



Laboratorio III

Thread e Processi

Java

Implements Runnable

Extends Thread

Thread Pool

Fixed Thread Pool

Cached Thread Pool

Input/Output

Java Blocking Queue

Callable

Meccanismo Produttore/Consumatore

Monitor

Monitor vs Lock

Concurrent Collections

ArrayList

LinkedList

Iterators e HashMap

HashMap

InetAddress

Factory methods

Instance methods

Spam checker

Paradigma client/server

Stream Sockets

Volatile e Atomic

Serializzazione: JSON e native serialization

JSON

GSON (Google GSON)

Serializzazione Java

Java NIO

Buffers

Channels

Channel Multiplexing

UDP DatagramPacket e DatagramSocket

Multicasting

Thread e Processi

La differenza principale tra *thread* e *processi* è che un processo rappresenta un vero e proprio programma in esecuzione (ad esempio due diverse applicazioni come Word o Excel sono due processi diversi) mentre un thread (light weight process) è un flusso di esecuzione all'interno di un processo.

Nonostante questa differenza, possiamo comunque utilizzare la parola *multitasking* per riferirci ad entrambi:

- a livello di processo è controllato esclusivamente dal sistema operativo.
- a livello di thread è controllato, almeno in parte, dal programmatore.

I thread, inoltre, condividono lo stesso spazio degli indirizzi e risultano meno costosi per quanto riguarda il *context switch* e la comunicazione. Possono essere eseguiti nei seguenti modi:

- single core: multiplexing, interleaving (meccanismi di time sharing...)
- multicore: più flussi di esecuzione eseguiti in parallelo, simultaneità di esecuzione.

Il *multithreading* avviene quando un programma sfrutta più thread per svolgere le operazioni e ha molti vantaggi:

- migliore utilizzazione delle risorse: quando un thread è sospeso, altri thread vengono mandati in esecuzione. In certi casi si può avere una riduzione del tempo complessivo di esecuzione.
- migliore performance per applicazioni computationally intensive: dividere l'applicazione in task ed eseguirli in parallelo.

I principali problemi ai quali si può andare in contro se si sfrutta il multithreading sono:

- debugging e manutenzione del software più difficile.
- race conditions, sincronizzazioni.
- deadlock, livelock, starvation,...

Per quanto riguarda la *programmazione di rete*, ci occuperemo principalmente di applicazioni client-server dove magari avremo più client serviti simultaneamente quindi un determinato client non dovrà aspettare che il server termini di elaborare la richiesta del client precedente. Il throughput dell'applicazione può essere incrementato se client diversi sono serviti da thread diversi, ma solo fino ad un certo punto. Oltre quel limite, i thread iniziano a competere per la CPU e il costo del context switch supera il beneficio del multithreading. Bisogna quindi limitare questo fenomeno con il meccanismo del *threadpooling*.

Java

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`.

Java è anche in grado di attivare altri thread automaticamente (gestore eventi, interfaccia, garbage collector,...) e ognuno di essi può a sua volta creare ed attivare altri threads.

Esistono due modalità per creare ed attivare esplicitamente un thread:

- Implements *Runnable*
- Extends *Thread*

Implements *Runnable*

Come prima cosa bisogna definire un task, poi bisogna creare un oggetto thread e passargli il task che contiene il codice da eseguire ed infine attivare il thread con il metodo `start()`.

Per definire un task bisogna definire una classe che implementi l'interfaccia *Runnable* e creare un'istanza *R* di questa classe (da passare al thread).

```
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();
    }
}
```

L'interfaccia *Runnable* appartiene al package `java.lang` e contiene solo la segnatura del metodo `void run()`, che deve essere implementato.

Un'istanza della classe che implementa *Runnable* è un task, ovvero un frammento di codice che può essere eseguito in un thread. La creazione del task non implica la creazione di un thread che lo esegua e lo stesso task può essere eseguito da più thread.

Si può definire un task anche attraverso una *classe anonima* nel seguente modo:

```

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

```

Extends Thread

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

```

public class ExtendingThread {
    public static class MyThread extends Thread {
        @Override
        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();
    }
}

```

La classe *Thread* memorizza un riferimento all'oggetto *Runnable*, eventualmente passato come parametro, nella variabile *runnable* e definisce il metodo *run()* come segue:

```

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

```

Quando viene invocato il metodo *start()* se il metodo *run()* è stato ridefinito mediante *overriding* si invoca il metodo *run()* più specifico, che è quello definito dal programmatore, altrimenti si esegue il metodo *run()* predefinito nella classe *Thread*. Se la variabile *runnable* è diversa da *null*, questo metodo, a sua volta, invoca il metodo *run()* dell'oggetto *Runnable* passato e si esegue il metodo definito dal programmatore.

Esempio di programma multithread (Implements *Runnable*):

```

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(); // ATTENZIONE: NON usare thread.run();
        }
        System.out.println("Avviato Calcolo Tabelline");
    }
}

```

```

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);
        }
    }
}

```

```
}  
}
```

Il metodo `start()` segnala allo *scheduler* (tramite JVM) che il thread può essere attivato (invocando un metodo nativo) e l'ambiente del thread viene inizializzato. Viene restituito immediatamente il controllo al chiamante, senza attendere che il thread attivato inizi la sua esecuzione e la stampa del messaggio "Avviato Calcolo Tabelline" precede quelle effettuate dai threads. Questo significa che il controllo è stato restituito al thread chiamante (in questo caso il main) prima che sia iniziata l'esecuzione dei threads attivati.

