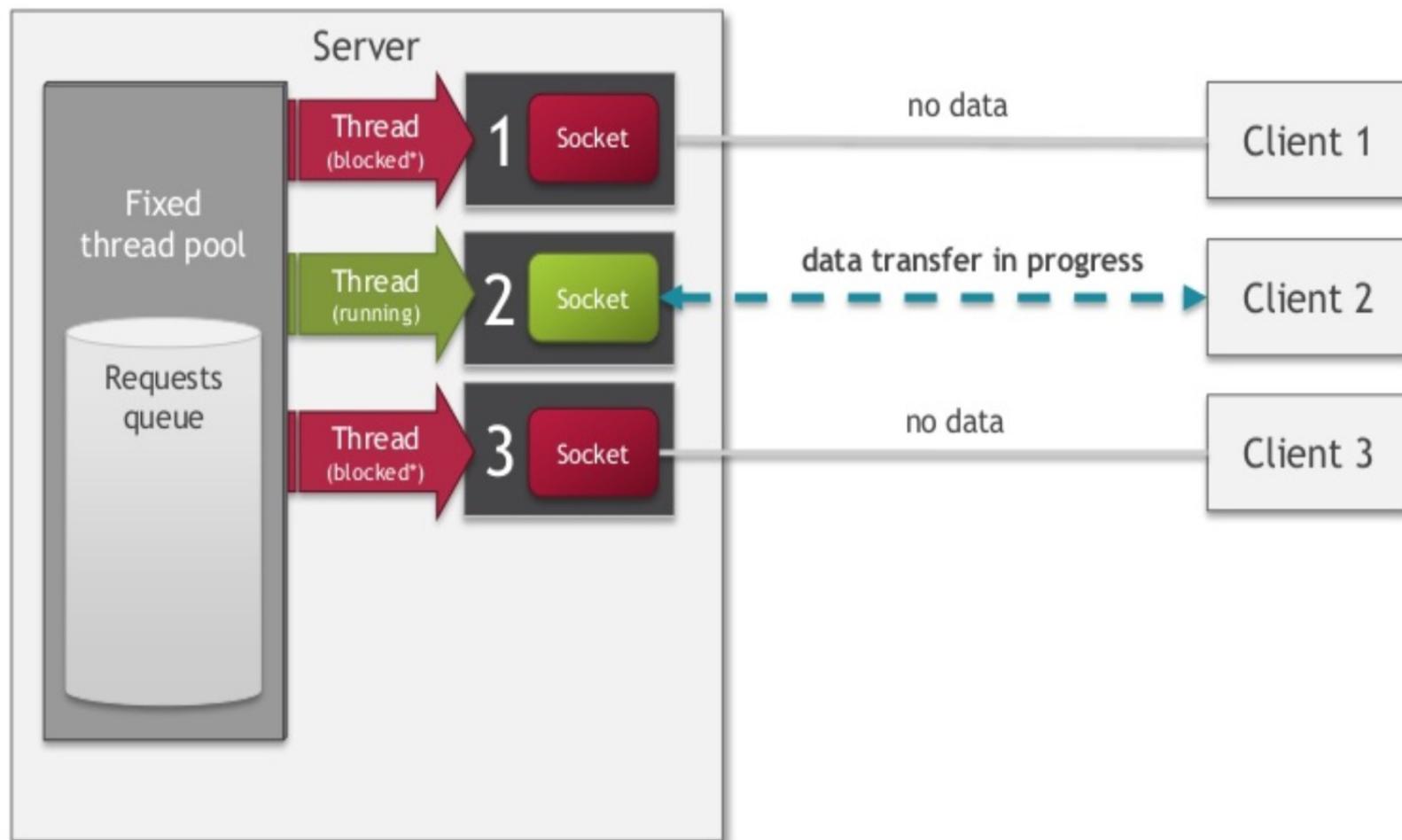
- **scalabilità:** possibile servire diversi clienti in maniera concorrente, fino al massimo numero di thread previsti per ogni processo
  - ogni thread alloca il proprio stack: memory pressure
  - impossibile predire il numero massimo di client: dipende da fattori esterni e può essere molto variabile
- **accept latency:** tempo tra l'accettazione di una connessione e la successiva è in genere basso rispetto a quello di interarrivo delle richieste
- **reply latency:** bassa, le risorse del server condivise tra connessioni diverse
  - ragionevole uso di CPU e RAM per centinaia di connessioni, se aumenta, il tempo di reply può non essere accettabile
- **efficiency:** bassa
  - ogni thread può essere bloccato in attesa di IO, ma utilizza risorse come la RAM

# UN THREAD PER OGNI CONNESSIONE



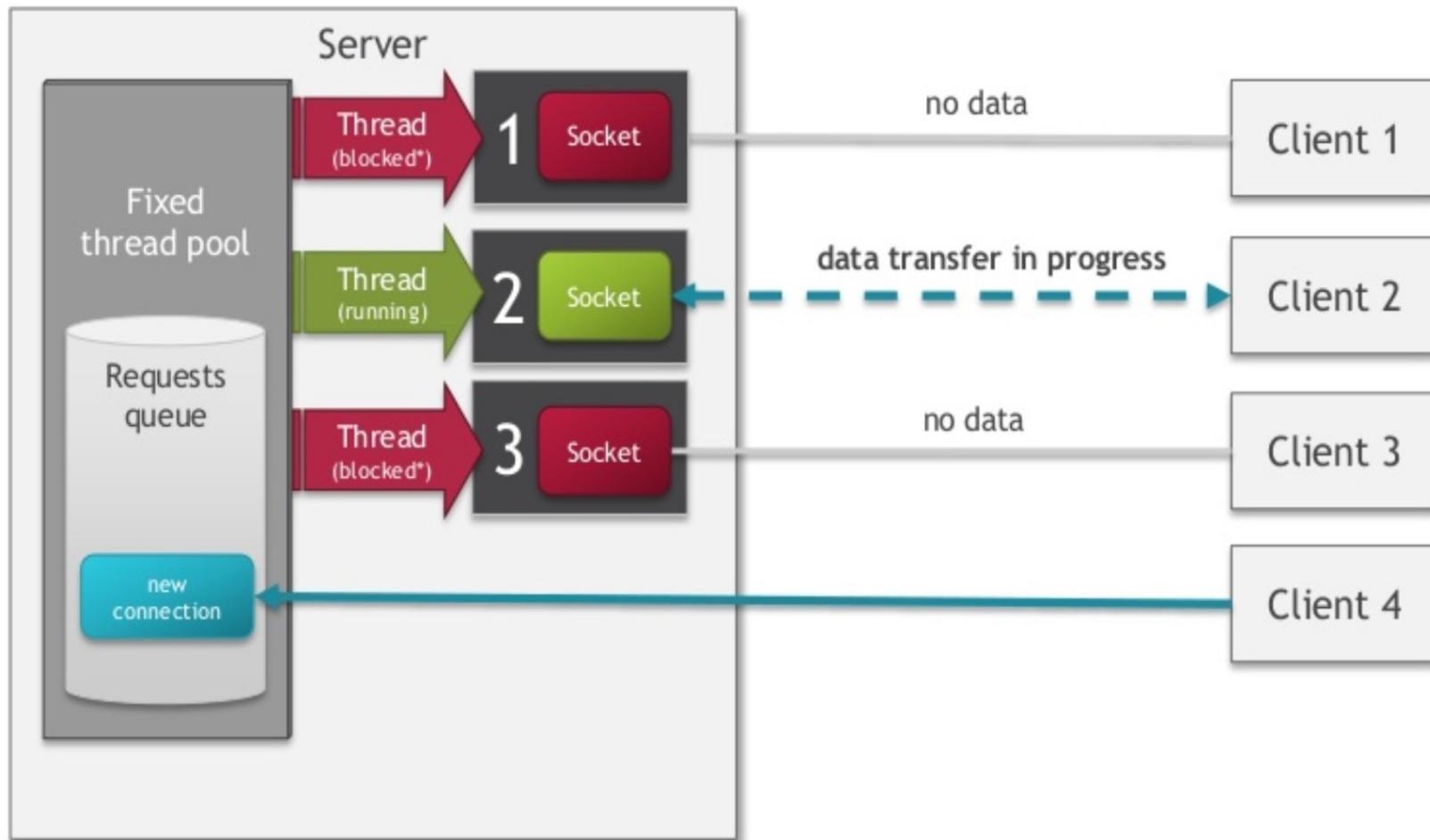
- attivazione di un thread per ogni connessione, de-attivazione a fine servizio
- quando un server monitora un grande numero di comunicazioni:
  - problemi di scalabilità: il tempo per il cambio di contesto può aumentare notevolmente con il numero di thread attivi
  - maggior parte del tempo impiegata in context switching

# UN THREAD PER CONNESSIONE: FIXEDTHREADPOOL



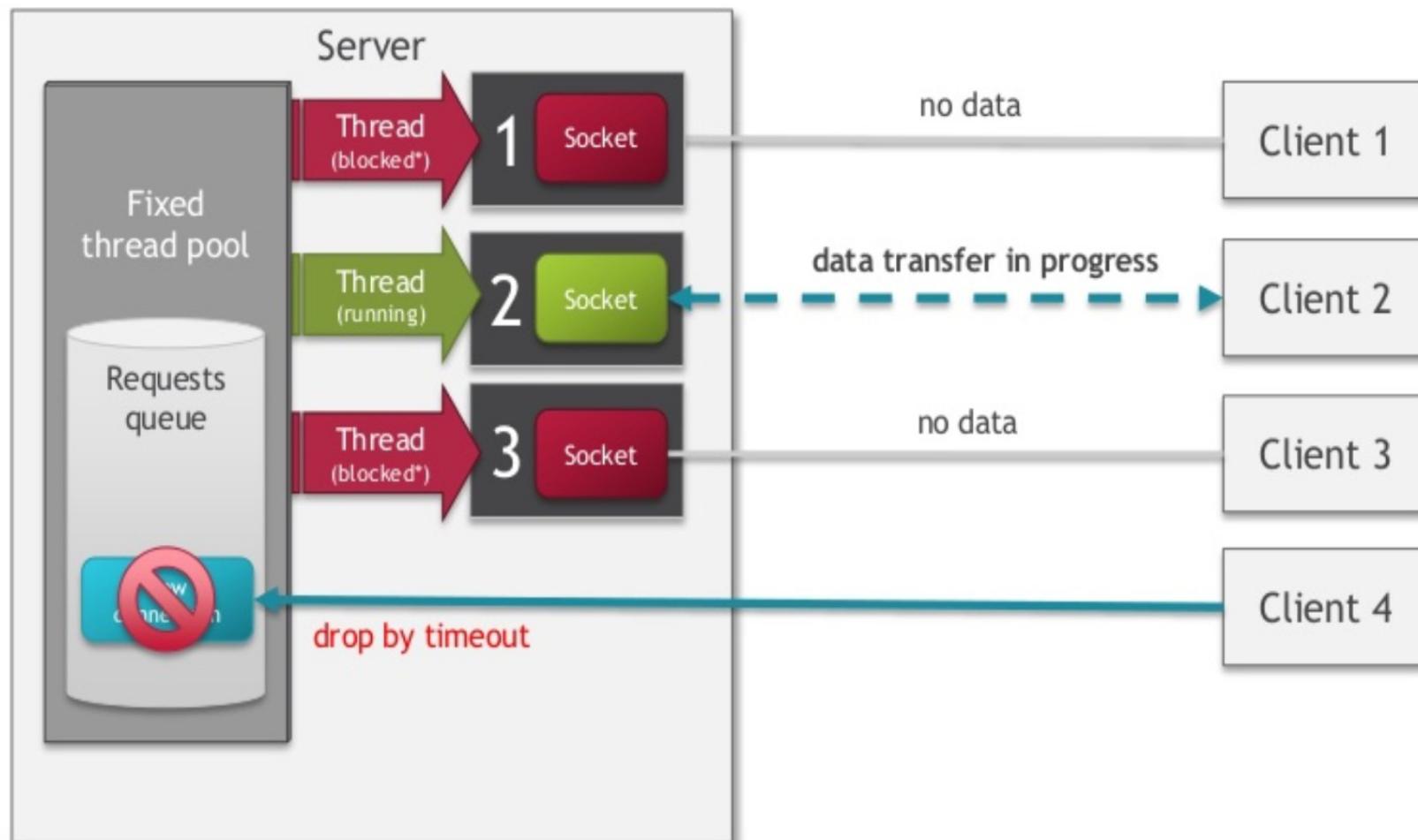
\*until keep alive timeout

# UN THREAD PER CONNESSIONE: FIXEDTHREADPOOL



\*until keep alive timeout

# UN THREAD PER CONNESSIONE: FIXEDTHREADPOOL



\*until keep alive timeout

# UN NUMERO FISSO DI THREAD

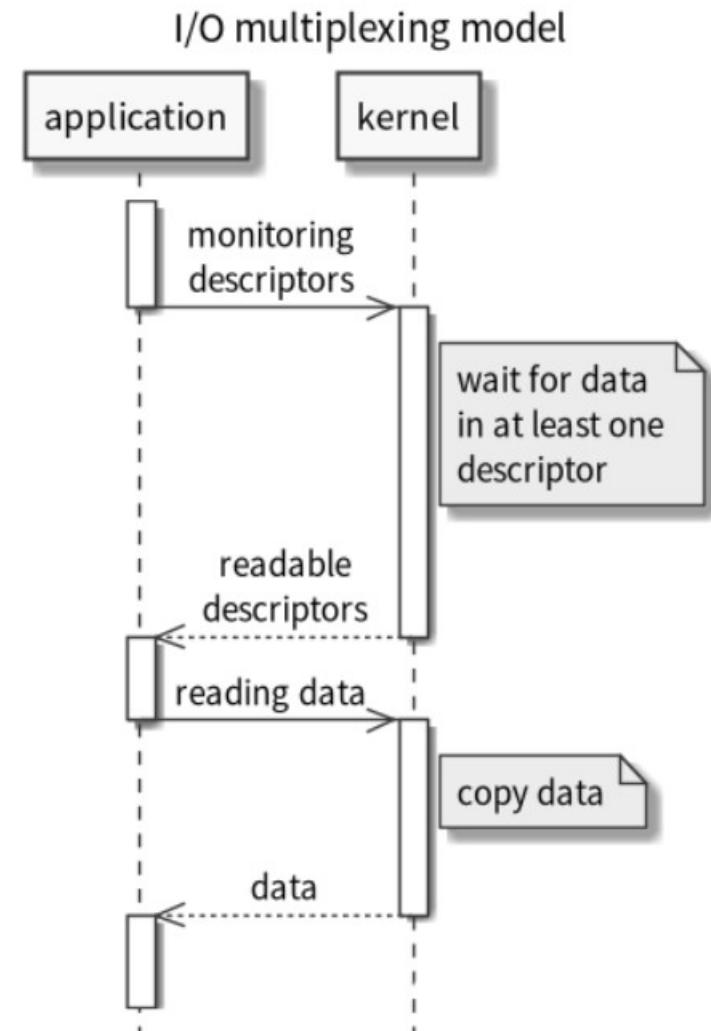
un numero costante di thread: utilizza Thread Pool

- **scalabilità:** limitata al numero di connessioni che possono essere supportate.
- **accept latency** bassa fino ad un certo numero di connessioni
- **reply latency:** bassa fino al numero massimo di thread fissato, degrada se il numero di connessioni è maggiore
- **efficiency:** trade-off rispetto al modello precedente

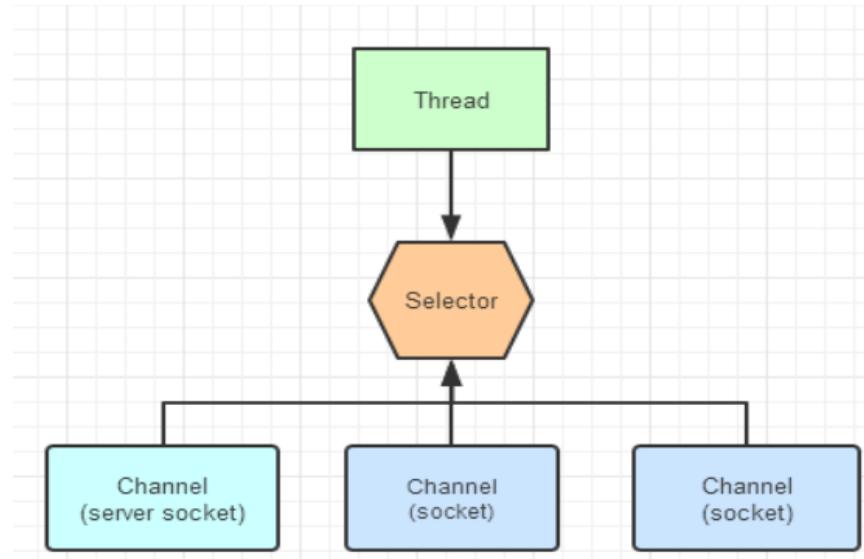


# MULTIPLEXED I/O

- non blocking I/O con notifiche bloccanti
- l'applicazione registra “descrittori” delle operazioni di I/O a cui è interessato
- l'applicazione esegue una operazione di **monitoring** di canali
  - una system call bloccante
  - restituisce il controllo quando almeno un descrittore indica che una operazione di I/O è “pronta”
  - a quel punto si effettua una read non bloccante

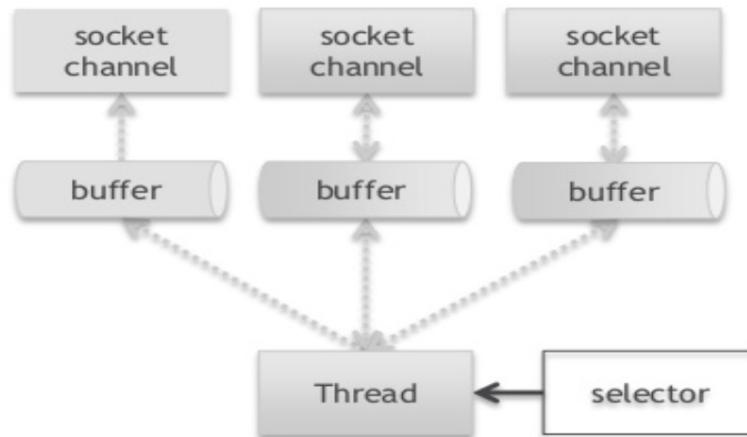


# MULTIPLEXING IN JAVA



- **selettore** un componente che esamina uno o più NIO Channels, e determina quali canali sono pronti per leggere/scrivere
- più connessioni di rete gestite mediante un **unico thread**, consente di ridurre
  - thread switching overhead
  - uso di risorse per thread diversi
- possibile anche l'utilizzazione insieme insieme a multithreading