Esempio di programma multithread (Estende *Thread*):

```
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");  
    }  
}
```

```
public class Calculator extends Thread {  
    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);  
        }  
    }  
}
```

In Java una classe può estendere una sola 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* e implementi l'interfaccia *Runnable*.

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. Se uno dei thread usa l'istruzione `System.exit()` per terminare l'esecuzione, allora tutti i threads terminano la loro esecuzione.

Thread Pool

Possiamo immaginare come scenario di riferimento il seguente: eseguire un gran numero di task (ad esempio un task per ogni client, nel server).

Avere un thread per ogni task, però, può diventare insostenibile, specialmente nel caso di lightweight tasks molto frequenti.

L'alternativa è quella di creare un pool di thread (*thread pool*): un gestore software di thread utilizzato per ottimizzare e semplificare l'utilizzo dei thread all'interno di un programma. L'obiettivo è, quindi, quello di riusare lo stesso thread per l'esecuzione di più tasks e diminuire il costo per l'attivazione/terminazione dei threads, ma anche controllare il numero massimo di thread che possono essere eseguiti concorrentemente.

Un thread pool funziona nel seguente modo:

1. Si crea una coda (FIFO) di task che aspettano l'esecuzione.
2. Un pool di thread disponibili per l'esecuzione del task.
3. Il sistema di gestione del thread pool chiede se esiste un thread libero per l'esecuzione del primo task della coda.
4. Il task viene assegnato al thread libero che viene tolto dal pool dei thread disponibili.

Se tutti i thread sono occupati nell'esecuzione del task e il pool è vuoto ho 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 thread pool.

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 *Executor* 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()`.

Fixed Thread Pool

Una *fixed thread pool* è un tipo thread pool con comportamento predefinito. Viene creato un numero fisso di thread, riutilizzati per l'esecuzione di più tasks e 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* illimitata.

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class ExampleFixed {
    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(10);
        for (int i=0; i<100; i++) {
            service.execute(new Task(i));
        }
        System.out.println("Thread Name: " + Thread.currentThread().getName());
    }
}
```

Cached Thread Pool

Un altro tipo di thread pool è la *cached thread pool* con comportamento predefinito. Viene attivata con `ExecutorService service = Executors.newCachedThreadPool();` e se tutti i thread del pool sono occupati nell'esecuzione di altri task e c'è un nuovo task da eseguire, viene creato un altro thread, quindi non si ha nessun limite alla dimensione del pool. Se disponibile, viene riutilizzato un thread che ha terminato l'esecuzione di un task precedente e se un thread rimane inutilizzato per 60 secondi, la sua esecuzione termina. Nel *cached thread pool* è importante il concetto di *elasticità*: "un pool che può espandersi all'infinito, ma si contrae quando la domanda di esecuzione di task diminuisce".

```
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));
        }
        System.out.println("Thread Name: " + Thread.currentThread().getName());
    }
}

```

Input/Output

Per I/O si intendono tutte quelle operazioni che permettono di 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.err*
- *in-memory buffers* (array)

La definizione di un insieme di astrazioni per la gestione dell'I/O è una delle parti più complesse di un linguaggio.

Diversi tipi di device di I/O: se il linguaggio dovesse gestire ogni tipo di device come caso speciale, la complessità sarebbe enorme, quindi si ha la necessità di astrazioni opportune per rappresentare un device di I/O.

In Java la prima astrazione definita è basata sul concetto di *stream*.

Tra le caratteristiche principali degli streams in Java:

- accesso sequenziale
- 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 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 e scritture.

InputStream e *OutputStream* operano su "row bytes" e vengono chiamate classi filtro che compiono trasformazioni sui dati a basso livello.

Tipi di filtro:

- *file stream*
- *readers/writers*: 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 e passa i dati al programma o al filtro successivo.

Usare dei *Buffered Stream/Data Stream* potrebbe risultare conveniente a volte in quanto 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, quindi si ha un miglioramento significativo delle performance.

Il package `java.io` definisce i concetti base per gestire l'I/O e ha l'obiettivo di fornire un'astrazione che incapsuli tutti i dettagli del dispositivo sorgente/destinazione dei dati.

Distingue tra:

- stream di byte (analoghi ai file binari del C): modellate da altrettante classi base astratte ovvero *InputStream* e *OutputStream*.
- stream di caratteri (analoghi ai file di testo del C): modellate da altrettante classi base astratte ovvero *Reader* e *Writer*.

La classe `InputStream` è la classe base e definisce il concetto generale di “canale di input” che lavora con byte. Il costrutto apre lo stream con `read()` e legge uno o più byte e chiude lo stream con `close()`. `InputStream` è una classe astratta e il metodo `read()` dovrà essere realmente definito dalle classi derivate.

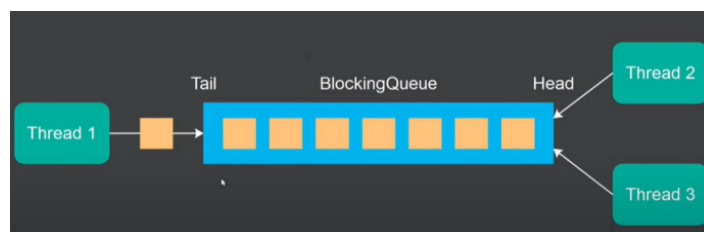
Il metodo `read()` permette di leggere uno o più byte dal file e restituisce il byte letto come intero tra 0 e 255, se lo stream è finito restituisce -1. Se non ci sono byte, ma lo stream non è finito, rimane in attesa dell’arrivo di un byte.

Per scrivere sul file si usa il metodo `write()` che permette di scrivere uno o più byte. Scrive l’intero compreso tra 0 e 255 passatogli come parametro e non restituisce nulla.

Sostituendo `FileInputStream` → `BufferedFileInputStream` e `FileOutputStream` → `BufferedFileOutputStream` i tempi di esecuzione del programma si abbassano notevolmente anche se la dimensione del file rimane invariata.

Java Blocking Queue

Per utilizzarla bisogna inserire `import java.util.concurrent` ed è una Java interface che rappresenta una coda (inserimento in coda e estrazione in testa). La differenza con `Queue<E>` è che la JBQ è pensata per essere utilizzata in un ambiente multithread (quindi in un ambiente *thread-safe*) e permette una corretta sincronizzazione tra i thread che inseriscono e quelli che eliminano elementi dalla coda (per esempio Thread1 si blocca se la coda è piena, Thread2 e Thread3 se è vuota).



Esistono varie implementazioni della JBQ ma le più importanti sono:

- **ArrayBlockingQueue**: dimensione limitata, definita in fase di inizializzazione. Memorizza gli elementi all’interno di un oggetto Array e non viene creato nessun ulteriore oggetto, non sono possibili inserzioni/rimozioni in parallelo e si ha una sola *lock* per tutta la scrittura.
- **LinkedBlockingQueue**: può essere illimitata (`size = Integer.MAX_VALUE`) o limitata. Mantiene gli elementi in una *LinkedList* con maggior occupazione della memoria e si ha un nuovo oggetto per ogni inserzione, con possibili inserzioni ed estrazioni concorrenti. Si hanno *lock* separate per lettura e scrittura con maggior *throughput*.

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

Operazioni

La classe `ThreadPoolExecutor` permette di creare un `ThreadPool` usando sei parametri.

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

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

`CorePoolSize`, `MaximumPoolSize`, `keepAliveTime` controllano la gestione dei thread del pool e `workQueue` è una struttura dati necessaria per memorizzare gli eventuali tasks in attesa di esecuzione.

Il core è il nucleo minimo di thread attivi nel pool, i thread del core possono essere attivati in due modi:

- 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. L'obiettivo è riempire il pool il prima possibile.

Quando tutti i threads sono stati creati, la politica cambia.

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 e di un'unità di misura.

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 `CorePoolSize` altrimenti viene lanciata `allowCoreThreadTimeOut(boolean value)`.

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

Esistono anche altri tipi di `ThreadPool` come:

- Single Threaded Executor: ha un singolo thread ed è equivalente ad invocare `FixedThreadPool` di dimensione 1.
- Scheduled Thread Pool: distanziare esecuzione dei task con un certo delay e task periodici.

È possibile gestire il rifiuto di un task scegliendo esplicitamente un *rejection policy* al momento della creazione del task.

- `AbortPolicy`: politica di default, consiste nel sollevare `RejectedExecutionException`.
- `DiscardPolicy`, `DiscardOldestPolicy`, `CallerRunPolicy`: altre politiche predefinite.

Bisogna quindi definire un *custom rejection handler* implementando l'interfaccia `RejectExecutionHandler` ed il metodo `rejectedExecution`.

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

- i tasks vengono eseguiti in modo asincrono rispetto alla loro sottomissione.
- in un certo istante, alcuni task sottomessi precedentemente possono essere completati, alcuni in esecuzione, alcuni in coda.

La terminazione può avvenire:

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

Il metodo `shutdown()` fa parte della terminazione graduale: inizia la terminazione e 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.

Callable

Un oggetto di tipo *Runnable* incapsula un'attività che viene eseguita in modo asincrono ed il metodo `run()` è un metodo asincrono, senza parametri e che non restituisce un valore di ritorno.

Tramite `Interface Callable` siamo in grado di definire un task che può restituire un risultato e sollevare eccezioni. Per accedere al risultato in modo asincrono si usa `Future interface` che 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
- Per cancellare una computazione

La classe `FutureTask` fornisce un'implementazione della interfaccia *Future*.

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

L'interfaccia *Callable* contiene solo il metodo `call()`, analogo al metodo `run()` dell'interfaccia *Runnable*. Il codice del task è implementato nel metodo `call()` ma a differenza del metodo `run()`, il metodo `call()` può restituire un valore e sollevare eccezioni. Il parametro di tipo `<V>` indica il tipo del valore restituito.

```
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;  
    }  
}
```

```
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 += f.get();  
                    list.remove(f);  
                }  
            }  
        System.out.println("La somma è " + s);  
        executor.shutdown();  
    }  
}
```