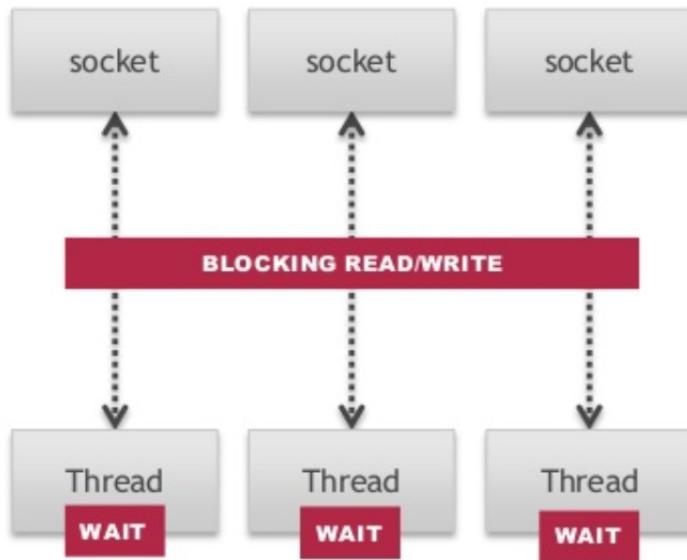
# UN UNICO THREAD: MULTIPLEXING



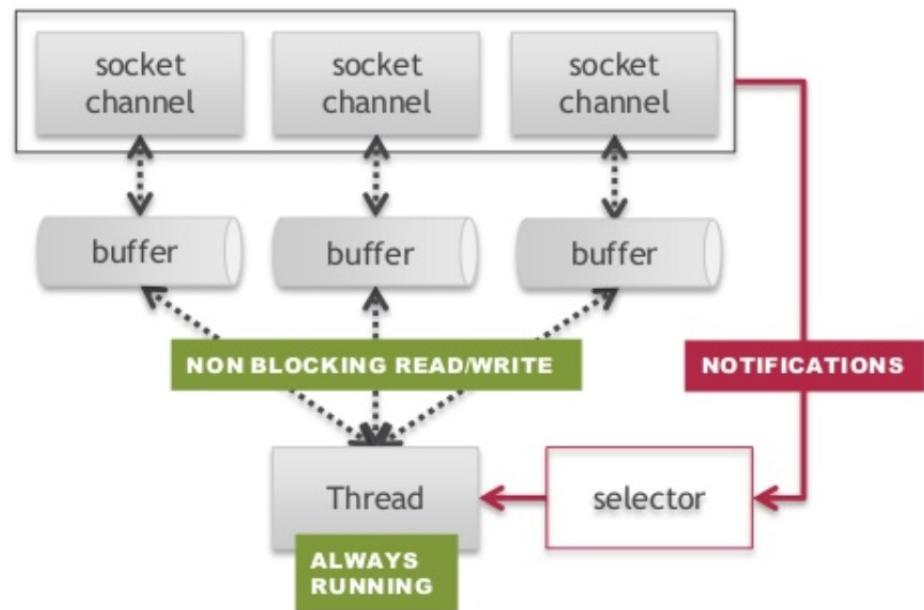
- un singolo thread che gestisce un numero arbitrario di sockets
- non un thread per connessione, ma un numero ridotto di threads
  - numero di thread basso anche con migliaia di sockets
  - caso limite: un solo thread
- miglioramento di performance e scalabilità
- architettura più complessa da capire e da implementare

# UN UNICO THREAD: MULTIPLEXING

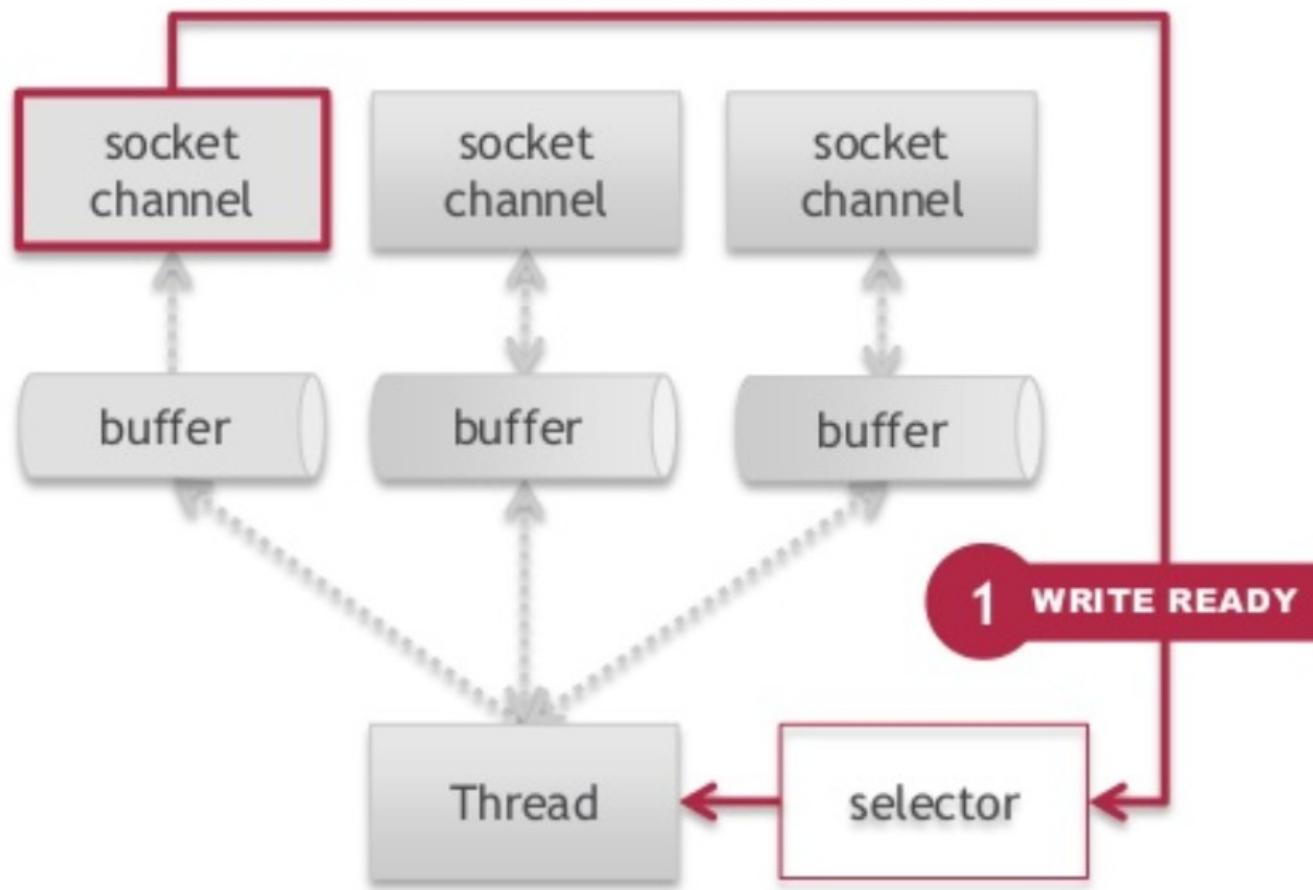
## IO (blocking)



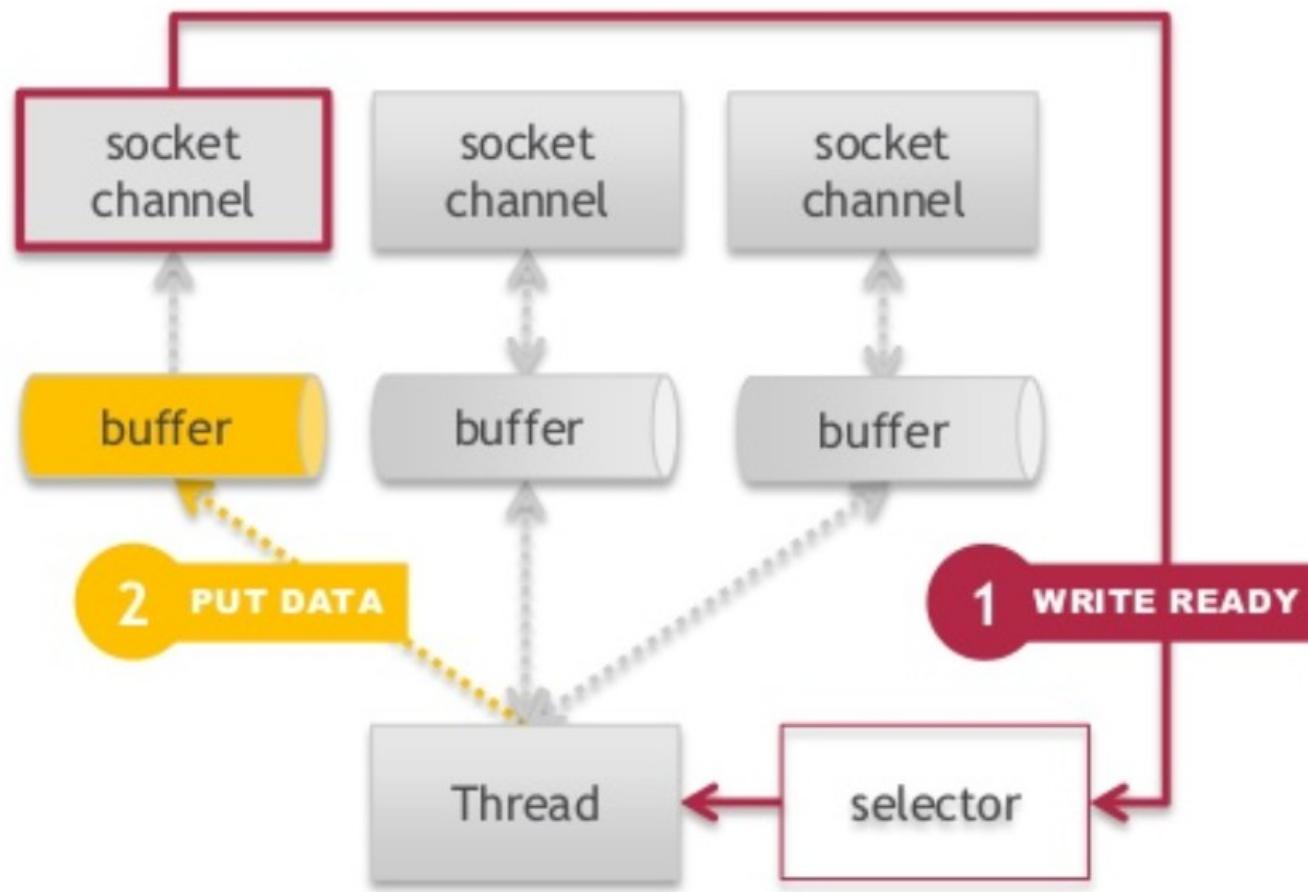
## NIO (non-blocking)