```
}  
}
```

Bisogna quindi sottomettere direttamente l'oggetto di tipo *Callable* al pool mediante il metodo `submit()`, la sottomissione restituisce un oggetto di tipo *Future* ed ogni oggetto *Future* è associato ad uno dei task sottomessi al *ThreadPool*, è possibile applicare diversi metodi all'oggetto *Future*.

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

Il metodo `get()` si blocca fino a che il thread non ha prodotto il valore richiesto e restituisce il valore calcolato.

Il metodo `get(long timeout, TimeUnit)` definisce un tempo massimo di attesa della terminazione del task, dopo cui viene sollevata una `TimeoutException`. Risulta possibile cancellare il task e verificare se la computazione è terminata oppure se è stata cancellata.

Come detto già in precedenza, esistono altri tipi di thread pool, come il *Single Threaded Executor* e lo *Scheduled Thread Pool*.

L'interfaccia `ScheduledExecutorService` dà la possibilità di schedulare un task dopo un certo periodo di tempo (delay) e periodicamente.

Il metodo `schedule(Runnable command, long delay, TimeUnit unit)` esegue un task *Runnable* (o *Callable*) dopo un certo intervallo di tempo.

Il metodo `scheduleAtFixedRate(Runnable command, long initialDelay, long delay, TimeUnit unit)` esegue un task dopo un intervallo di tempo iniziale e lo ripete periodicamente. Se il tempo di esecuzione del task è maggiore del periodo specificato, le sue seguenti esecuzioni possono essere ritardate.

Il metodo `scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)` esegue un task dopo un intervallo iniziale e lo ripete periodicamente con un intervallo dato tra la terminazione di una esecuzione e l'inizio della successiva.

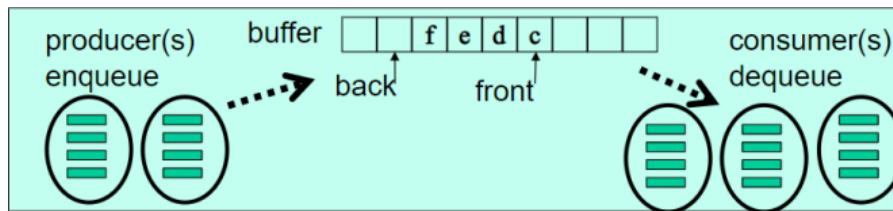
Meccanismo Produttore/Consumatore

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, si parla quindi di *race conditions*.

Una *sezione critica* è un blocco di codice in cui si effettua l'accesso ad una risorsa condivisa e che deve essere eseguito da un thread per volta. Quindi è necessario implementare classi *thread safe*, il codice dei metodi della classe può essere utilizzato/condiviso in un ambiente concorrente senza provocare inconsistenze o comportamenti inaspettati.

Si hanno quindi le seguenti alternative per definire classi thread safe:

- Usare classi thread safe predefinite:
 - *Concurrent-aware interfaces*: Blocking Queue, Transfer Queue, Blocking Dequeue, Concurrent Map, Concurrent Navigable Map
 - *Concurrent-aware classes*: LinkedBlockingQueue, ArrayBlockingQueue, PriorityBlockingQueue, DelayQueue, SynchronousQueue, CopyOnWriteArrayList, CopyOnWriteArraySet, ConcurrentHashMap
- Usare i monitor
- Usare le *lock* a basso livello



Di solito, il meccanismo produttore/consumatore viene mostrato con il 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.

L'obiettivo principale è garantire che il produttore non provi ad aggiungere un dato nella coda se è piena ed il consumatore non provi a rimuovere un dato da una coda vuota.

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.

Si ha bisogno di 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 e si riattiva quando c'è una posizione libera).

Esistono due tipi di sincronizzazione:

- *Implicita*: la mutua esclusione dell'oggetto condiviso è garantita dall'uso di lock
- *Esplicita*: occorrono altri meccanismi

```
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();
    }
}
```

```
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();}
    }
}
}

```

Monitor

Il *monitor* è un meccanismo linguistico ad alto livello per la sincronizzazione. Incapsula un oggetto condiviso e le operazioni che vengono invocate dai threads su di esso, in modo concorrente.

Un monitor offre:

- *Mutua esclusione* sulla struttura: lock implicite gestite dalla JVM dove un solo thread per volta accede all'oggetto condiviso.
- Coordinazione tra i thread: si hanno dei meccanismi per la sospensione sullo stato dell'oggetto condiviso, simili a variabili di condizione (`wait()`) e meccanismi per la notifica di una condizione ai thread sospesi su quella condizione con `notify` e `notifyall`.

In Java ci sono dei monitor built-in, ovvero una classe di oggetti utilizzabili concorrentemente in modo thread safe. Ad ogni oggetto, cioè ad ogni istanza di una classe, viene associata:

- Una *intrinsic lock* o lock implicita. Viene acquisita con metodi o blocchi di codice `synchronized` e garantisce la mutua esclusione nell'accesso all'oggetto, quindi si ha una gestione automatica della coda di attesa da parte della JVM.
- Una *wait queue* gestita dalla JVM: che usa `wait()` e `notify/notifyall`.

Vengono gestite anche due code in modo implicito:

- *Entry Set*: threads in attesa di acquisire la lock.
- *Wait Set*: threads che hanno eseguito una `wait` e sono in attesa di una `notifyAll`.

I metodi di un built-in monitor possono essere resi thread safe annotandoli con la parola chiave *synchronized*. Si ha inoltre una 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 un thread alla volta può eseguire uno dei metodi *synchronized* del monitor sulla stessa istanza di una classe.

In altre parole, il metodo *synchronized*, quando viene invocato:

- Tenta di acquisire la lock intrinseca associata all'istanza dell'oggetto su cui esso è invocato e 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 in maniera normale o in maniera eccezionale (ad esempio con una *uncaught exception*).

I costruttori non devono essere dichiarati *synchronized* (altrimenti 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 e la lock viene

associata ad un'istanza dell'oggetto, non alla classe, quindi metodi su oggetti che sono istanze diverse della stessa classe possono essere eseguiti in modo concorrente.

Come già anticipato, Java fornisce tre metodi di base per coordinare i thread:

- `void wait()`: sospende il thread fino a che un altro thread invoca una `notify()` o `notifyAll()` sullo stesso oggetto. Implementa attesa passiva del verificarsi di una condizione e rilascia la lock sull'oggetto.
- `void notify()`: sveglia un singolo thread in attesa su questo oggetto, *non* se nessun thread è in attesa.
- `void notifyAll()`: sveglia tutti i thread in attesa su questo oggetto, che competono per riacquisire la lock.

I thread invocati su un oggetto appartengono alla classe `Object`. Occorre acquisire la lock intrinseca prima di invocarli, altrimenti viene sollevata l'eccezione `IllegalMonitorException`. Vengono eseguiti all'interno di metodi sincronizzati e se non si fa riferimento ad un oggetto, il riferimento implicito è `this`.

```
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 (InterruptedException e) {}
        }
        // gestione puntatori coda
        items[putptr] = x;
        putptr++;
        ++count;
        if (putptr == items.length) putptr = 0;
        System.out.println("Message Produced"+x);
        notifyAll();
    }
    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;
    }
}
```

Una regola fondamentale quando si usano `wait()` e `notify()/notifyAll()` è testare sempre la condizione relativa al `wait()` sempre 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.

Se non si intende sincronizzare un intero metodo, si può sincronizzare un blocco di codice all'interno di un metodo:

```
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 e la rilascia quando termina il blocco sincronizzato.

Risulta possibile anche attendere il verificarsi di una condizione su un oggetto diverso da `this`:

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

E segnalare una condizione:

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

Monitor vs Lock

Monitor (Vantaggi)

- L'unità di sincronizzazione è il metodo: tutte le sincronizzazioni sono visibili esaminando la segnatura dei metodi.
- Costrutti strutturati, diminuisce la complessità del programma concorrente: deadlocks, mancato rilascio di lock, maggior manutenibilità del software.

Lock (Vantaggi)

- Maggior numero di funzioni disponibili e maggiore flessibilità.
- `tryLock()` il thread prova ad acquisire la lock ma senza bloccare.
- *read/write locks*: multiple reader, single writer.

Monitor (Svantaggi)

- "coarse grain" synchronization
- per-object synchronization
- Possono diminuire il livello di concorrenza

Lock (Svantaggi)

- Codice poco leggibile se usate in modo non strutturato.

Concurrent Collections

Java Collection Framework è un insieme di classi che consentono di lavorare con gruppi di oggetti, ovvero una collezione di oggetti contenute nel package `java.util`.

Si dividono in:

- *synchronized collections*
- *concurrent collections*

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

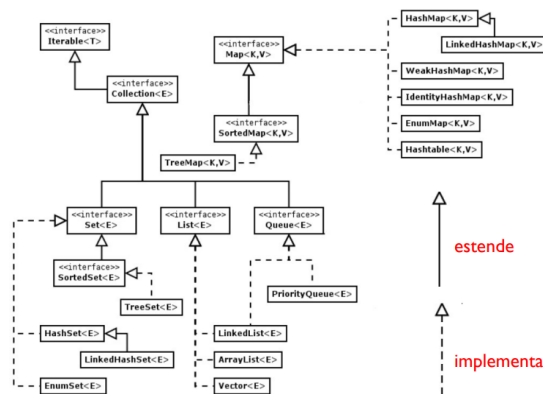
- *list*: è una collezione ordinata (una sequenza) di valori dove possono esistere duplicati.

-
- ```

classDiagram
 class IterableT["<<interface>>
Iterable<T>"]
 class MapKV["<<interface>>
Map<K,V>"]
 class CollectionE["<<interface>>
Collection<E>"]
 class SetE["<<interface>>
Set<E>"]
 class ListE["<<interface>>
List<E>"]
 class QueueE["<<interface>>
Queue<E>"]
 class SortedSetE["<<interface>>
SortedSet<E>"]
 class SortedMapKV["<<interface>>
SortedMap<K,V>"]

 IterableT <|-- CollectionE
 MapKV <|-- SortedMapKV
 CollectionE <|-- SetE
 CollectionE <|-- ListE
 CollectionE <|-- QueueE
 SetE <|-- SortedSetE

```



*Collections* (con la "s finale) contiene metodi utili per l'elaborazione di collezioni di qualunque tipo:

- ## ArrayList

- 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

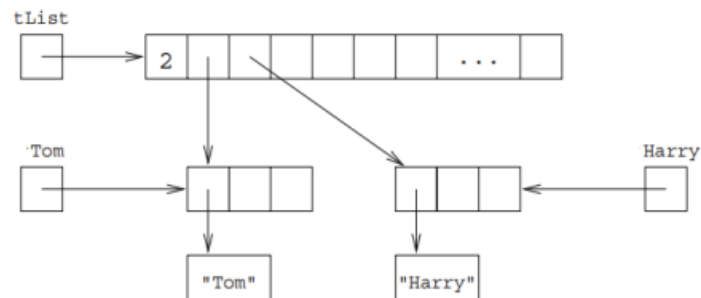
*Vector* è thread safe.