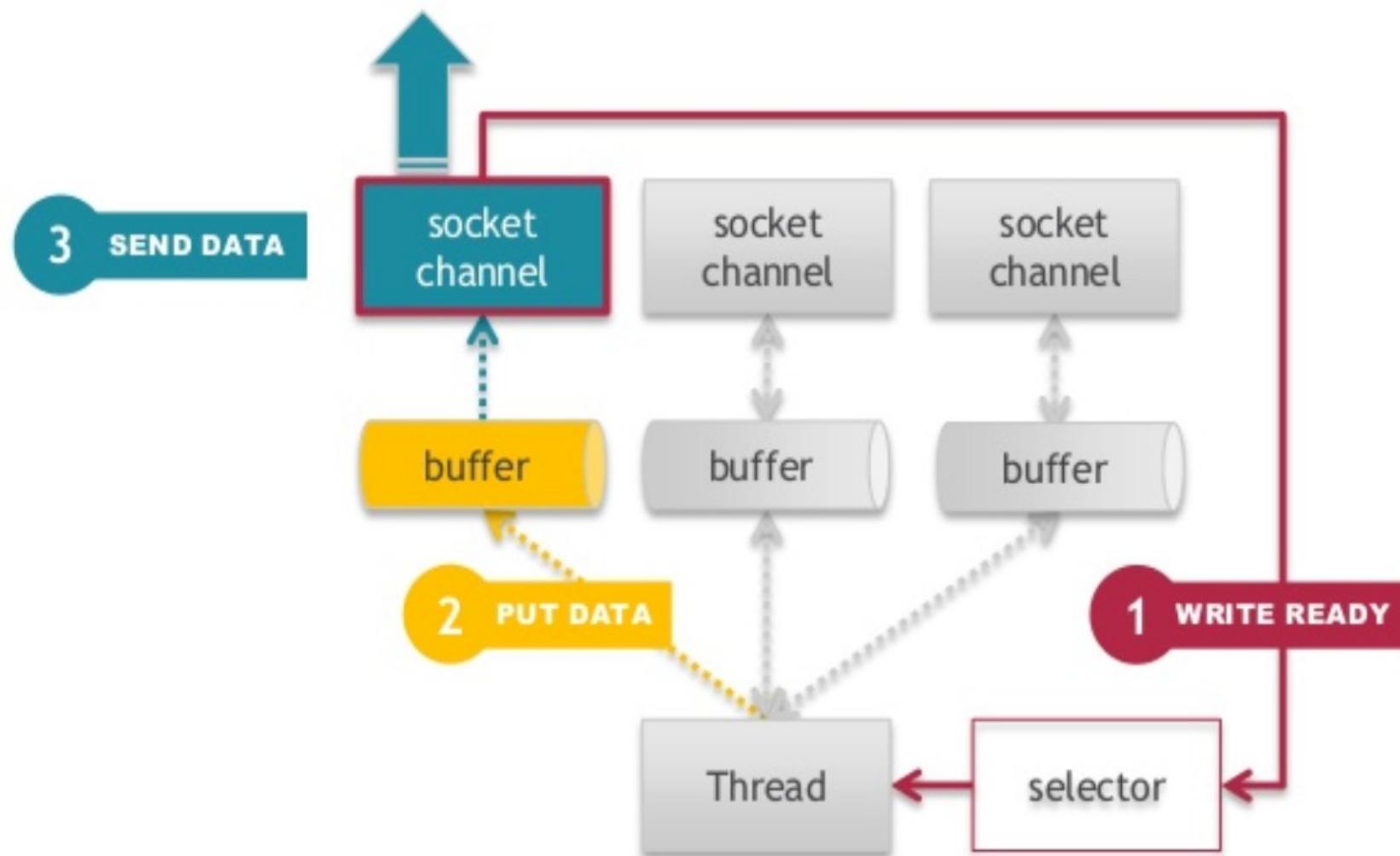
# MULTIPLEXING: INVIO DEI DATI



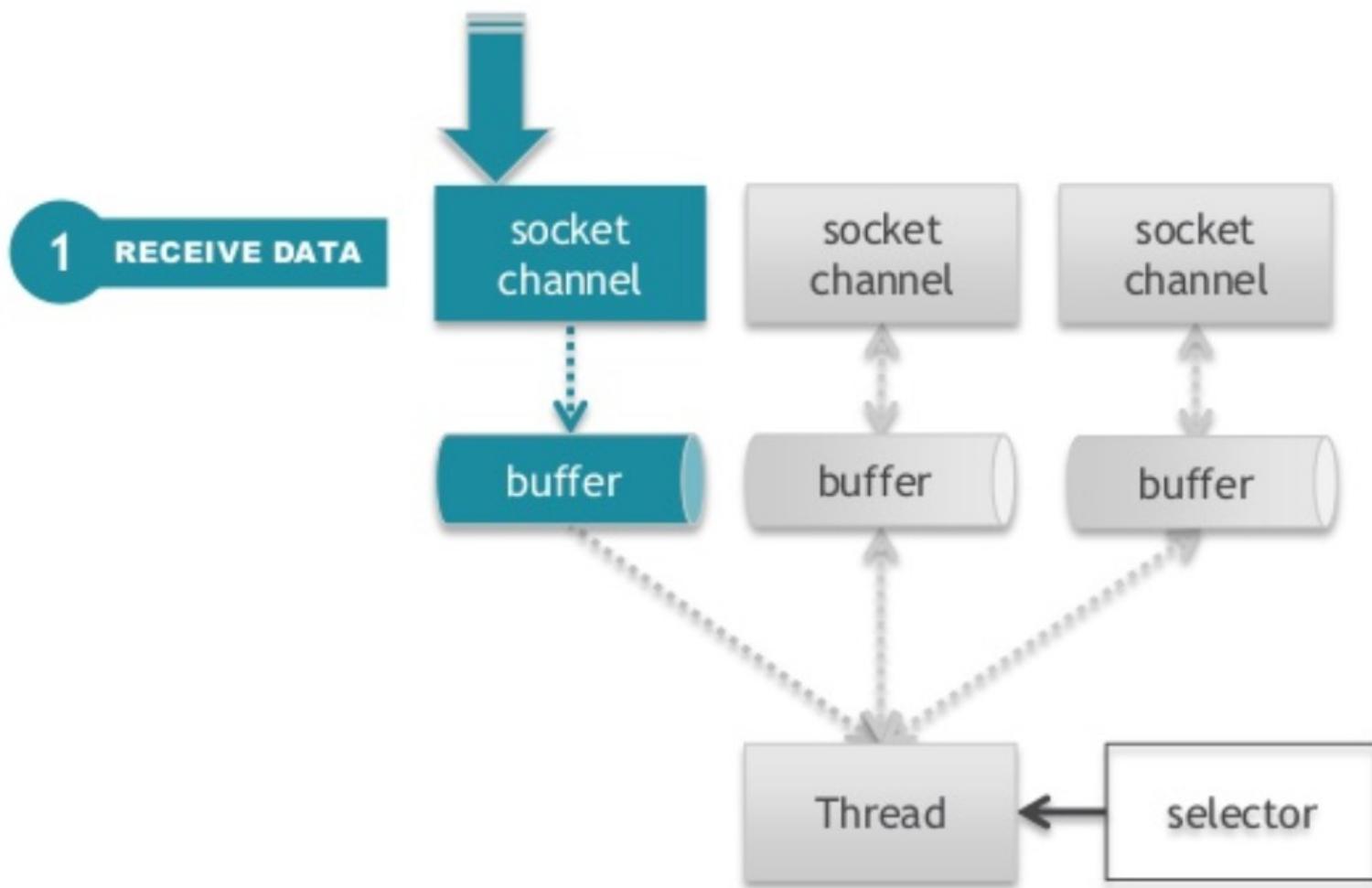
# MULTIPLEXING: INVIO DEI DATI



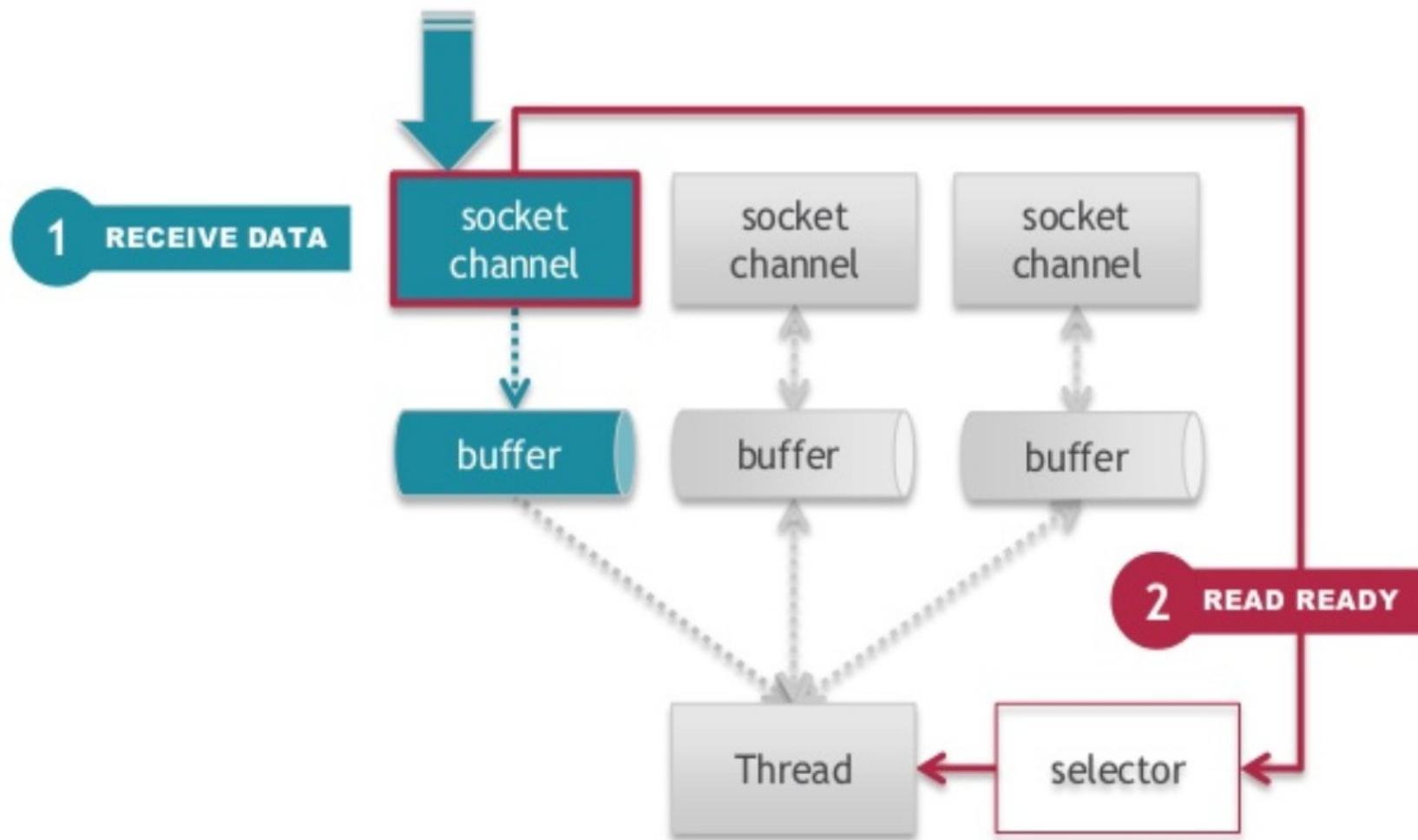
# MULTIPLEXING: INVIO DEI DATI



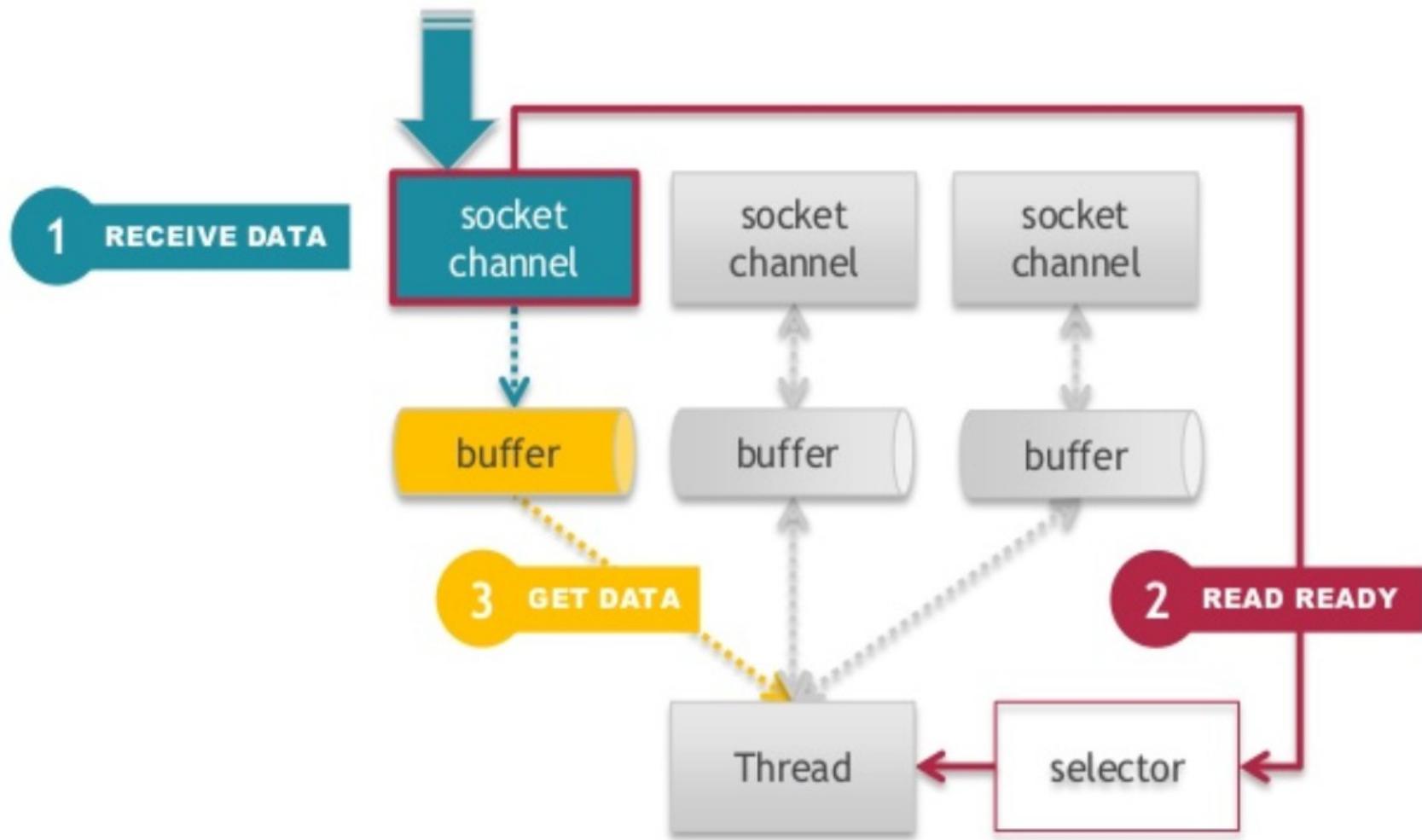
# MULTIPLEXING: RICEZIONE DEI DATI



# MULTIPLEXING: RICEZIONE DEI DATI



# MULTIPLEXING: RICEZIONE DEI DATI



# L'OGGETTO SELECTOR

- componente base per il multiplexing

```
Selector selector = Selector.open();
```

- permette di selezionare un SelectableChannel che è pronto per operazioni di rete
  - accept, write, read, connect
  - stesso thread che gestisce più eventi che possono avvenire simultaneamente
- selectable channels
  - ServerSocketChannel
  - SocketChannel
  - DatagramChannel
  - Pipe.SinkChannel
  - Pipe.SourceChannels
  - file non inclusi



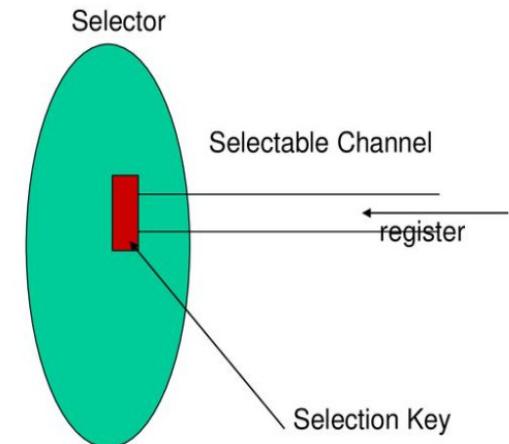
# REGISTRAZIONE DEI CANALI: SELECTION KEYS

- registrazione di un **canale** su un **selettore**

```
channel.configureBlocking(false);
```

```
Selectionkey key = channel.register(selector, ops, attach);
```

- il canale deve essere in modalità non bloccante
  - non si possono usare Filechannels con i Selector
- secondo parametro della register (ops) è l' "interest set"
  - indica quali eventi si è interessati a monitorare su quel canale



```
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

- terzo parametro della register (attach) è un buffer associato al canale

# L'INTEREST SET COME BITMASK

- bitmask di 8 bit (un intero) codifica le operazioni di interesse su quel canale
- attualmente sono supportati 4 tipi di operazioni, ad ogni operazione corrisponde una bitmask

• connect	OP_READ - 1	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1			
• accept	OP_WRITE - 4	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0			
• read	OP_ACCEPT - 16	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0			
• write	OP_READ   OP_WRITE - 5	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr></table>	0	0	0	0	0	1	0	1
0	0	0	0	0	1	0	1			

- è manipolato con gli operatori JAVA |, &, ^, ~, che eseguono operazioni bit a bit su operandi interi o booleani
- non tutte le operazioni valide per tutti i SelectableChannel, ad esempio SocketChannel non supporta accept()

# L'INTEREST SET COME BITMASK

- nella classe SelectionKey, 4 costanti predefinite che corrispondono alle bitmask

- 1 SelectionKey.OP\_CONNECT
- 2 SelectionKey.OP\_ACCEPT
- 3 SelectionKey.OP\_READ
- 4 SelectionKey.OP\_WRITE

- utilizzabili in fase di registrazione del canale con il Selector per impostare il valore iniziale dell'Interest Set

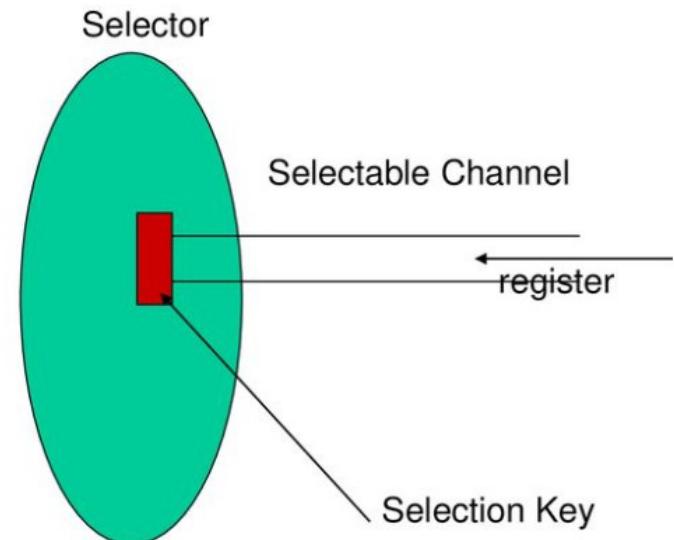
```
Selector selector = Selector.open();
channel.register(selector,SelectionKey.OP_READ | SelectionKey.OP_WRITE);
```

- per reperibile l'interest set

```
int interestSet = selectionKey.interestOps();
```

# REGISTRAZIONE DEI CANALI: SELECTION KEYS

- ogni registrazione di un canale su un selettore
  - restituisce una chiave, un “token” che la rappresenta
  - un oggetto di tipo SelectionKey.
  - valida fino a che non viene cancellata esplicitamente
- lo stesso canale può essere registrato con più selettori
  - una chiave diversa per ogni registrazione.



# L'OGGETTO SELECTION KEY

è il risultato della registrazione di un canale su un selettore e memorizza

- il canale a cui si riferisce
- il selettore a cui si riferisce
- l'**interest set**
  - utilizzato quando il metodo select viene invocato per monitorare i canali del selettore
  - definisce le operazioni su cui si deve fare il controllo di “readiness”,
- il **ready set**
  - dopo la invocazione della select, contiene gli eventi che sono pronti su quel canale
- un allegato, **attachment**
  - uno spazio di memorizzazione associato a quel canale per quel selettore

# REGISTRAZIONE CANALI: PATTERN GENERALE

```
// Crea il socket channel e configuralo come non bloccante  
  
ServerSocketChannel server = ServerSocketChannel.open();  
server.configureBlocking(false);  
server.socket().bind(new java.net.InetSocketAddress(host,8000));  
System.out.println("Server attivo porta 2001");  
  
// Crea il selettor e registra il server al Selector  
  
Selector selector = Selector.open();  
server.register(selector,SelectionKey.OP_ACCEPT, null);
```

Tipo di registrazione	Significato: il Selector riporta che ...
OP_ACCEPT	Il client richiede una connessione al server
OP_CONNECT	Il server ha accettato la richiesta di connessione
OP_READ	Il channel contiene dati da leggere
OP_WRITE	Il channel contiene dati da scrivere

L'eventuale allegato

# MULTIPLEXING DEI CANALI: LA SELECT

- **int selector.select( );**
  - bloccante, seleziona, tra i canali registrati sul selettore selector, quelli pronti per almeno una delle operazioni dell'interest set.
  - si blocca fino a che una delle seguenti condizioni è vera
    - almeno un canale è “pronto”
    - il thread che esegue la selezione viene interrotto
    - il selettore viene sbloccato mediante il metodo wakeup()
  - restituisce il numero di canali pronti
    - che hanno generato un evento dopo l'ultima invocazione della select()
    - costruisce un insieme contenente le SelectionKeys dei canali pronti
- **int select(**long** timeout)**
  - si blocca fino a che non è trascorso il timeout, oppure vale una delle condizioni precedenti
- **int selectNow()**
  - non bloccante, nel caso nessun canale sia pronto restituisce il valore 0

# IL READY SET

- aggiornato quando si esegue una operazione di monitoring dei canali, mediante una select
- identifica le chiavi per cui il canale è “pronto”, per l'esecuzione
  - sottoinsieme dell'interest set
  - interest set={read, write}                              ready set={read}
- inizializzato a 0 quando la chiave viene creata
- non può essere modificato direttamente
- operazioni su bitmask per verificare se si è verificato un evento

$$\forall 1 \leq i \leq 8 e_i \in \{0, 1\} \quad e$$

e <sub>8</sub>	e <sub>7</sub>	e <sub>6</sub>	e <sub>5</sub>	e <sub>4</sub>	e <sub>3</sub>	e <sub>2</sub>	e <sub>1</sub>
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

&

OP\_READ

0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

Different than 0 iff  $e_1 = 1$

0	0	0	0	0	0	0	$e_1$
---	---	---	---	---	---	---	-------

# IL READY SET

- restituito dal metodo readyOps( ) invocato su una SelectionKey
- supponiamo key sia una SelectionKey, per testare se ci sono dati pronti per essere letti

```
if ((key.readyOps( ) & SelectionKey.OP_READ) != 0)  
{ myBuffer.clear( );  
  key.channel( ).read (myBuffer);  
  doSomethingWithBuffer (myBuffer.flip( ));}
```

- shortcuts
  - key.isReadable() equivale a key.readyOps( )& SelectionKey.OP\_READ) != 0
  - analoghi shortcuts per le altre operazioni

# LA CLASSE SELECTION KEY

```
import java.nio.channels.*;

public abstract class SelectionKey

{public static final int OP_READ; public static final int OP_WRITE;
public static final int OP_CONNECT;Public static final int OP_ACCEPT;
public abstract SelectableChannel channel( );
public abstract Selector selector( );
public abstract void cancel( );
public abstract boolean isValid( );
public abstract int interestOps( );
public abstract void interestOps (int ops);
public abstract int readyOps( );
public final boolean isReadable( ) {};
public final boolean isWritable( ) {};
public final boolean isConnectable( ) {};
public final boolean isAcceptable( ) {};
public final Object attach (Object ob) {};
public final Object attachment( ) {};
```



# ANALISI PROCESSO DI SELEZIONE

ogni oggetto selettore mantiene, al suo interno, i seguenti insiemi di chiavi:

- **Key Set:**
  - SelectionKeys dei canali registrati con quel selettore.
  - restituito dal metodo **keys()**
- **Selected Key Set**
  - restituito dal metodo **selectedKeys()**, invocato sul selettore
  - insieme di chiavi precedentemente registrate tali per cui una delle operazioni nell'interest set è pronta per l'esecuzione
  - dopo una select() consente di accedere ai canali pronti per l'esecuzione di qualche operazione
- **Cancelled Key Set**
  - chiavi che sono state cancellate (quelle su cui è stato invocato il metodo **cancel()**, ma il cui canale è ancora registrato sul selettore

# COSA FA LA SELECT?

1. “delayed cancellation”: cancella ogni chiave appartenente al Cancelled Key Set dagli altri due insiemi.
2. interagisce con il sistema operativo per verificare lo stato di “readiness” di ogni canale registrato, per ogni operazione specificata nel suo interest set.
3. per ogni canale con almeno una operazione “ready”
  - se il canale già esiste nel Selected Key Set
    - aggiorna il ready set della chiave corrispondente: calcola l'or bit a bit tra il valore precedente del ready set e la nuova maschera
    - i bit ad 1 si “accumulano” con le operazioni pronte.
  - altrimenti
    - resetta il ready set e lo imposta con la chiave della operazione pronta
    - aggiunge la chiave al Selected Key Set

“comportamento cumulativo” della selezione

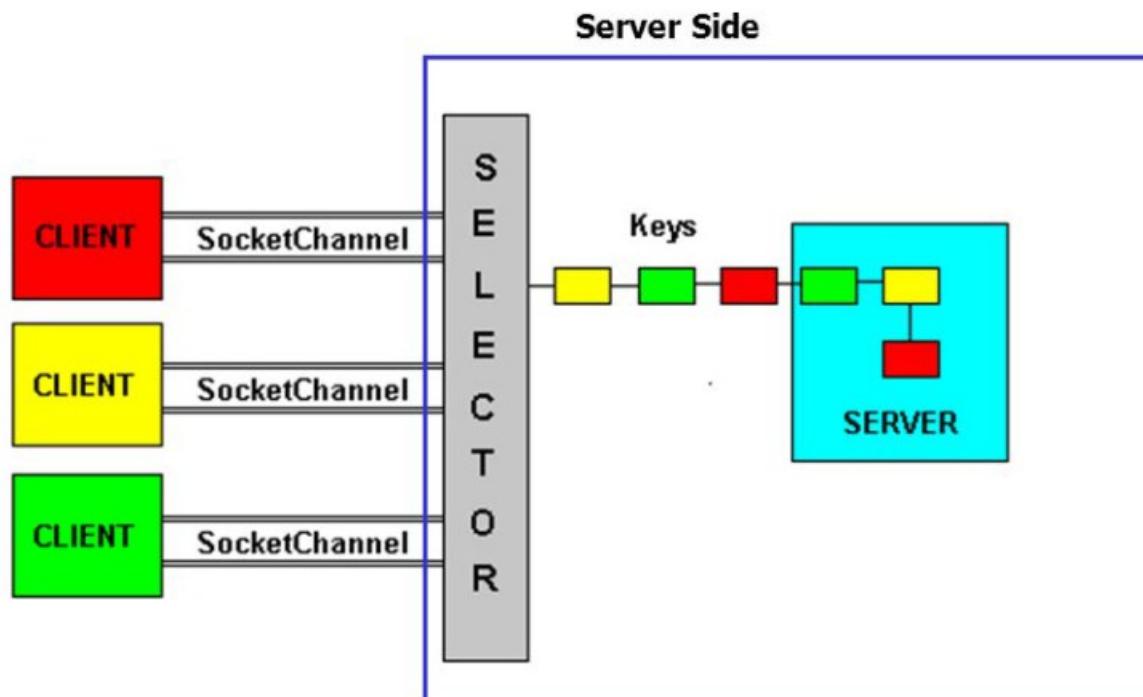
- una chiave aggiunta al selected key set, può essere rimossa solo con una operazione di rimozione esplicita
- il ready set di una chiave inserita nel selected key set, non viene mai resettato, ma viene aggiornato incrementalmente
- scelta di progetto: assegnare al programmatore la responsabilità di aggiornare esplicitamente le chiavi
- per resettare il ready set
  - rimuovere la chiave dall'insieme delle chiavi selezionate

# SELEZIONE: PATTERN GENERALE