15

```

System.out.println(Tom);
System.out.println(Harry);
System.out.println(pList.size());
}
}

```



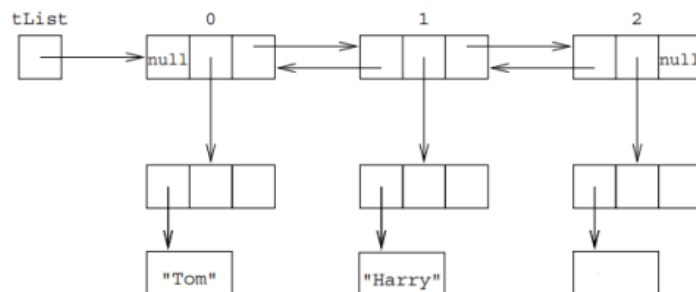
## LinkedList

```

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(pList.size());
 }
}

```



## Iterators e HashMap

Un *iterator* è usato per accedere agli elementi di una collezione, uno alla volta ma 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.

```

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 un nuovo iteratore.

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

Nel *Java Concurrency Framework* sono tre i package principali:

- `java.util.concurrent`
  - *Exeutor, concurrent collections, semaphores, ...*
- `java.util.concurrent.atomic`
  - *AtomicBoolean, AtomicInteger, ...*
- `java.util.concurrent.locks`
  - *Condition, Lock, ReadWriteLock*

Le collezioni non thread safe, non offrono alcun supporto per la sincronizzazione dei threads:

- `java.util.Map`
- `java.util.LinkedList`
- `java.util.ArrayList`

Esistono però delle soluzioni per la sincronizzazione, ovvero le:

- *Thread safe collections*, sincronizzate automaticamente da Java:
  - `java.util.Vector`
  - `java.util.Hashtable`
- *Synchronized collections*
- *Concurrent collections*:
  - `java.util.concurrent`

Esistono casi in cui ci sono delle operazioni atomiche quindi che possono essere thread safe (singolarmente) ma potrebbe capitare che la combinazione di operazioni, che singolarmente sono thread safe, non lo è.

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

`isEmpty()` e `remove()` sono entrambe operazioni atomiche, ma la loro combinazione non lo è.

Le *Java Synchronized Collections* sono *conditionally thread-safe*: le operazioni individuali sulle collezioni sono safe, ma funzioni composte da più di una operazione singola possono non

esserlo.

Si può risolvere questo problema tramite la richiesta di sincronizzazione esplicita da parte del programmatore di una sequenza di operazioni.

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

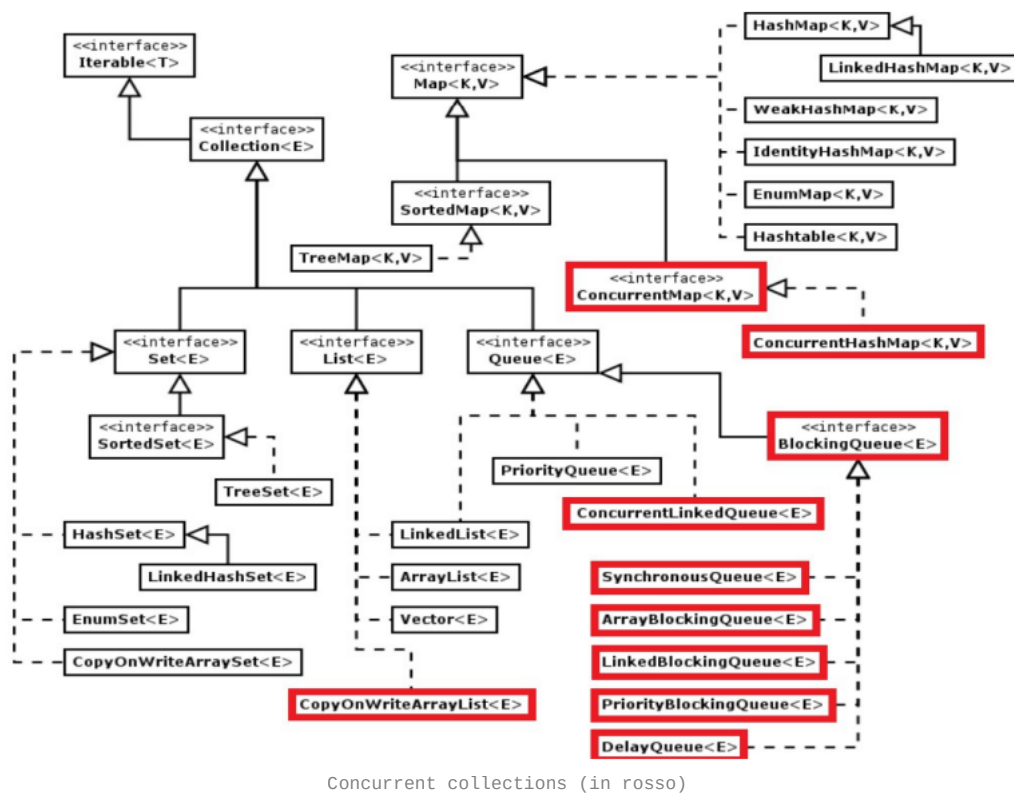
Questo è un tipico esempio di utilizzo di blocchi sincronizzati, il thread che esegue l'operazione composta acquisisce la lock sulla struttura synchList più di una volta:

- Quando esegue il blocco sincronizzato
- Quando esegue i metodi della collezione ma il comportamento corretto è garantito perchè le lock sono rientranti.

A volte può capitare che vengano sollevate delle eccezioni quando si usano questi costrutti come quando 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.



Sono un'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:

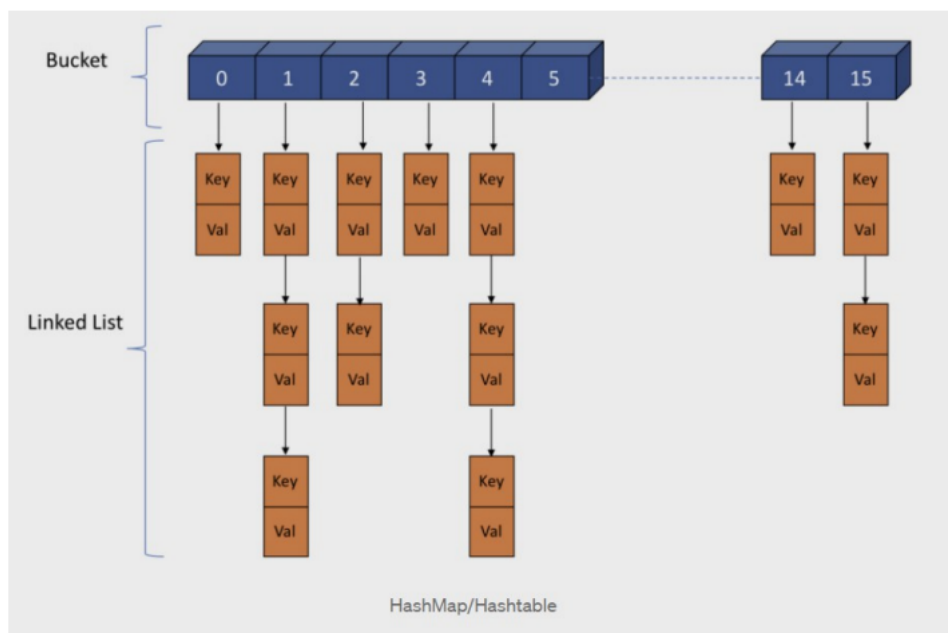
- *put-if-absent*
- *remove-if-equal*
- *replace-if-equal*

## HashMap

L'*HashMap* non è thread safe.

L'idea fondamentale è quella di usare una diversa strategia di locking che offra migliore concorrenza e scalabilità. Introduce un array di segmenti dove ogni segmento punta ad una *HashMap*. Si usa il fine grained locking: una lock per ogni segmento, lock striping e il numero di segmenti determina il livello di concorrenza.

Si possono fare modifiche simultanee, se modificano segmenti diversi: 16 o più threads possono operare in parallelo su segmenti diversi e lettori possono accedere in parallelo a modifiche.



Le concurrent collections offrono inoltre un insieme di operazioni composte atomiche, ovvero 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 garantendo inoltre la corretta sincronizzazione su tale operazione.

```
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
}
```

```
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();
 }
}
```

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`. Rappresenta un fallimento immediato dell'operatore, per evitare comportamenti non deterministici, la maggior parte delle collezioni “non-concurrent” sono fail-fast:
  - Vector, ArrayList, HashMap, ed altre

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` e 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 le modifiche operano sulla stessa copia, non sollevano `ConcurrentModificationException` e hanno comportamento fail-safe. L'iteratore considera gli elementi che esistevano al momento della costruzione dell'iteratore, anche se non è garantito: *ConcurrentHashMap*.

## InetAddress

Tra le più comuni “killer network applications” abbiamo:

- Web browser
- SSH
- Email
- Social networks
- Teleconferences (Skype, Zoom, Meet, ecc...)
- Program development environments: GIT
- Collaborative work: Overleaf
- Multiplayer games: War of Warcraft
- P2P File sharing: Bittorrent
- Blockchain: cryptocurrencies (Bitcoin), supply chain, ecc...
- Metaverse, e molte altre..

Due o più processi (non thread) in esecuzione su hosts diversi, distribuiti geograficamente sulla rete, comunicano e cooperano per realizzare una funzionalità globale.

Per *cooperazione* si intende lo scambio di informazioni utili per perseguire l'obiettivo globale, quindi implica comunicazione.

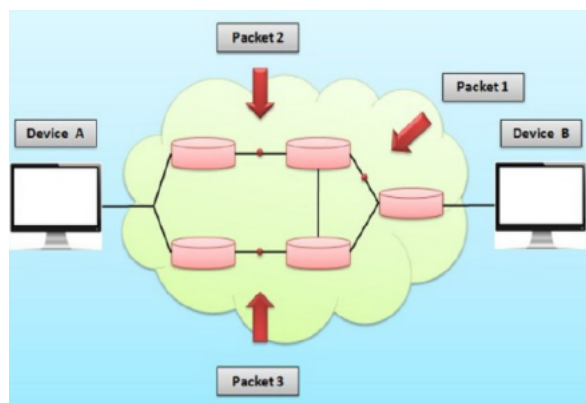
Per *comunicazione* si intende l'utilizzo di protocolli, ovvero un insieme di regole che i partners devono seguire per comunicare correttamente.

Vedremo:

- Connection-oriented: TCP (Transmission Control Protocol). Come una chiamata telefonica, si ha una connessione stabile con un canale di comunicazione dedicato tra mittente e destinatario. Principalmente sono *stream socket*.

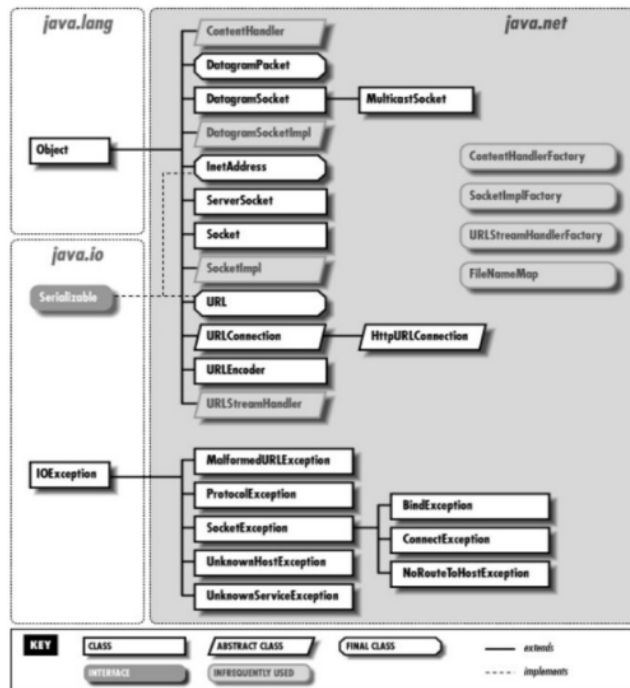


- Connectionless: UDP (User Datagram Protocol). Come l'invio di una lettera, non si stabilisce un canale di comunicazione dedicato e ogni messaggio viene instradato in modo indipendente dagli altri. Principalmente sono *datagramsocket*.



Tramite `java.net` siamo in grado di svolgere networking in Java in maniera:

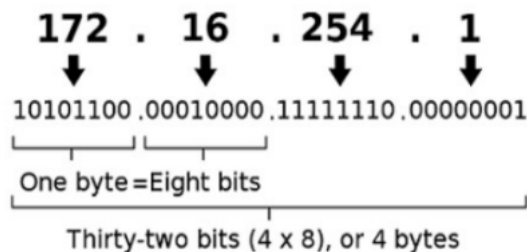
- *connection-oriented*: connessione modellata come stream, con socket asimetrici in quanto abbiamo una *Socket* class per il client e una *ServerSocket* class e una *Socket* class per il server.
- *connectionless*: socket simmetrici per client e server in quanto entrambi usano *datagramSocket* e *datagramPacket*.



Per gli indirizzi IP (Internet Protocol Address) abbiamo:

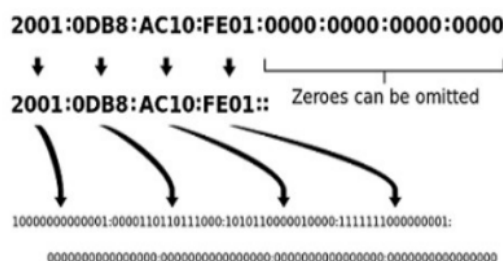
- **IPv4**: composto da 4 byte ovvero  $2^{32}$  indirizzi. Si utilizza la *dotted quad form* e ogni byte viene interpretato come un numero decimale senza segno. Alcuni indirizzi sono riservati come ad esempio i *loopback address* (127.0.0.0) e i *broadcast* (255.255.255.255).

An IPv4 address (dotted-decimal notation)



- **IPv6**: composta da 16 byte ovvero  $2^{128}$  indirizzi. Composto da 8 blocchi di 4 cifre esadecimali.