```
Selector selector = Selector.open();
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
while(true) {
    int readyChannels = selector.selectNow();
    if(readyChannels == 0) continue;
    Set<SelectionKey> selectedKeys = selector.selectedKeys();
    Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
    while(keyIterator.hasNext()) {
        SelectionKey key = keyIterator.next();
        if(key.isAcceptable()) { // a connection was accepted by a ServerSocketChannel.
        } else if (key.isConnectable()) { // a connection was established with a remote
                                         Server (client side)
        } else if (key.isReadable()) { // a channel is ready for reading
        } else if (key.isWritable()) { // a channel is ready for writing }
        keyIterator.remove();
    }
}
```

# SELEZIONE: PATTERN GENERALE

- iterazione sull'insieme di chiavi che individuano i “canali pronti”
- dalla chiave si può ottenere un riferimento al canale su cui si è verificato l'evento
- keyIterator.remove() deve essere invocata, poiché il Selector non rimuove le chiavi

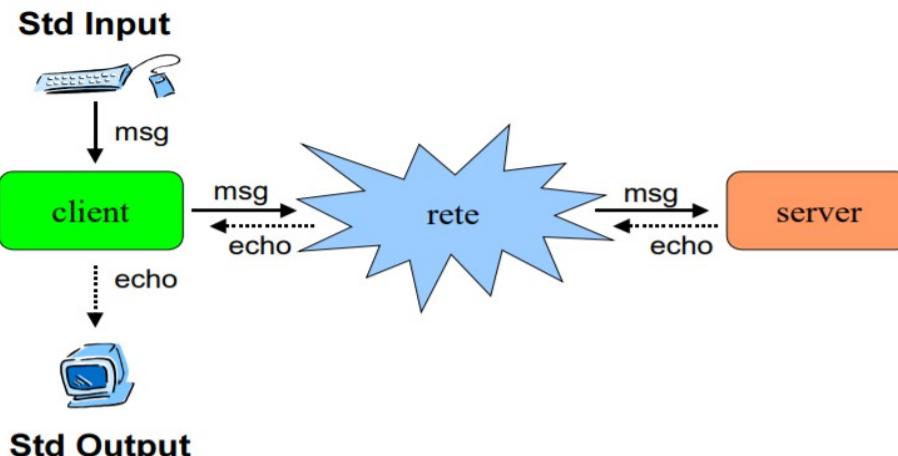


# SELECTION KEY: L'ATTACHMENT

- attachment: riferimento ad un generico Object
- utile quando si vuole accedere ad informazioni relative al canale (associato ad una chiave) che riguardano il suo stato pregresso
- necessario perchè le operazioni di lettura o scrittura non bloccanti non possono essere considerate atomiche
  - nessuna assunzione sul numero di bytes letti
- consente di tenere traccia di quanto è stato fatto in una operazione precedente.
  - l'attachment può essere utilizzato per accumulare i byte restituiti da una sequenza di letture non bloccanti
  - memorizzare il numero di bytes che si devono leggere in totale.

# ASSIGNMENT 7: NIO ECHO SERVER

- scrivere un programma echo server usando la libreria java NIO e, in particolare, il Selector e canali in modalità non bloccante, e un programma echo client, usando NIO (va bene anche con modalità bloccante).
- Il server accetta richieste di connessioni dai client, riceve messaggi inviati dai client e li rispedisce (eventualmente aggiungendo "echoed by server" al messaggio ricevuto).
- Il client legge il messaggio da inviare da console, lo invia al server e visualizza quanto ricevuto dal server.



# Reti e Laboratorio III

## Modulo Laboratorio III

**AA. 2022-2023**

**docente: Laura Ricci**

**[laura.ricci@unipi.it](mailto:laura.ricci@unipi.it)**

**Lezione 10**

**UDP**

**DATAGRAMPACKET E  
DATAGRAMSOCKET**

**24/11/2022**

# TCP ED UDP: CONFRONTO

- in certi casi TCP offre “più di quanto necessario”
  - non interessa garantire che tutti i messaggi vengano recapitati
  - si vuole evitare l'overhead dovuto alla ritrasmissione dei messaggi
  - non è necessario leggere i dati nell'ordine con cui sono stati spediti
- UDP supporta una comunicazione connectionless e fornisce un insieme molto limitato di servizi, rispetto a TCP
  - aggiunge un ulteriore livello di indirizzamento a quello offerto dal livello IP, quello delle porte.
  - offre un servizio di scarto dei pacchetti corrotti.
- uno slogan per UDP:

*“Timely, rather than orderly and reliable delivery”*



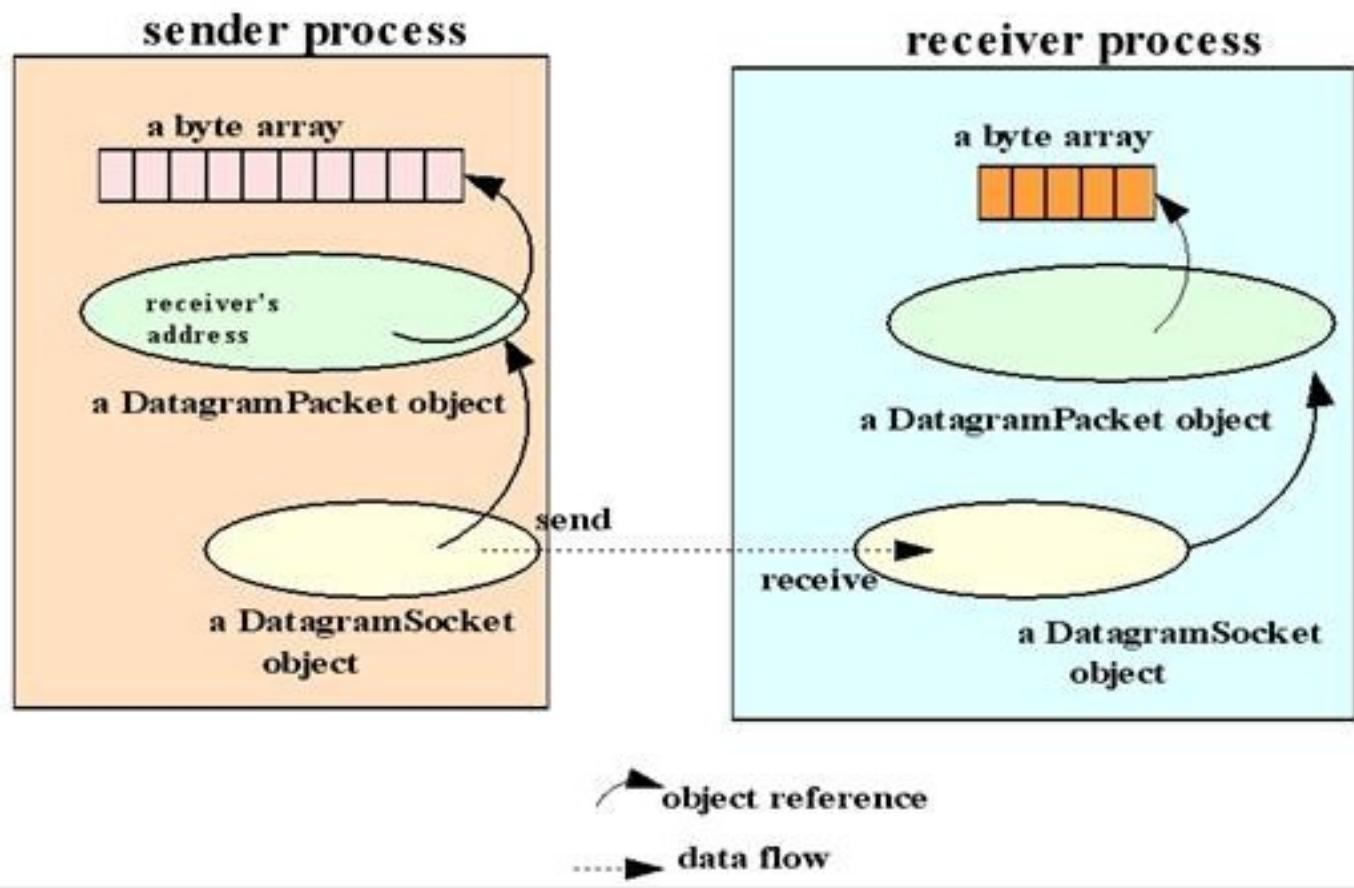
# UDP: QUANDO USARLO?

- stream video/audio: meglio perdere un frame che introdurre overhead nella trasmissione di ogni frame
- tutti gli host di un ufficio inviano, ad intervalli di tempo brevi e regolari, un keep-alive ad un server centrale
  - la perdita di un keep alive non è importante
  - non è importante che il messaggio spedito alle 10:05 arrivi prima di quello spedito alle 10:07
- compravendita di azioni, le variazioni di prezzo tracciate in uno “stock ticker”
  - la perdita di una variazione di prezzo può essere tollerata per titoli azionari di minore importanza
  - il prezzo deve essere controllato al momento della compra/vendita
- alcuni servizi su UDP: DNS, prime versioni di NFS, TFTP (retrivial file transfer protocol), alcuni protocolli peer-to-peer

# CONNECTION ORIENTED VS. CONNECTIONLESS

- JAVA socket API: interfacce diverse per UDP e TCP
- TCP: Stream Sockets
  - “apertura di una connessione”, collegare il client socket al server
- UDP: Datagram Sockets
  - non è richiesto un collegamento prima di inviare una lettera
  - piuttosto è necessario specificare l'indirizzo del destinatario per ogni lettera spedita
  - lettera = pacchetto
    - ogni pacchetto, chiamato **DatagramPacket**, è indipendente dagli altri e porta l'informazione per il suo instradamento

# UDP IN JAVA



# IL SERVIZIO DAYTIME IN UDP

- svilupperemo un daytime client
- il client si collega ad un server noto che offre sulla porta nota il servizio daytime: client in JAVA, server su porta nota
- specifica del servizio: RFC 867
- successivamente svilupperemo il DayTimeServer, un server scritto in JAVA per il servizio di DayTime



# DAYTIME UDP CLIENT: “HOW TO DO”

1. aprire il socket: se si sceglie la porta 0 il sistema sceglie una porta libera “effimera”

```
DatagramSocket socket = new DatagramSocket(0);
```

2. impostare un timeout sul socket, opzionale, ma consigliato

```
setSoTimeout(15000)
```

3. costruire due pacchetti: uno per inviare la richiesta al server, uno per ricevere la risposta

```
InetAddress host = InetAddress.getByName(HOSTNAME);  
DatagramPacket request = new DatagramPacket(new byte[1], 1, host , PORT);  
byte [] data = new byte[1024];  
DatagramPacket response = new DatagramPacket(data, data.length);
```

4. mandare la richiesta ed aspettare la risposta

```
socket.send(request);  
socket.receive(response);
```

5. estrarre i byte dalla risposta e convertirli in String

```
String daytime = new String(response.getData(),0,response.getLength(),"Us-ASCII");  
System.out.println(daytime);
```



# DAYTIME CLIENT

```
import java.io.*;
import java.net.*;
public class DayTimeUDPClient {
    // RFC-867
    private final static int PORT = 13;
    private static final String HOSTNAME = "test.rebex.net";
    public static void main(String[] args) {
        try (DatagramSocket socket = new DatagramSocket(0)) {
            socket.setSoTimeout(15000);
            InetAddress host = InetAddress.getByName(HOSTNAME);
            DatagramPacket request = new DatagramPacket(new byte[1], 1, host, PORT);
            DatagramPacket response = new DatagramPacket(new byte[1024], 1024);
            socket.send(request);
            socket.receive(response);
            String daytime = new String(response.getData(), 0, response.getLength(), "Us-ASCII");
            System.out.println(daytime);
        }
        catch (IOException ex) { ex.printStackTrace(); }}}
try with resources
```

```
$Java DayTimeUDPClient
Wed, 23 Nov 2022 22:37:43 GMT
```



# DAYTIME UDP SERVER: “HOW TO DO”

## I. aprire un DatagramSocket su una porta “nota” (“well known port”)

```
DatagramSocket socket = new DatagramSocket(13);
```

- porta nota perchè i client devono inviare i packet a quella destinazione
- a differenza di TCP, stesso tipo di socket per il client e per il server

## 2. creare un pacchetto in cui ricevere la richiesta del client

```
DatagramPacket request = new DatagramPacket(new byte[1024], 1024);  
socket.receive(request);
```

## 3. creare un pacchetto di risposta

```
String daytime = new Date().toString();  
byte[] data = daytime.getBytes("US-ASCII");  
InetAddress host = request.getAddress();  
int port = request.getPort();  
DatagramPacket response = new DatagramPacket(data, data.length, host, port)
```

## 4. inviare la risposta usando lo stesso socket da cui si è ricevuto il pacchetto

```
socket.send(response);
```



# DAYTIME UDP SERVER

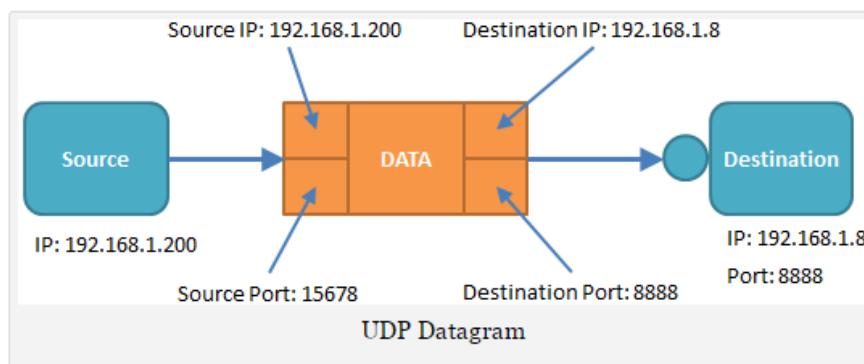
```
import java.net.*; import java.util.Date; import java.io.*;  
public class DatTimeUDPServer {  
    private final static int PORT = 13;  
    public static void main(String[] args) {  
        try (DatagramSocket socket = new DatagramSocket(PORT)) {  
            while (true) {  
                try {  
                    DatagramPacket request = new DatagramPacket(new byte[1024], 1024);  
                    socket.receive(request);  
                    System.out.println("ricevuto un pacchetto da"+request.getAddress()+"  
                                         "+request.getPort());  
                    String daytime = new Date().toString();  
                    byte[] data = daytime.getBytes("US-ASCII");  
                    DatagramPacket response = new DatagramPacket(data, data.length,  
                                                               request.getAddress(), request.getPort());  
                    socket.send(response);  
                } catch (IOException | RuntimeException ex) {ex.printStackTrace();}  
            }  
        } catch (IOException ex) { ex.printStackTrace();}}}
```



# IL DATAGRAM UDP

- dimensione massima teorica di un pacchetto: 65597 bytes
  - IP header= 20 bytes, UDP header=8 bytes
  - molte piattaforme limitano la dimensione massima a 8192 bytes
- in JAVA un datagram UDP è rappresentato come una istanza della classe DatagramPacket

UDP Datagram Header Format								
Bit #	0	7	8	15	16	23	31	
0	Source Port				Destination Port			
32	Length				Header and Data Checksum			



- il mittente deve inizializzare
  - il campo DATA
  - destination IP e destination port
- source IP inserito automaticamente
- source port può essere effimera

# DATAGRAMPACKET: 6 COSTRUTTORI

- 2 costruttori per ricevere i dati

```
public DatagramPacket(byte[ ] buffer, int length)
```

```
public DatagramPacket(byte[ ] buffer, int offset, int length)
```

- 4 costruttori per inviare dati

```
public DatagramPacket(byte[ ] buffer, int length,  
                      InetAddress remoteAddr, int remotePort)
```

```
public DatagramPacket(byte[ ] buffer, int offset, int length,  
                      InetAddress remoteAddr, int remotePort)
```

```
public DatagramPacket(byte[ ] buffer, int length, SocketAddress destination)
```

```
public DatagramPacket(byte[ ] buffer, int offset, int length,  
                      SocketAddress destination)
```

- in ogni caso un riferimento ad un vettore di byte buffer che contiene i dati da spedire oppure quelli ricevuti
- eventuali informazioni di addressing, se il DatagramPacket deve essere spedito



# RICEVERE DATI: COSTRUZIONE DATAGRAM

```
public DatagramPacket (byte[ ] buffer, int length)
```

- definisce la struttura utilizzata per memorizzare il pacchetto ricevuto.
- il buffer viene passato vuoto alla receive che lo riempie con il payload del pacchetto ricevuto
- se settato l'offset, la copia avviene a partire dalla posizione indicata
- il parametro length
  - indica il numero massimo di bytes che possono essere copiati nel buffer
  - deve essere minore di buffer.length, altrimenti viene sollevata eccezione
- la copia del payload termina quando
  - l'intero pacchetto è stato copiato
  - oppure quando length bytes sono stati copiati, se il payload è più grande
  - getLength restituisce il numero di bytes effettivamente copiati



# INVIARE DATI: COSTRUZIONE DEL DATAGRAM

```
public DatagramPacket(byte[] buffer, int length, InetAddress destination, int port)
```

- definisce il `DatagramPacket` da inviare
- `length` indica il numero di bytes che devono essere copiati dal byte buffer nel pacchetto, a partire dal byte 0 o da offset
  - solleva un'eccezione se `length` è maggiore di `buffer.length`
  - se il byte buffer contiene più di `length` bytes, questi non vengono copiati
- `destination + port` individuano il destinatario
- molti altri costruttori sono disponibili
- notare che, per essere memorizzato nel buffer, il messaggio deve essere trasformato in una sequenza di bytes. Per generare vettori di bytes:
  - il metodo `getBytes()`
  - la classe `java.io.ByteArrayOutputStream`



# LA CLASSE DATAGRAM SOCKETS: COSTRUTTORI

- DatagramSocket() crea un socket e lo collega ad una qualsiasi porta libera disponibile sull'host locale
  - try {DatagramSocket client=new DatagramSocket(); //send packets}
  - utilizzato generalmente lato client, per spedire datagrammi
  - getLocalPort( ) per reperire la porta allocata
  - il server può inviare la risposta, prelevando l'indirizzo del mittente (IP+porta) dal pacchetto ricevuto
- DatagramSocket(int p) crea un socket e lo collega alla porta specificata, sull'host locale
  - il server crea un socket collegato ad una porta che rende nota ai clients.
  - la porta è allocata a quel servizio (porta non effimera)
  - solleva un'eccezione quando la porta è già utilizzata, oppure se non si hanno i diritti

# LA CLASSE DATAGRAM SOCKETS: COSTRUTTORI

- DatagramSocket: crea un socket e lo collega ad una qualsiasi porta libera disponibile sull'host locale

```
try {  
    DatagramSocket client = new DatagramSocket();  
    //send packets  
}
```

- utilizzato generalmente lato client, per spedire datagrammi
- per reperire la porta allocata utilizzare il metodo `getLocalPort()`
- esempio:
  - un client si connette ad un server mediante un socket collegato ad una porta anonima.
  - il server preleva l'indirizzo del mittente (IP+porta) dal pacchetto ricevuto e può così inviare una risposta.
  - quando il socket viene chiuso, la porta viene utilizzata per altre connessioni.

# DECIDERE LA DIMENSIONE DEL DATAGRAMPACKET

- ad ogni socket sono associati due buffers: uno per la ricezione ed uno per la spedizione, gestiti dal sistema operativo, non dalla JVM

```
import java.net.*;  
  
public class udpproof {  
    public static void main (String args[]) throws Exception  
    {DatagramSocket dgs = new DatagramSocket( );  
        int r = dgs.getReceiveBufferSize();  
        int s = dgs.getSendBufferSize();  
        System.out.println("receive buffer"+r);  
        System.out.println("send buffer"+s); } }
```

- stampa prodotta : **receive buffer 8192 send buffer 8192**
- in generale la dimensione massima di un pacchetto UDP è 64k bytes, ma in molte piattaforme è 8k
- pacchetti più grandi vengono in generale troncati
- safety: DatagramPacket minori di 512 bytes

# I METODI SET

**void setData(byte[ ] buffer)**

- setta il payload di “this” packet, prendendo i dati dal buffer

**void setData(byte[ ] buffer, int offset, int length)**

- setta il payload di “this” packet, prendendo dati da una parte del buffer
- utile quando si deve mandare una grande quantità di dati

```
int offset = 0;  
DatagramPacket dp = new DatagramPacket(bigarray, offset, 512);  
int bytesSent = 0;  
while (bytesSent < bigarray.length) {  
    socket.send(dp);  
    bytesSent += dp.getLength();  
    int bytesToSend = bigarray.length - bytesSent;  
    int size = (bytesToSend > 512) ? 512 : bytesToSend;  
    dp.setData(bigarray, bytesSent, size);
```

**void setPort(int iport)**

- setta la porta nel datagram



**void** setLength(**int** length)

- setta la lunghezza del payload del Datagram

**void** setAddress(**InetAddress** iaddr)

- setta l'**InetAddress** della macchina a cui il payload è diretto
- utile quando si deve mandare lo stesso Datagram a più destinatari

```
DatagramSocket socket= new DatagramSocket();
String s = "Really important message";
byte [] data = s.getBytes("UTF-8");
DatagramPacket dp = new DatagramPacket(data, data.length);
dp.setPort(2000);
String network = "128.238.5.";
for (int host =1; host <255; host++)
{InetAddress remote = InetAddress.getByName(network+host);
 dp.setAddress(remote);
socket.send(dp);
System.out.println("sent");}
```

**void** setSocketAddress(SocketAddress addr)

- utile per inviare risposte

```
DatagramPacket input = new DatagramPacket(new byte[8192], 8192);
socket.receive(input);

DatagramPacket output = new DatagramPacket ("Hello
                                             here".getBytes("UTF-8"),11);
SocketAddress address = input.getSocketAddress();
output.setSocketAddress(address);
socket.send(output);
```

# I METODI GET

`InetAddress getAddress()`

- restituisce l'indirizzo IP della macchina a cui il Datagram è stato inviato oppure della macchina da cui è stato spedito

`int getPort( )`

- restituisce il numero di porta sull'host remoto a cui il Datagram è stato inviato, oppure della macchina da cui è stato spedito

`byte[] getData()`

- restituisce un byte array contenente i dati del buffer associato al Datagram
- ignora offset e lunghezza

`int [] getLength(), int [] getOffset()`

- restituiscono la lunghezza/offset del Datagram da inviare o da ricevere

`SocketAddress getSocketAddress()`

- restituisce (IP+numero di porta) del Datagram sull'host remoto cui il Datagram è stato inviato, o dell'host a cui è stato spedito

# INVIARE E RICEVERE DATAGRAM

invio di pacchetti

sock.*send* (dp)

- `sock` è il socket attraverso il quale voglio spedire il Datagram `dp`

ricezione di pacchetti sock.*receive*(dp)

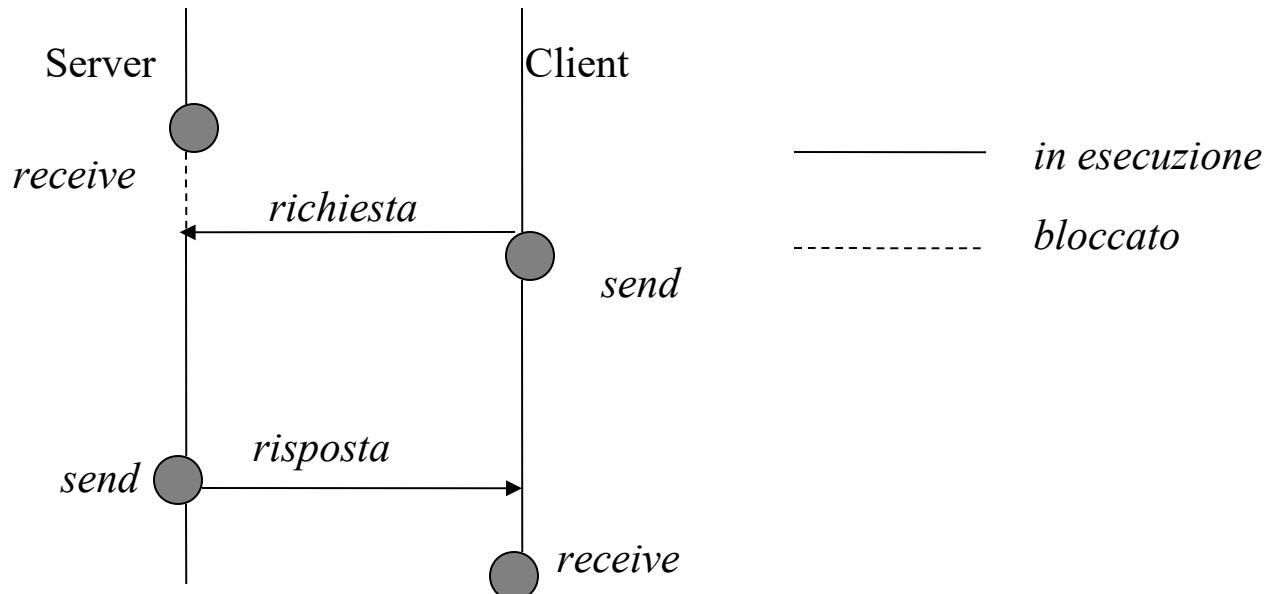
- `sock` è il socket attraverso il quale ricevo il Datagram `dp`
- riceve un Datagram dal socket
- riempie il buffer associato al socket con i dati ricevuti
- Il Datagram ricevuto contiene anche indirizzo IP e porta del mittente

# COMUNICAZIONE UDP: CARATTERISTICHE

send non bloccante nel senso che il processo che esegue la send prosegue la sua esecuzione, senza attendere che il destinatario abbia ricevuto il pacchetto

receive bloccante il processo che esegue la receive si blocca fino al momento in cui viene ricevuto un pacchetto.

per evitare attese indefinite è possibile associare al socket un timeout. Quando il timeout scade, viene sollevata una `InterruptedException`



# GESTIONE BUFFER RICEZIONE

```
import java.net.*;  
  
public class Sender  
{ public static void main (String args[]) {  
    try  
    {DatagramSocket clientsocket = new DatagramSocket();  
    byte[] buffer="1234567890abcdefghijklmnopqrstuvwxyz".getBytes("US-  
    ASCII");  
    InetAddress address = InetAddress.getByName("localhost");  
    for (int i = buffer.length; i >0; i--) {  
        DatagramPacket mypacket = new DatagramPacket(buffer,i,address,  
                                                       40000);  
        clientSocket.send(mypacket);  
        Thread.sleep(200); }  
    System.exit(0);}  
    catch (Exception e) {e.printStackTrace();}}}
```



# DATI INVIATI

**Dati inviati dal mittente:**

Length 36 data 1234567890abcdefghijklmnopqrstuvwxyz

Length 35 data 1234567890abcdefghijklmnopqrstuvwxyz

Length 34 data 1234567890abcdefghijklmnopqrstuvwxyz

Length 33 data 1234567890abcdefghijklmnopqrstuvwxyz

Length 32 data 1234567890abcdefghijklmnopqrstuvwxyz

Length 31 data 1234567890abcdefghijklmnopqrstuvwxyz

.....

Length 5 data 12345

Length 4 data 1234

Length 3 data 123

Length 2 data 12

Length 1 data 1



# GESTIONE BUFFER RICEZIONE

```
import java.net.*;
public class Receiver
{public static void main(String args[]) throws Exception {
    DatagramSocket serverSock= new DatagramSocket(40000);
    byte[] buffer = new byte[100];
    DatagramPacket receivedPacket = new DatagramPacket(buffer,
                                                       buffer.length);
    while (true) {
        serverSock.receive(receivedPacket);
        String byteToString = new String(receivedPacket.getData(),"US-
                                         ASCII");
        int l=byteToString.length();
        System.out.println(l);
        System.out.println("Length " + receivedPacket.getLength() +
                           " data " + byteToString);}}}
```



# DATI RICEVUTI

100

Length 36 data 1234567890abcdefghijklmnopqrstuvwxyz

100

Length 35 data 1234567890abcdefghijklmnopqrstuvwxyz

100

Length 34 data 1234567890abcdefghijklmnopqrstuvwxyz

100

Length 33 data 1234567890abcdefghijklmnopqrstuvwxyz

....

100

Length 2 data 1234567890abcdefghijklmnopqrstuvwxyz

100

Length 1 data 1234567890abcdefghijklmnopqrstuvwxyz

# GESTIONE DATI RICEZIONE

- per ricevere correttamente i dati  
individuare i dati disponibili specificando offset e lunghezza

```
String byteToString =  
  
    new String(receivedPacket.getData(),0,receivedPacket.getLength(),"US-ASCII")
```

- sempre specificare lunghezza ed offset dei dati ricevuti, anche se si utilizzano stream (vedi slide successive)



# RECEIVE CON TIMEOUT

- `SO_TIMEOUT` proprietà associata al socket, indica l'intervallo di tempo, in millisecondi, di attesa di ogni receive eseguita su quel socket
- nel caso in cui l'intervallo di tempo scada prima che venga ricevuto un pacchetto dal socket, viene sollevata una eccezione di tipo `InterruptedException`
- metodi per la gestione di time out

`public synchronized void setSoTimeout(int timeout) throws  
SocketException`

esempio: se ds è un datagram socket,

```
ds.setSoTimeout(30000)
```

associa un timeout di 30 secondi al socket ds.

# COSTRUZIONE/LETTURA DI VETTORI DI BYTES

- i dati inviati mediante UDP devono essere rappresentati come vettori di bytes
- alcuni metodi per la conversione stringhe/vettori di bytes
  - `Byte[] getBytes()`
    - applicato ad un oggetto String
    - restituisce una sequenza di bytes che codificano i caratteri della stringa usando la codifica di default dell'host e li memorizza nel vettore
  - `String (byte[] bytes, int offset, int length)`
    - costruisce un nuovo oggetto di tipo String prelavando length bytes dal vettore bytes, a partire dalla posizione offset
- altri meccanismi per generare pacchetti a partire da dati strutturati:
  - utilizzare i **filtri** per generare streams di bytes a partire da dati strutturati/ad alto livello



# DATI STRUTTUATI IN PACCHETTI UDP

**public** `ByteArrayOutputStream ()`

**public** `ByteArrayOutputStream (int size)`

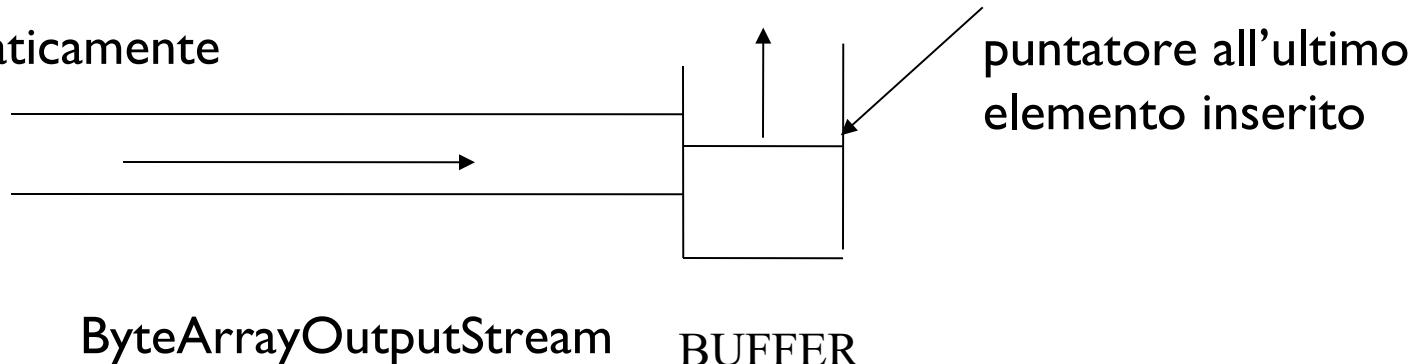
- gli oggetti istanze di questa classe rappresentano stream di bytes
- ogni dato scritto sullo stream viene riportato in un **buffer** di memoria a **dimensione variabile** (dimensione di default = 32 bytes).

**protected byte** buf []

**protected int** count

**count** indica quanti sono i bytes memorizzati in buf

- quando il buffer si riempie la sua dimensione viene **raddoppiata automaticamente**



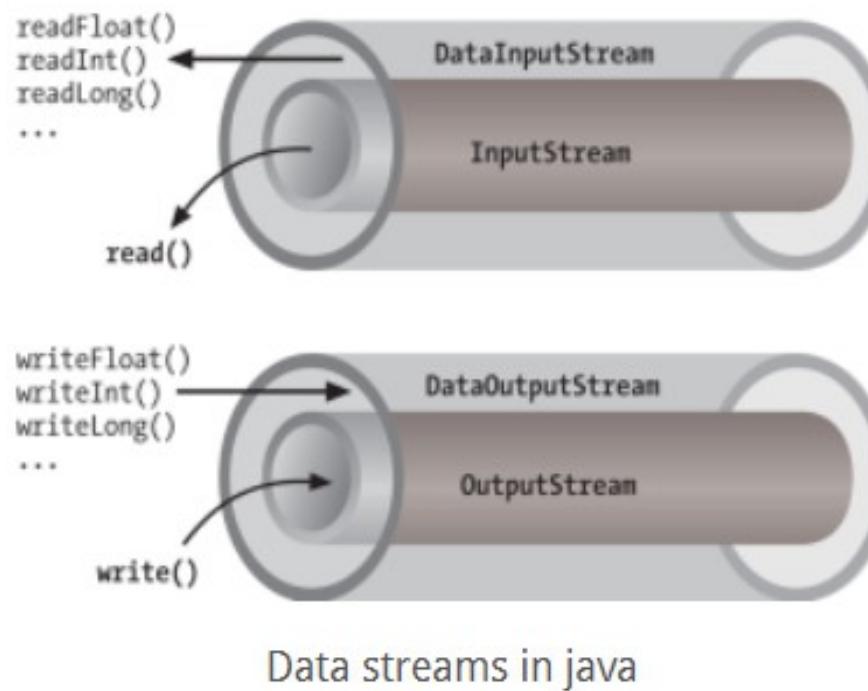
ByteOutputStream      BUFFER

# UDP E DATA STREAM

- è possibile collegare ad un `ByteArrayOutputStream` un altro filtro

```
ByteArrayOutputStream baos= new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos)
```

- `DataOutput/InputStream` consente di scrivere dati primitivi sullo stream, la trasformazione in bytes è effettuata automaticamente

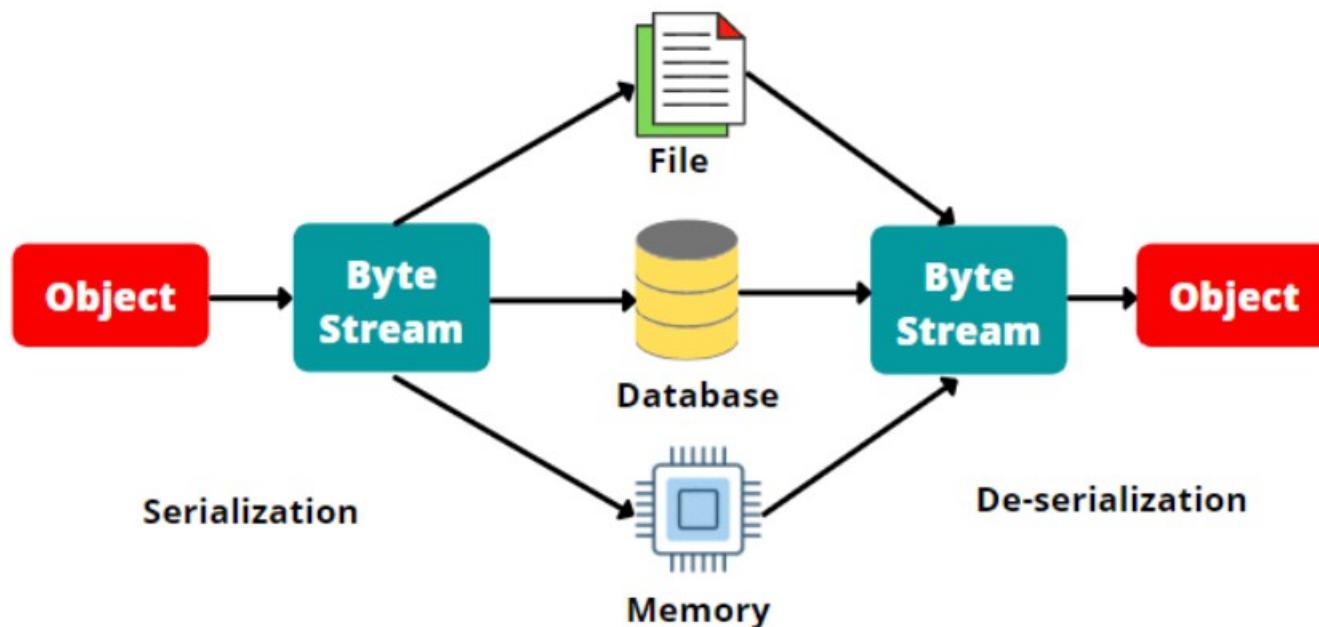


# UDP E SERIALIZZAZIONE

- inviare oggetti in pacchetti?
- usare la serializzazione per generare uno stream di Byte
- collegare l'outputstream generato ad un `ByteArrayOutputStream`

```
ByteArrayOutputStream baos= new ByteArrayOutputStream();
```

```
ObjectOutputStream dos = new ObjectOutputStream(baos)
```



# BYTE ARRAY OUTPUT STREAMS

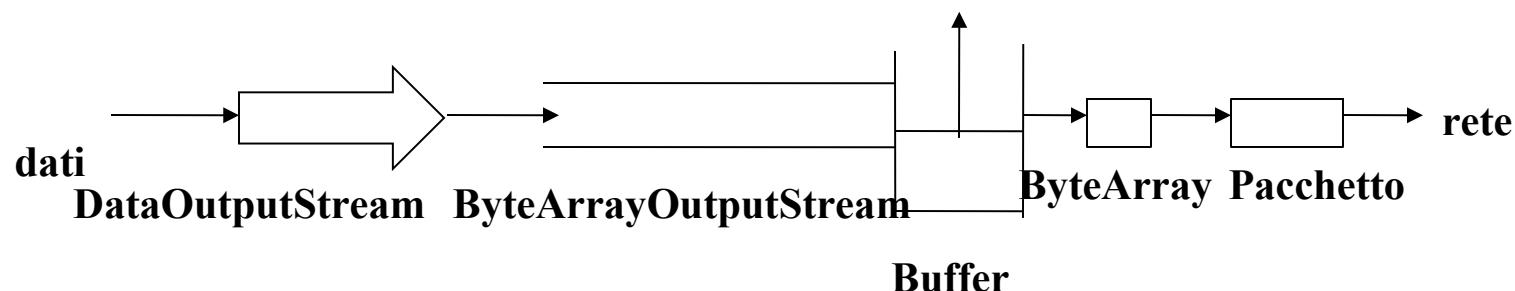
- ad un `ByteArrayOutputStream` può essere collegato un altro filtro

```
ByteArrayOutputStream baos= new ByteArrayOutputStream( );  
DataOutputStream dos = new DataOutputStream(baos)
```

- i dati presenti nel buffer B associato ad un `ByteArrayOuputStream` baos possono essere copiati in un array di bytes

```
byte [ ] barr = baos.toByteArray( )
```

Flusso dei dati:

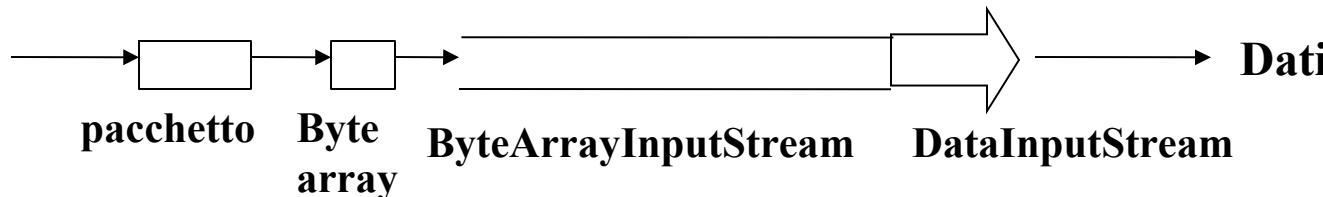


# BYTE ARRAY INPUT STREAMS

```
public ByteArrayInputStream ( byte [ ] buf )
public ByteArrayInputStream ( byte [ ] buf, int offset,
                             int length )
```

- creano stream di byte a partire dai dati contenuti nel vettore di byte buf.
- il secondo costruttore copia length bytes iniziando alla posizione offset.
- è possibile concatenare un **DataInputStream**

Flusso dei dati:



# LA CLASSE BYTEARRAYOUTPUTSTREAM

metodi per la gestione dello stream:

- **public int size( )** restituisce count, cioè il numero di bytes memorizzati nello stream (non la lunghezza del vettore buf!)
- **public synchronized void reset( )** svuota il buffer, assegnando 0 a count. tutti i dati precedentemente scritti vengono eliminati.  
`baos.reset ( )`
- **public synchronized byte toByteArray ( )** restituisce un vettore in cui sono stati copiati tutti i bytes presenti nello stream.
  - non modifica count
  - il metodo **toByteArray** non svuota il buffer.



# BYTE ARRAY INPUT/OUTPUT STREAMS

Ipotesi semplificativa: non consideriamo perdita/riordinamento di pacchetti:

```
import java.io.*;
import java.net.*;
public class multidatastreamsender{
    public static void main(String args[ ]) throws Exception
    { // fase di inizializzazione
        InetAddress ia=InetAddress.getByName("localhost");
        int port=13350;
        DatagramSocket ds= new DatagramSocket();
        ByteArrayOutputStream bout= new ByteArrayOutputStream();
        DataOutputStream dout = new DataOutputStream (bout);
        byte [ ] data = new byte [20];
        DatagramPacket dp= new DatagramPacket(data,data.length, ia , port);
```



# BYTE ARRAY INPUT/OUTPUT STREAMS

```
for (int i=0; i< 10; i++)
{dout.writeInt(i);
data = bout.toByteArray();
dp.setData(data,0,data.length);
dp.setLength(data.length);
ds.send(dp);
bout.reset( );
dout.writeUTF("/**");
data = bout.toByteArray( );
dp.setData (data,0,data.length);
dp.setLength (data.length);
ds.send (dp);
bout.reset( )} }
```

# BYTE ARRAY INPUT/OUTPUT STREAMS

Ipotesi semplificativa: non consideriamo perdita/riordinamento di pacchetti

```
import java.io.*;
import java.net.*;

public class multidatastreamreceiver
{public static void main(String args[ ]) throws Exception
{ // fase di inizializzazione
    FileOutputStream fw = new FileOutputStream("text.txt");
    DataOutputStream dr = new DataOutputStream(fw);
    int port =13350;
    DatagramSocket ds = new DatagramSocket (port);
    byte [ ] buffer = new byte [200];
    DatagramPacket dp= new DatagramPacket
        (buffer, buffer.length);
```



# BYTE ARRAY INPUT/OUTPUT STREAMS

```
for (int i=0; i<10; i++)
{ds.receive(dp);
 ByteArrayInputStream bin= new ByteArrayInputStream
 (dp.getData(),0,dp.getLength());
DataInputStream ddis= new DataInputStream(bin);
int x = ddis.readInt();
dr.writeInt(x);
System.out.println(x);
ds.receive(dp);
bin= new ByteArrayInputStream(dp.getData(),0,dp.getLength());
ddis= new DataInputStream(bin);
String y=ddis.readUTF( );
System.out.println(y); }}}
```



# BYTE ARRAY INPUT/OUTPUT STREAMS

- nel programma precedente, la corrispondenza tra la **scrittura** nel mittente e la **lettura** nel destinatario potrebbe non essere più corretta
- esempio:
  - il mittente alterna la spedizione di pacchetti contenenti valori interi con pacchetti contenenti stringhe
  - il destinatario alterna la lettura di interi e di stringhe
  - ma se un pacchetto viene perso: il destinatario scritture/letture possono non corrispondere
- realizzazione di UDP affidabile: utilizzo di ack per confermare la ricezione + identificatori unici

# CONCLUSIONI: STREAM ARE EVERYWHERE!

- trasmissione connection oriented: una connessione viene modellata con uno stream.

invio di dati                scrittura sullo stream

ricezione di dati    lettura dallo stream

- trasmissione connectionless: stream utilizzati per la generazione dei pacchetti:

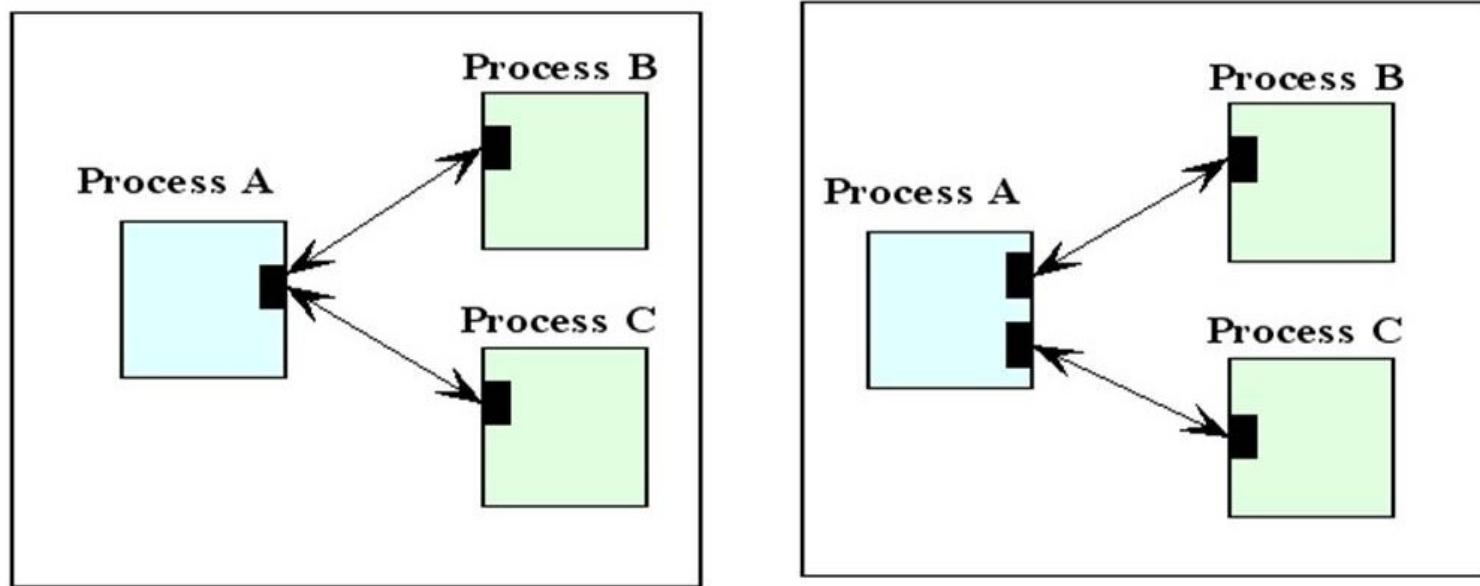
**ByteArrayOutputStream**, consentono la conversione di uno stream di bytes in un vettori di bytes da spedire con i pacchetti UDP

**ByteArrayInputStream**, converte un vettore di bytes in uno stream di byte.



# CONCLUSIONI: SOCKET UDP

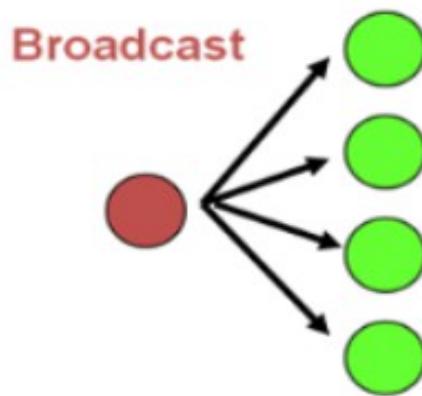
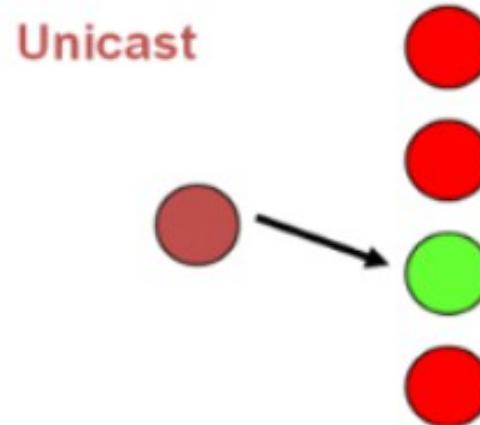
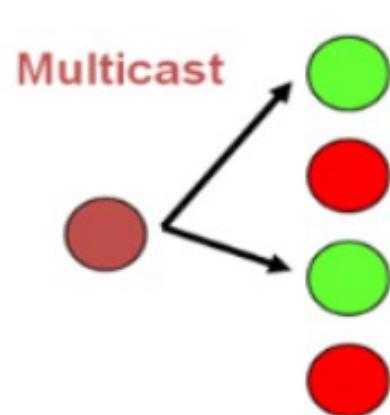
- possibile usare lo stesso socket per spedire pacchetti verso destinatari diversi
- processi (applicazioni) diverse possono spedire pacchetti sullo stesso socket in questo caso l'ordine di arrivo dei massaggi è non deterministico, in accordo con il protocollo UDP
- ...ma è possibile anche utilizzare socket diversi per comunicazioni diverse



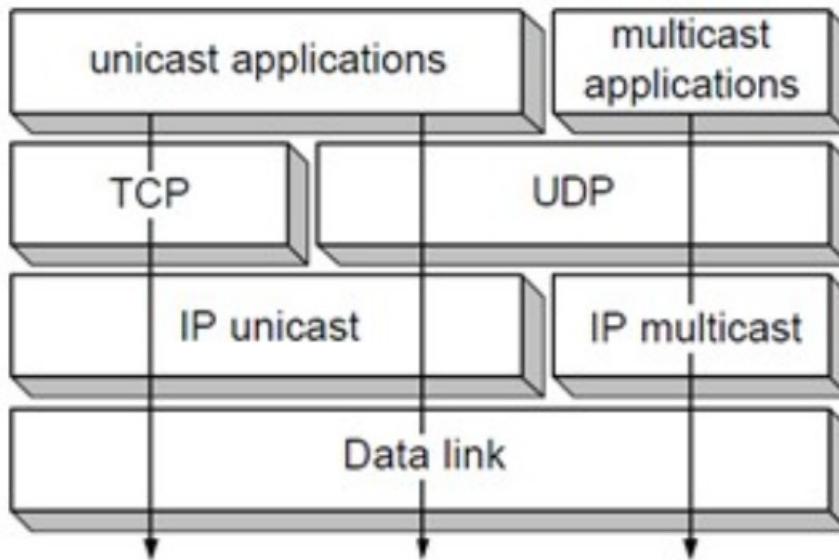
# MULTICASTING

- inviare dati da un host ad un insieme di altri nodi
  - non tutti gli host della rete: solo quelli che hanno espresso interesse ad unirsi ad un gruppo di multicast
- esempi:
  - diverse applicazioni usano IP multicasting per notificare/scoprire dinamicamente i servizi in una rete, senza usare DNS o terze parti
- la maggior parte del lavoro viene svolto dai router ed è trasparente al programmatore
  - i router assicurano che il pacchetto spedito dal mittente sia consegnato a tutti gli host interessati
  - usa Time-To-Live IP: massimo numero di router che il datagramma può attraversare
  - il problema maggiore è che non tutti i router supportano il multicast

# UNICASTING/MULTICASTING/BROADCASTING

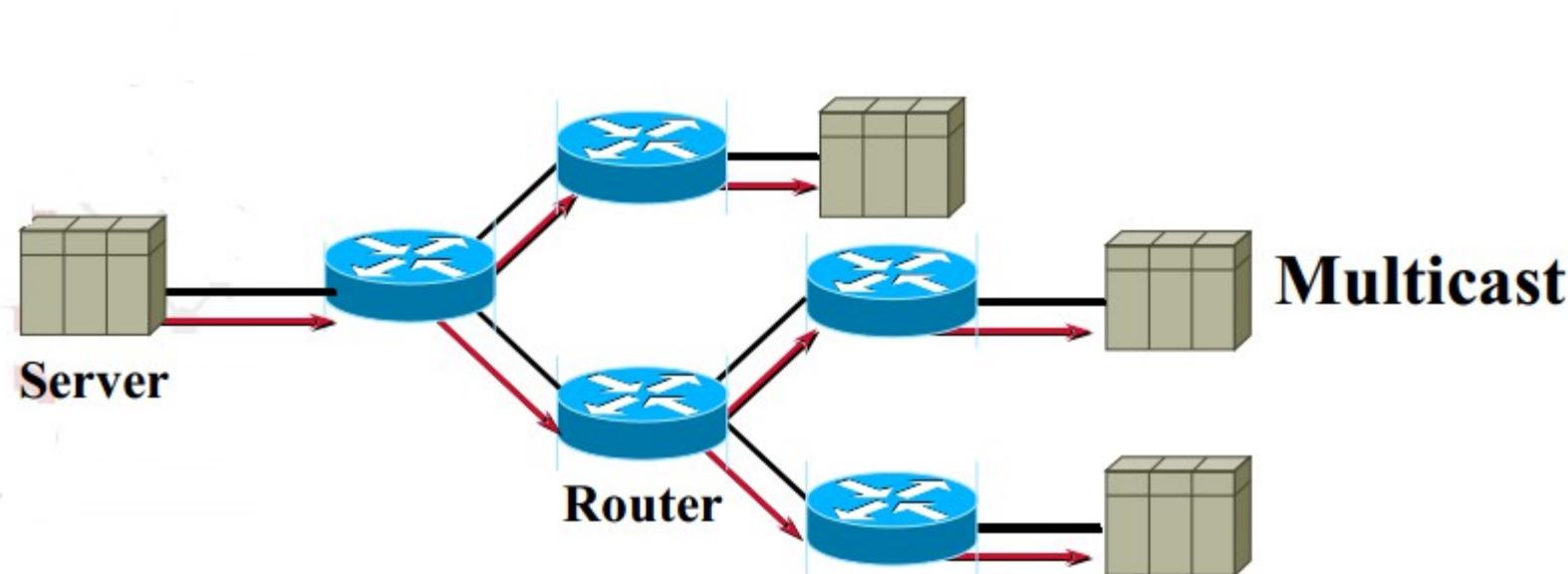
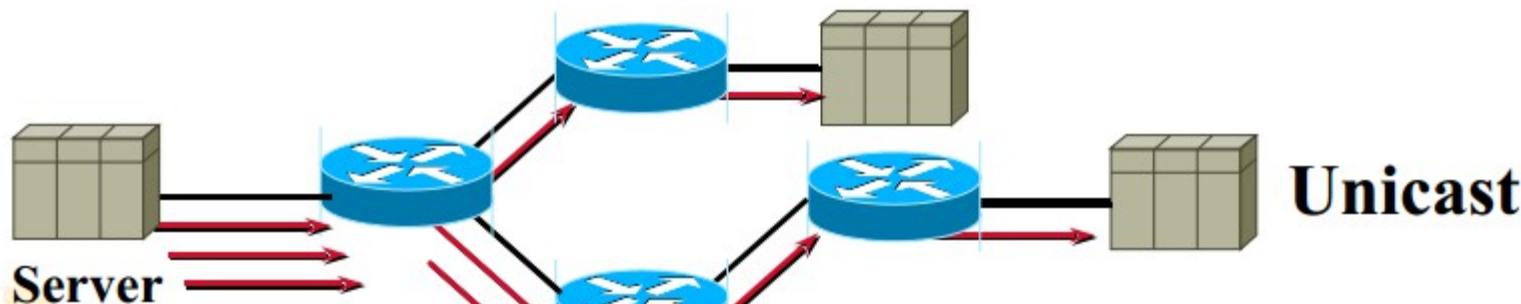


# MULTICAST E PROTOCOL STACK

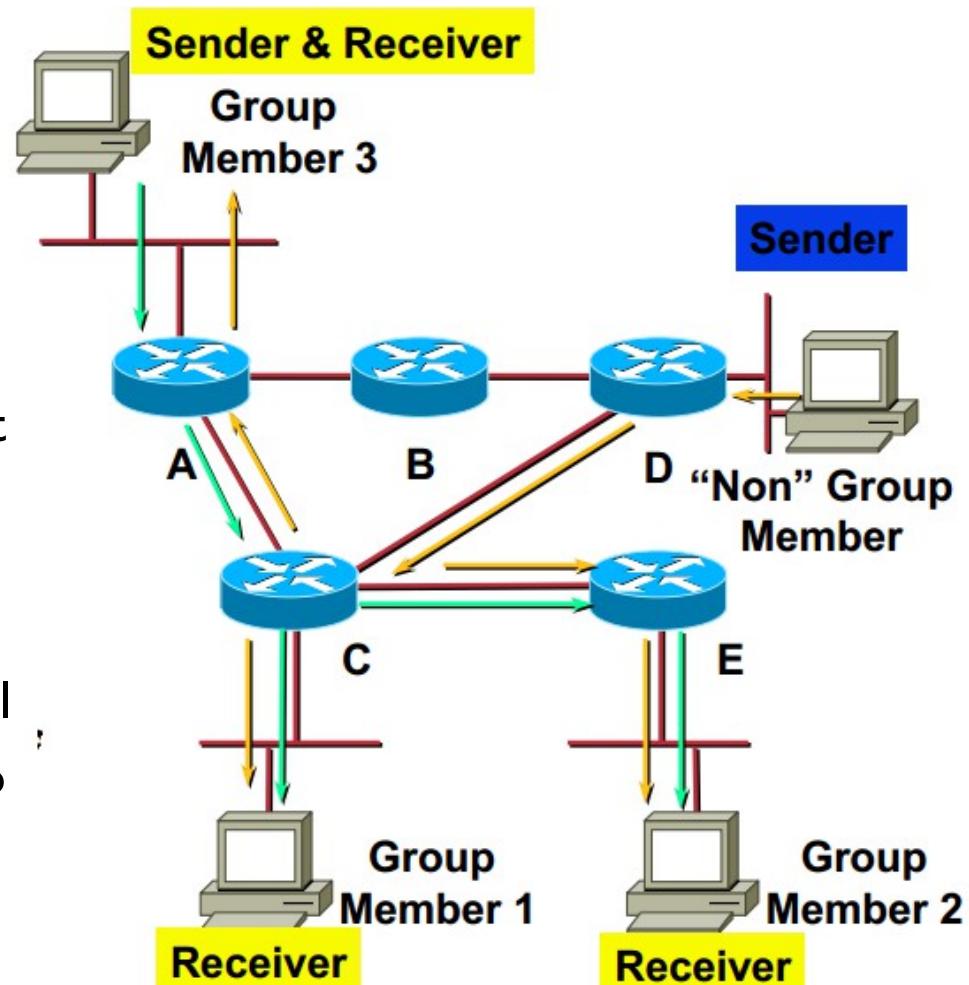


- IP multicasting utilizza UDP
- non esiste multicast TCP!

# UNICAST VERSO MULTICAST



- IP multicast basato sul **concetto di gruppo**
  - insieme di processi in esecuzione su host diversi
- tutti i membri di un gruppo di multicast ricevono un messaggio spedito su quel gruppo
- mentre non occorre essere membri del gruppo per inviare i messaggi su di esso
- gestiti a livello IP dal protocollo IGMP



deve contenere almeno primitive per:

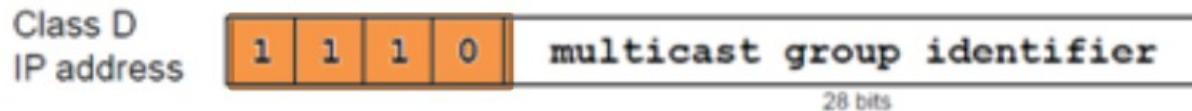
- **unirsi** ad un gruppo di multicast
- **lasciare** un gruppo
- **spedire** messaggi ad un gruppo
  - il messaggio viene recapitato a tutti i processi che fanno parte del gruppo in quel momento
- **ricevere** messaggi indirizzati ad un gruppo

Il supporto deve fornire

- uno **schema di indirizzamento** per identificare univocamente un gruppo.
- un meccanismo che registri la corrispondenza tra un gruppo ed i suoi partecipanti (**IGMP**)
- un meccanismo che ottimizzi l'uso della rete nel caso di invio di pacchetti ad un gruppo di multicast

# SCHEMA DI INDIRIZZAMENTO MULTICAST

- basato sull'idea di riservare un insieme di indirizzi IP per il multicast
- IPv4: indirizzo di un gruppo è un indirizzo in classe D
  - [224.0.0.0 – 239.255.255.255]



i primi 4 bit del primo ottetto = 1110

i restanti bit identificano il particolare gruppo

*IPV6*: tutti gli indirizzi di multicast iniziano con FF o 1111 1111 in binario

# MULTICAST: CARATTERISTICHE

- utilizza il paradigma **connectionless** (UDP):
  - sarebbero richieste  $n \times (n-1)$  connessioni per un gruppo di **n** applicazioni, se si usa la comunicazione **connection oriented**
- comunicazione **connectionless** adatta per il tipo di applicazioni verso cui è orientato il multicast
  - trasmissione di dati video/audio: invio dei frames di una animazione.
  - è più accettabile la **perdita occasionale** di un frame piuttosto che un **ritardo costante** tra la spedizione di due frames successivi
- si perde la affidabilità della trasmissione, ma esistono librerie JAVA non standard che forniscono multicast affidabile
  - garantiscono che il messaggio venga recapitato una sola volta a tutti i processi del gruppo
  - possono garantire altre proprietà relative all'ordinamento con cui i messaggi spediti al gruppo di multicast vengono recapitati ai singoli partecipanti



# SOCKET MULTICAST

## Java.net.MulticastSocket

- estende `DatagramSocket`
- socket su cui ricevere i messaggi da un gruppo di multicast
- effettua overriding dei metodi esistenti in `DatagramSocket` e fornisce nuovi metodi per l'implementazione di funzionalità tipiche del multicast

```
import java.net.*;
import java.io.*;
public class multicast
{public static void main (String [ ] args)
{try
    {MulticastSocket ms = new MulticastSocket( );
    catch (IOException ex) {System.out.println("errore"); }
    }}
```

# UNIRSI AD UN GRUPPO MULTICAST

```
import java.net.*;
import java.io.*;
public class multicast
{public static void main (String [ ] args)
{ try {MulticastSocket ms = new MulticastSocket(4000);
InetSocketAddress
        ia=InettAddress.getByName("226.226.226.226");
ms.joinGroup (ia);
}
catch (IOException ex) {System.out.println("errore"); }}}
```

- `joinGroup` necessaria nel caso si vogliano ricevere messaggi dal gruppo di multicast
- lega il **multicast socket** ad un **gruppo di multicast**: tutti i messaggi ricevuti tramite quel socket provengono da quel gruppo
- `IOException` sollevata se l'indirizzo di multicast è errato

# RICEVERE PACCHETTI DA MULTICAST

```
import java.io.*; import java.net.*;  
  
public class provemulticast {  
  
    public static void main (String args[]) throws Exception  
  
    { byte[] buf = new byte[10];  
  
        InetAddress ia = InetAddress.getByName("228.5.6.7");  
        DatagramPacket dp = new DatagramPacket (buf,buf.length);  
        MulticastSocket ms = new MulticastSocket (4000);  
        ms.joinGroup(ia);  
        ms.receive(dp);    } }
```

- se attivo due istanze di provemulticast sullo stesso host (la reuse socket settata true) non viene sollevata una BindException
- l'eccezione verrebbe invece sollevata se si utilizzasse un DatagramSocket
- servizi diversi in ascolto sulla stessa porta di multicast
- non esiste una corrispondenza biunivoca porta-servizio

# JAVA API PER MULTICAST

- ogni socket multicast ha una proprietà, la **reuse socket**, che se settata a true, dà la possibilità di associare più socket alla stessa porta
- nelle ultime versioni è possibile impostarne il valore

```
try {sock.setReuseAddress(true);}  
catch (SocketException se) {...}
```
- nelle prime versioni di JAVA la proprietà era settata per default a true



# CONCLUSIONI

- TCP: trasmissione vista come uno stream continuo di bytes provenienti dallo stesso mittente
- UDP: trasmissione orientata ai messaggi: “preserves message boundaries”
  - send, riceve DatagramPacket
  - socket come una mailbox: in essa possono essere inseriti messaggi in arrivo da diverse sorgenti (mittenti) o i messaggi inviati a diverse destinazioni
  - ogni ricezione si riferisce ad un singolo messaggio inviato mediante una unica send.
    - dati inviati dalla stessa send non possono essere ricevuti in receive diverse



# ASSIGNMENT: JAVA PINGER

- PING è una utility per la valutazione delle performance della rete utilizzata per verificare la raggiungibilità di un host su una rete IP e per misurare il round trip time (RTT) per i messaggi spediti da un host mittente verso un host destinazione.
- lo scopo di questo assignment è quello di implementare un server PING ed un corrispondente client PING che consenta al client di misurare il suo RTT verso il server.
- la funzionalità fornita da questi programmi deve essere simile a quella della utility PING disponibile in tutti i moderni sistemi operativi. La differenza fondamentale è che si utilizza UDP per la comunicazione tra client e server, invece del protocollo ICMP (Internet Control Message Protocol).
- inoltre, poichè l'esecuzione dei programmi avverrà su un solo host o sulla rete locale ed in entrambe i casi sia la latenza che la perdita di pacchetti risultano trascurabili, il server deve introdurre un ritardo artificiale ed ignorare alcune richieste per simulare la perdita di pacchetti

# PING CLIENT

- accetta due argomenti da linea di comando: nome e porta del server. Se uno o più argomenti risultano scorretti, il client termina, dopo aver stampato un messaggio di errore del tipo ERR -arg x, dove x è il numero dell'argomento.
- utilizza una comunicazione UDP per comunicare con il server ed invia 10 messaggi al server, con il seguente formato:

**PING seqno timestamp**

in cui seqno è il numero di sequenza del PING (tra 0-9) ed il timestamp (in millisecondi) indica quando il messaggio è stato inviato

- non invia un nuovo PING fino che non ha ricevuto l'eco del PING precedente, oppure è scaduto un timeout.



# PING CLIENT

- stampa ogni messaggio spedito al server ed il RTT del ping oppure un \* se la risposta non è stata ricevuta entro 2 secondi
- dopo che ha ricevuto la decima risposta (o dopo il suo timeout), il client stampa un riassunto simile a quello stampato dal PING UNIX

----- PING Statistics -----

10 packets transmitted, 7 packets received, 30% packet loss  
round-trip (ms) min/avg/max = 63/190.29/290

- il RTT medio è stampato con 2 cifre dopo la virgola

# PING SERVER

- è essenzialmente un echo server: rimanda al mittente qualsiasi dato riceve
- accetta un argomento da linea di comando: la porta, che è quella su cui è attivo il server + un argomento opzionale, il seed, un valore long utilizzato per la generazione di latenze e perdita di pacchetti. Se uno qualunque degli argomenti è scorretto, stampa un messaggio di errore del tipo ERR -arg x, dove x è il numero dell'argomento.
- dopo aver ricevuto un PING, il server determina se ignorare il pacchetto (simulandone la perdita) o effettuarne l'eco. La probabilità di perdita di pacchetti di default è del 25%.
- se decide di effettuare l'eco del PING, il server attende un intervallo di tempo casuale per simulare la latenza di rete
- stampa l'indirizzo IP e la porta del client, il messaggio di PING e l'azione intrapresa dal server in seguito alla sua ricezione (PING non inviato, oppure PING ritardato di x ms).

# PING SERVER

```
java PingServer 10002 123
```

```
128.82.4.244:44229> PING 0 1360792326564 ACTION: delayed 297 ms
128.82.4.244:44229> PING 1 1360792326863 ACTION: delayed 182 ms
128.82.4.244:44229> PING 2 1360792327046 ACTION: delayed 262 ms
128.82.4.244:44229> PING 3 1360792327309 ACTION: delayed 21 ms
128.82.4.244:44229> PING 4 1360792327331 ACTION: delayed 173 ms
128.82.4.244:44229> PING 5 1360792327505 ACTION: delayed 44 ms
128.82.4.244:44229> PING 6 1360792327550 ACTION: delayed 19 ms
128.82.4.244:44229> PING 7 1360792327570 ACTION: not sent
128.82.4.244:44229> PING 8 1360792328571 ACTION: not sent
128.82.4.244:44229> PING 9 1360792329573 ACTION: delayed 262 ms
```

# PING CLIENT

```
java PingClient localhost 10002
```

```
PING 0 1360792326564 RTT: 299 ms
```

```
PING 1 1360792326863 RTT: 183 ms
```

```
PING 2 1360792327046 RTT: 263 ms
```

```
PING 3 1360792327309 RTT: 22 ms
```

```
PING 4 1360792327331 RTT: 174 ms
```

```
PING 5 1360792327505 RTT: 45 ms
```

```
PING 6 1360792327550 RTT: 20 ms
```

```
PING 7 1360792327570 RTT: *
```

```
PING 8 1360792328571 RTT: *
```

```
PING 9 1360792329573 RTT: 263 ms
```

```
---- PING Statistics ----
```

```
10 packets transmitted, 8 packets received, 20% packet loss
```

```
round-trip (ms) min/avg/max = 20/158.62/299
```



# JAVA PINGER

Invocazione corretta client/server:

**java PingClient**

**Usage: java PingClient hostname port**

**java PingServer**

**Usage: java PingServer port [seed]**

Invocazione non corretta client/server:

**java PingClient atria three**

**ERR - arg 2**

**java PingServer abc**

**ERR - arg 1**

# **Reti e Laboratorio III**

## **Modulo Laboratorio III**

**AA. 2022-2023**

**docente: Laura Ricci**

**[laura.ricci@unipi.it](mailto:laura.ricci@unipi.it)**

**Lezione II**  
**Esercitazione**  
**01/12/2022**

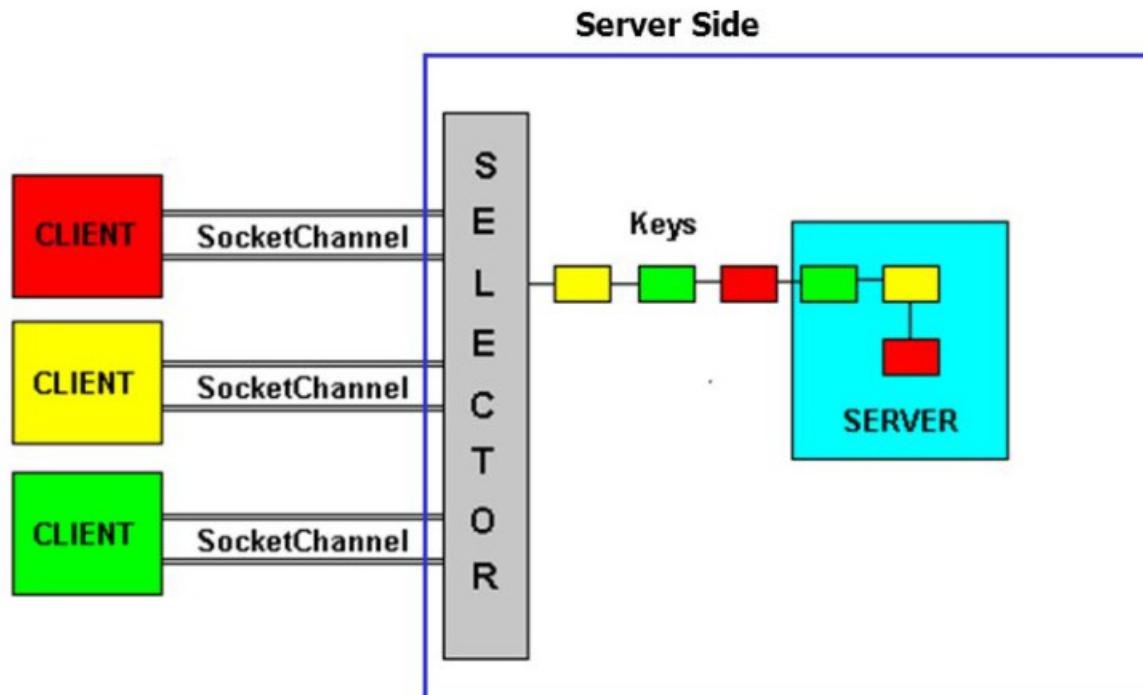
# NIO SELECTOR: PATTERN GENERALE

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
Iterator <SelectionKey> keyIterator = selectedKeys.iterator();
while(keyIterator.hasNext()) {
    SelectionKey key = (SelectionKey) keyIterator.next();
    keyIterator.remove();
    if(key.isAcceptable()) {
        // a connection was accepted by a ServerSocketChannel.
    }
    else if (key.isConnectable()) {
        // a connection was established with a remote server.
    }
    else if (key.isReadable()) {
        // a channel is ready for reading
    }
    else if (key.isWritable()) {
        // a channel is ready for writing }
```



# NIO SELECTOR: PATTERN GENERALE

- iterazione sull'insieme di chiavi che individuano i “canali pronti”
- dalla chiave si può ottenere un riferimento al canale su cui si è verificato l'evento
- keyIterator.remove() deve essere invocata, poiché il Selector non rimuove le chiavi



# SELECTION KEY: L'ATTACHMENT

- attachment: riferimento ad un generico Object
- utile quando si vuole accedere ad informazioni relative al canale (associato ad una chiave) che riguardano il suo stato pregresso
- necessario perchè le operazioni di lettura o scrittura non bloccanti non possono essere considerate atomiche: nessuna assunzione sul numero di bytes letti
- consente di tenere traccia di quanto è stato fatto in una operazione precedente.
  - l'attachment può essere utilizzato per accumulare i byte restituiti da una sequenza di letture non bloccanti
  - memorizzare il numero di bytes che si devono leggere in totale.

- sviluppare un servizio di generazione di una sequenza di interi il cui scopo è testare l'affidabilità della rete, mediante generazione di numeri binari
- quando il server è contattato dal client, esso invia al client una sequenza di interi rappresentati su 4 bytes

0, 1, 2, ...

- il server genera una sequenza infinita di interi
- il client interrompe la comunicazione quando ha ricevuto sufficiente informazioni

```
import java.nio.*;
import java.nio.channels.*;
import java.net.*;
import java.util.*;
import java.io.IOException;
public class IntGenServer {
    public static int DEFAULT_PORT = 1919;
    public static void main(String[] args) {
        int port;
        try {
            port = Integer.parseInt(args[0]);
        }
        catch (RuntimeException ex) {port = DEFAULT_PORT; }
        System.out.println("Listening for connections on port " + port);
    }
}
```

```
ServerSocketChannel serverChannel;  
Selector selector;  
  
try {  
    serverChannel = ServerSocketChannel.open();  
    ServerSocket ss = serverChannel.socket();  
    InetSocketAddress address = new InetSocketAddress(port);  
    ss.bind(address);  
    serverChannel.configureBlocking(false);  
    selector = Selector.open();  
    serverChannel.register(selector, SelectionKey.OP_ACCEPT);  
}  
  
catch (IOException ex) {  
    ex.printStackTrace();  
    return;  
}
```

```
while (true) {  
    try {  
        selector.select();  
    } catch (IOException ex) {  
        ex.printStackTrace();  
        break;  
    }  
    Set <SelectionKey> readyKeys = selector.selectedKeys();  
    Iterator <SelectionKey> iterator = readyKeys.iterator();
```

```
while (iterator.hasNext()) {  
    SelectionKey key = iterator.next();  
    iterator.remove();  
    // rimuove la chiave dal Selected Set, ma non dal Registered Set  
    try {  
        if (key.isAcceptable()) {  
            ServerSocketChannel server = (ServerSocketChannel) key.channel();  
            SocketChannel client = server.accept();  
            System.out.println("Accepted connection from " + client);  
            client.configureBlocking(false);  
            SelectionKey key2 = client.register(selector,SelectionKey.OP_WRITE);  
            ByteBuffer output = ByteBuffer.allocate(4);  
            output.putInt(0);  
            output.flip();  
            key2.attach(output); }  
    }
```

```
else if (key.isWritable())
{ SocketChannel client = (SocketChannel) key.channel();
  ByteBuffer output = (ByteBuffer) key.attachment();
  if (! output.hasRemaining())
  {
    output.rewind();
    int value = output.getInt();
    output.clear();
    output.putInt(value + 1);
    output.flip();
  }
  client.write(output);}
} catch (IOException ex) { key.cancel();
  try { key.channel().close(); }
  catch (IOException cex) {} }}}
```

# NIO INTEGER GENERATION CLIENT

```
import java.nio.*;
import java.nio.channels.*;
import java.net.*;
import java.io.IOException;
public class IntGenClient {

    public static int DEFAULT_PORT = 1919;
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Usage: java IntgenClient host [port]");
            return;
        }
        int port;
        try {
            port = Integer.parseInt(args[1]);
        } catch (RuntimeException ex) {
            port = DEFAULT_PORT;
        }
    }
}
```



# NIO INTEGER GENERATION CLIENT

```
try { SocketAddress address = new InetSocketAddress(args[0], port);
    SocketChannel client = SocketChannel.open(address);
    ByteBuffer buffer = ByteBuffer.allocate(4);
    IntBuffer view = buffer.asIntBuffer();
    for (int expected = 0; ; expected++) {
        client.read(buffer);
        int actual = view.get();
        buffer.clear();
        view.rewind();
        if (actual != expected) {
            System.err.println("Expected " + expected + "; was " + actual);
            break;
        }
        System.out.println(actual);
    }
} catch(IOException ex) { ex.printStackTrace(); } }
```



# SSDP SIMPLE SERVICE DISCOVERY PROTOCOL (SSDP)

- su una rete locale attivi molti protocolli che usano il multicast
- SSDP attivo sulla porta 1900 associato all'indirizzo di multicast 239.255.255.250
- un protocollo di discovery usato per scoprire quali servizi sono disponibili in una rete
- ogni servizio è definito mediante specifiche UPnP
- il server SSDP invia pacchetti di advertisement sul gruppo di multicast
- formato di un pacchetto di advertisement

```
HTTP/1.1 200 OK
CACHE-CONTROL: max-age=120
ST: urn:schemas-upnp-org:device:WANDevice:1
USN: uuid:fc4ec57e-b051-11db-88f8-
0060085db3f6::urn:schemas-upnp-org:device:WANDevice:1
EXT:
SERVER: Net-OS 5.xx UPnP/1.0
LOCATION: http://192.168.0.1:2048/etc/linuxigd/gatedesc.xml
```

# SSDP SIMPLE SERVICE DISCOVERY PROTOCOL (SSDP)

## Messaggio di advertisement

- USN, Unique Service Name
  - UUID (Universally Unique Identifier) che identifica una device o un servizio
- LOCATION
  - punta ad una URL
  - a quella URL l'utente può trovare una descrizione XML delle capability del servizio

```
HTTP/1.1 200 OK
CACHE-CONTROL: max-age=120
ST: urn:schemas-upnp-org:device:WANDevice:1
USN: uuid:fc4ec57e-b051-11db-88f8-
0060085db3f6::urn:schemas-upnp-org:device:WANDevice:1
EXT:
SERVER: Net-OS 5.xx UPnP/1.0
LOCATION: http://192.168.0.1:2048/etc/linuxigd/gatedesc.xml
```

# SNIFFING SSDP IN JAVA

```
import java.io.*; import java.net.*;

public class MulticastSniffer {

    public static void main(String[] args) {

        InetAddress group = null;

        int port = 0;

        // read the address from the command line

        try {

            group = InetAddress.getByName(args[0]);

            port = Integer.parseInt(args[1]);

        } catch (ArrayIndexOutOfBoundsException | NumberFormatException |
                  UnknownHostException ex)

        {

            System.err.println( "Usage: java MulticastSniffer multicast_address
                               port");

            System.exit(1);
        }

        MulticastSocket ms = null;
```



# SNIFFING SSDP IN JAVA

```
try {ms = new MulticastSocket(port);
ms.joinGroup(group);
byte[] buffer = new byte[8192];
while (true) {
    DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
    ms.receive(dp);
    String s = new String(dp.getData(), "8859_1");
    System.out.println(s); }
} catch (IOException ex) { System.err.println(ex);
} finally {
if (ms != null) {
    try {
        ms.leaveGroup(group);
        ms.close(); } catch (IOException ex) { } } }
```



# SNIFFING SSDP IN JAVA: OUTPUT GENERATO

```
$ Java MulticastSniffer 239.255.255.250 1900

NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
CACHE-CONTROL: max-age=1801
NTS: ssdp:alive
LOCATION: http://192.168.1.1:49152/gKgq6lu34h/wps_device.xml
SERVER: Unspecified, UPnP/1.0, Unspecified
NT: urn:schemas-wifialliance-org:service:WFAWLANConfig:1
USN: uuid:03fef68b-319f-5ea7-80a8-2aea5bb7e216::urn:schemas-
wifialliance-org:service:WFAWLANConfig:1
.....
```

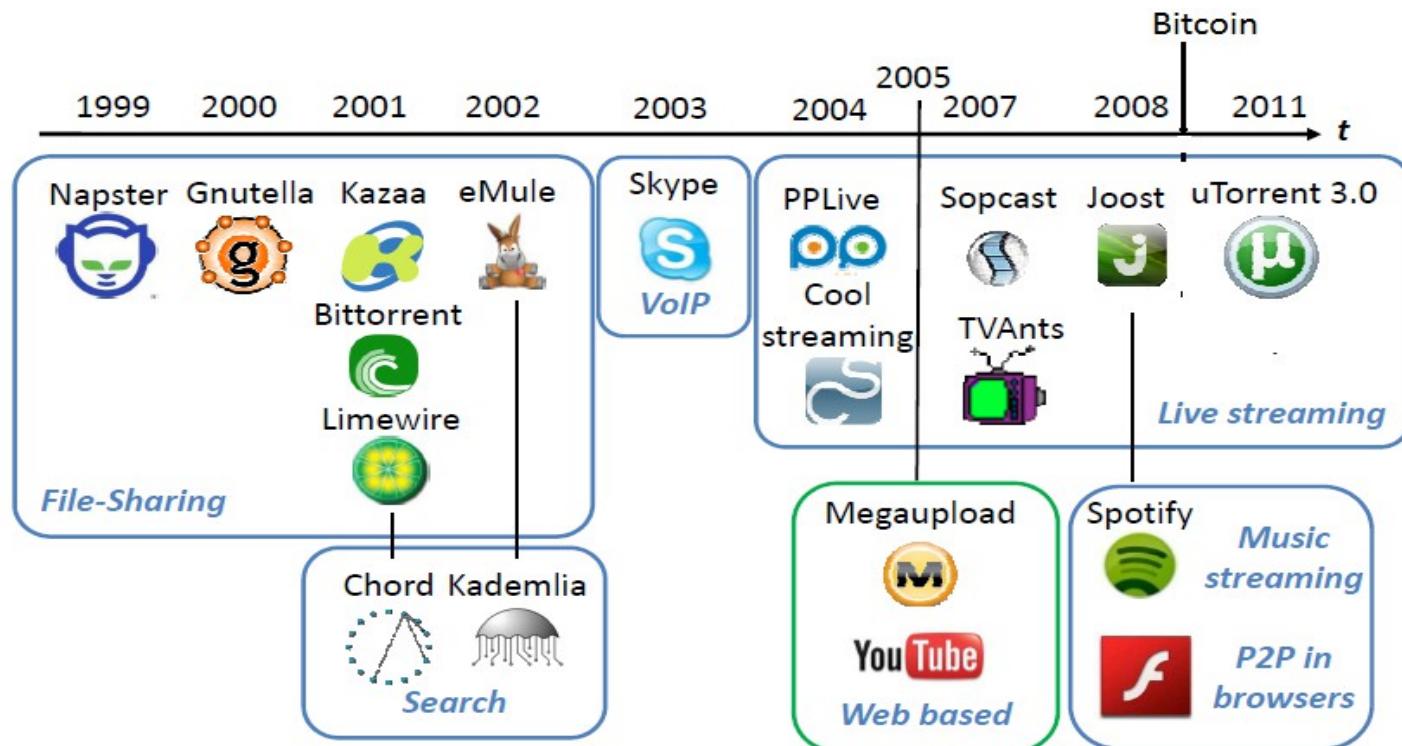
# ASSIGNMENT

Definire un Server TimeServer, che

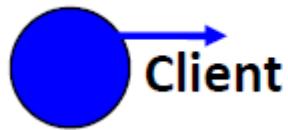
- invia su un gruppo di multicast dategroup, ad intervalli regolari, la data e l'ora.
- attende tra un invio ed il successivo un intervallo di tempo simulata mediante il metodo sleep().
- l'indirizzo IP di dategroup viene introdotto da linea di comando.
- definire quindi un client TimeClient che si unisce a dategroup e riceve, per dieci volte consecutive, data ed ora, le visualizza, quindi termina.

# IL MODELLO PEER TO PEER

- in questo corso abbiamo analizzato il modello client server, come base per costruire applicazioni di rete
- molte applicazioni sono sviluppate secondo un modello alternativo: il modello peer to peer: nato nel 2000 per applicazioni di file-sharing

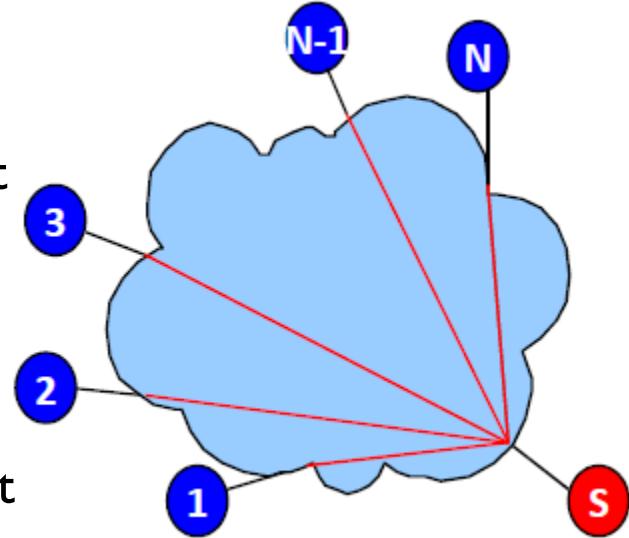


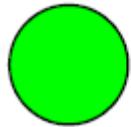
# IL PARADIGMA CLIENT SERVER



- in esecuzione sugli end host
- comportamento on/off
- utilizza servizi
- inoltra richieste
- nessuna interazione tra client
- deve conoscere un riferimento al servizio

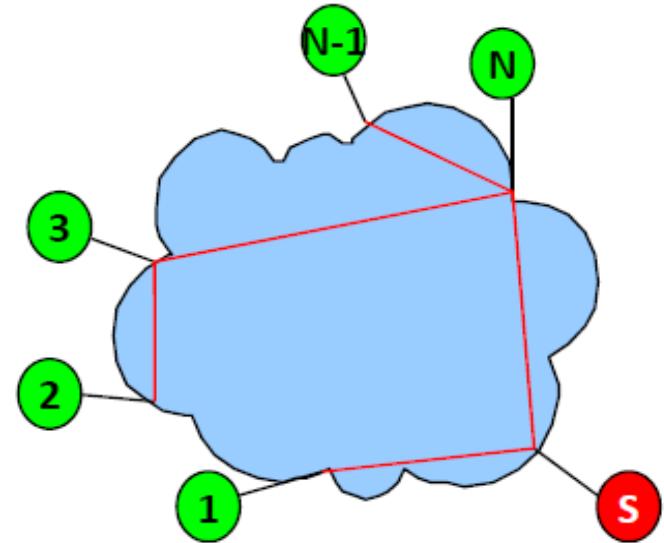
- in esecuzione su un host dedicato
- “always on”
- fornisce servizi
- riceve richieste dai client
- soddisfa le richieste dell'utente
- deve avere un IP fisso (o DNS name)



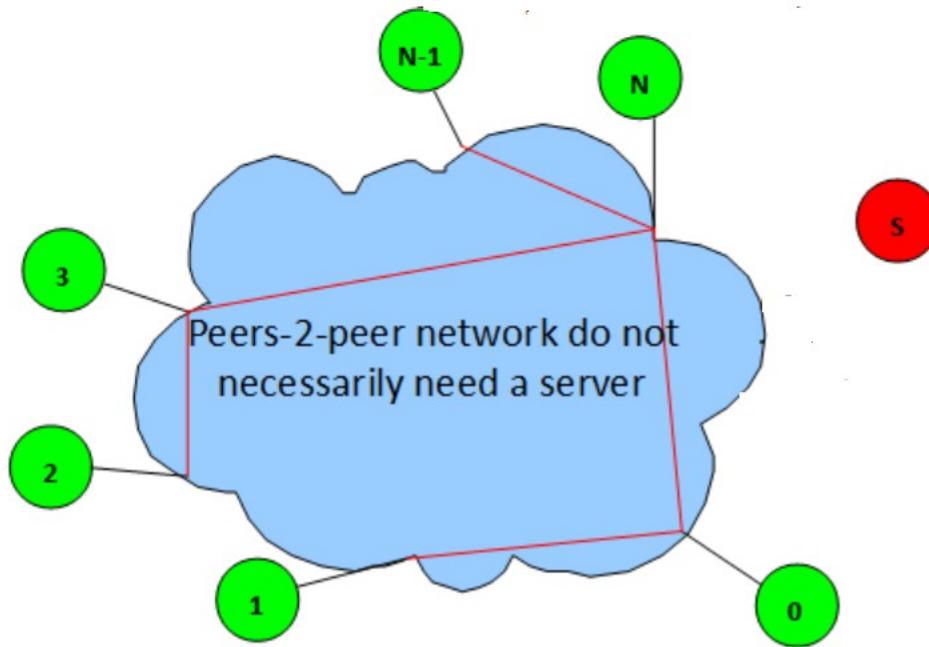


## Peer

- in esecuzione sugli end-hosts
- produttori e consumatori di servizi
- comportamento on/off
- alto livello di dinamicità (**churn**)
- necessaria una fase di bootstrap
- necessari meccanismi per la scoperta di peer
- comunicano tra di loro
- necessari protocolli a livello applicazione per
  - evitare il fenomeno dei free riders
  - incentivare partecipazione e collaborazione



# IL PARADIGMA P2P



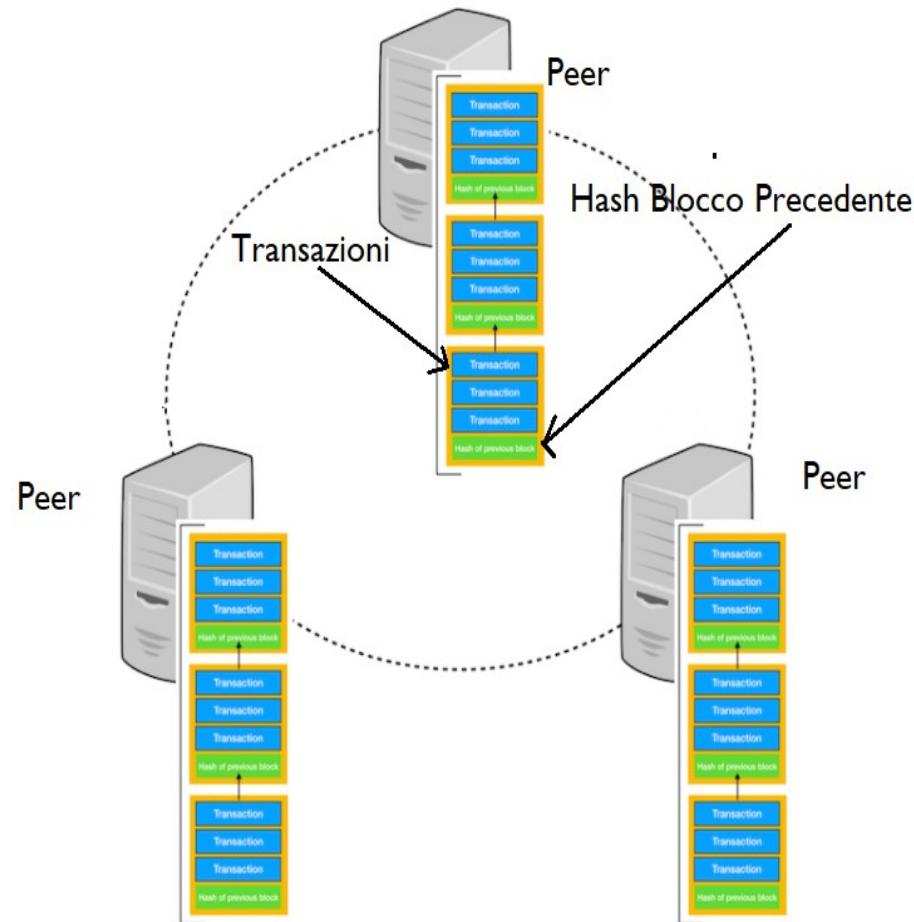
notare che:

- un server è sempre presente, ma serve solo nella fase di bootstrap
- i server non sono necessari per la condivisione delle risorse

- P2P file sharing
  - file sharing: light weight/ best effort
  - la persistenza e la sicurezza non sono l'obiettivo principale
  - l'anonimato è importante
  - applicazione
    - Napster
    - Gnutella, KaZaa
    - eMule
    - BitTorrent
- P2P Media Streaming
  - Skype (first versions)
- Cryptocurrencies and blockchains
- Distributed file System: Internet Planetary File System (IPFS)

# BLOCKCHAIN IN BREVE

- database **distribuito** e **replicato** sui nodi di un sistema **peer to peer** (P2P)
  - un insieme di blocchi collegati mediante **puntatori hash**
  - in ogni blocco è presente l'**hash** blocco precedente
  - ogni peer possiede una copia consistente dell'intero database
- operazioni
  - **append only:** accodare progressivamente registrazioni organizzate **in blocchi**
  - leggere il contenuto di una qualsiasi registrazione



# LE BLOCKCHAIN: EVOLUZIONE

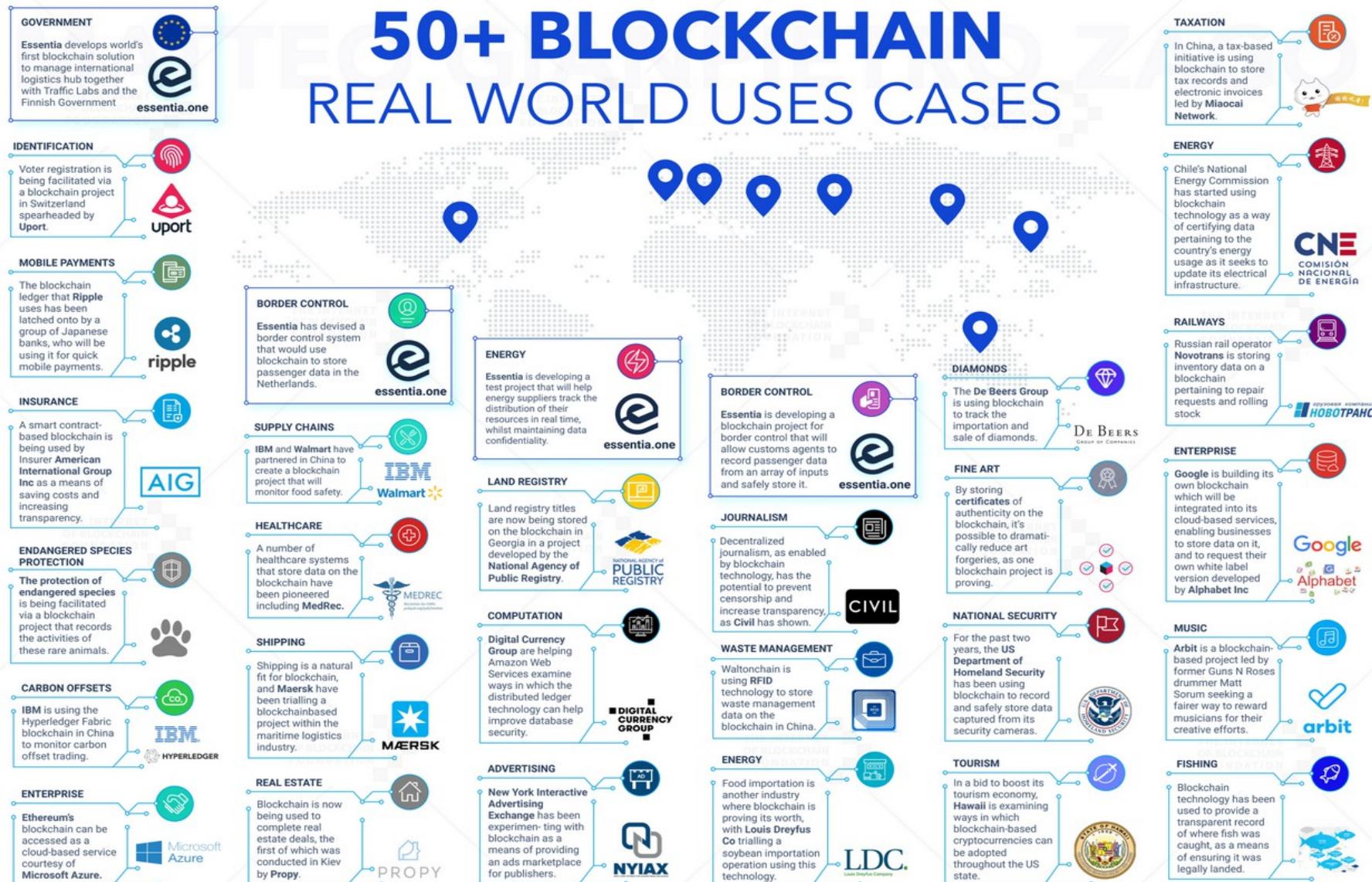


# L'ECOSISTEMA DI BITCOIN

- non solo blockchain, ma diversi attori coinvolti
  - Exchangers, Wallets, NFT Marketplaces
- una vera e propria economia basata su cryptocurrencies



# 50+ BLOCKCHAIN REAL WORLD USES CASES



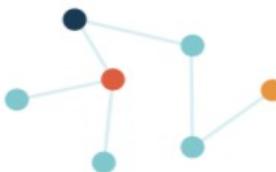
# P2P FILE SYSTEM: IPFS



Your file, and all of the **blocks** within it, is given a **unique fingerprint** called a **cryptographic hash**.



IPFS **removes duplications** across the network.



Each **network node** stores only content it is interested in, plus some indexing information that helps figure out which node is storing what.



When you **look up a file** to view or download, you're asking the network to find the nodes that are storing the content behind that file's hash.



You don't need to remember the hash, though — every file can be found by **human-readable names** using a decentralized naming system called **IPNS**.

# P2P FILE SYSTEM: IPFS

- IPFS: integrazione file system distribuito con blockchain
- dati di grande dimensione memorizzati su IPFS, hash dei dati su blockchain
- implementato in LibP2P
  - una libreria per lo sviluppo di applicazioni P2P
  - nuove tecnologie: Multiprotocol Project
  - Distributed Hash Table: Kademlia



# CORSI INSEGNATI SUL TEMA BLOCKCHAIN

- *P2P & Blockchain*, corso fondamentale, Laurea Magistrale in Informatica, Curriculum ICT
- *Blockchain e AI*, corso complementare, Laurea Magistrale Diritto per le nuove tecnologie
- *laboratorio web scraping*: corso complementare Laurea Triennale in Informatica
  - focus del corso è sull' analisi di dati provenienti dall'ecosistema delle blockchain
  - parte teorica:
    - richiami di elementi di statistica di base
    - graph models: random graph, power law, small worlds
    - proprietà caratteristiche di un grafo (degree, centrality, clustering coefficient,...)
    - struttura delle transazioni di diversi tipi di blockchain, (Bitcoin, Ethereum,...), transaction graphs
  - laboratorio:
    - uso di API per il reperimento di dati
    - breve introduzione a Python
    - Pandas: Numpy e matplotlib
    - librerie per l'analisi di grafi (NetworkX, Networkkit,...)
    - applicazione degli strumenti a casi s'uso legati all'ecosistema delle blockchain (transazioni, Exchange services, mixers, miners,...)



# TESI DI LAUREA TRIENNALI E MAGISTRALI

- applicazione di tecniche crittografiche in collaborazione con la Prof. Bernasconi
  - zero-knowledge
  - authenticated data structures
- analisi di transazioni
  - bot discovery
  - attacchi
  - NFT markets e NFT communities
- layer-2 technologies
  - lightning network: analisi e algoritmi di routing
  - cross-chain solutions: Polygon, ChainBridge
  - oracoli: ChainLink
- applicazioni della blockchain
  - identità digitale: Self Sovereign Identity (SSI)
  - meccanismi di access control mediante smart contract