An IPv6 address (in hexadecimal)



Gli indirizzi IP semplificano l'elaborazione effettuata dai routers, ma sono poco leggibili per gli utenti della rete. La soluzione sarebbe quella di assegnare un nome simbolico unico ad ogni host della rete e di 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). I livelli della gerarchia sono separati dal punto e i nomi sono interpretati da destra verso sinistra, un nome può essere mappato a più indirizzi IP.

*Domain Name System* (DNS) traduce nomi in indirizzi IP.

La classe `public class InetAddress extends Object implements Serializable` può gestire sia indirizzi IPv4 che indirizzi IPv6 e viene usata per incapsulare in un unico oggetto di tipo *InetAddress* sia l'indirizzo IP numerico (`byte[] address`) che il nome di dominio per quell'indirizzo (`String`). La classe non contiene alcun costruttore e per costruire oggetti di tipo *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 (è necessaria una connessione di rete) e possono sollevare una `UnknownHostException` se non riescono a risolvere il nome dell'host.

```
import java.net.*;

public class FindAllIP {
 public static void main(String[] args) {
 try {
 InetAddress[] addresses = InetAddress.getAllByName("www.repubblica.it"); // lookup di tutti gli indirizzi di un host
 for (InetAddress address: addresses) {
 System.out.println(address);
 }
 }
 catch (UnknownHostException ex) {System.out.println("Could not find www.repubblica.it");}
 }
}
```

```
import java.net.*;

public class MyAddress {
 public static void main(String[] args) {
 try {
 InetAddress address = InetAddress.getLocalHost(); // restituisce l'InetAddress del local host
 System.out.println(address);
 }
 catch (UnknownHostException ex) {System.out.println("Could not find this computer address");}
 }
}
```

I metodi descritti effettuano *caching* dei nomi/indirizzi risolti. L'accesso al DNS è un'operazione potenzialmente molto costosa, i nomi vengono risolti con i dati nella cache quando possibile (di default per sempre) e possono esserci anche dei tentativi di risoluzione non andati a buon fine in cache.

La permanenza dei dati in cache è di 10 secondi se la risoluzione non ha avuto successo, spesso il primo tentativo di risoluzione fallisce a causa di un time out. Altrimenti i dati restano in cache per tempo illimitato anche se possono esserci problemi con gli indirizzi dinamici. Il controllo dei tempi di permanenza in cache viene fatto tramite

`java.security.Security.setProperty`, mentre per i tentativi non andati a buon fine si usa `networkaddress.cache.negative.ttl`.

```
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);
 }
}
```

## Factory methods

Sono 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) e non si ha nessuna garanzia sulla correttezza di hostname/IP, `UnknownHostException` viene sollevata solo se l'indirizzo è malformato:

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

## 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()`
- ECC...

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

## Spam checker

Diversi servizi monitorano gli spammers: *real-time black-hole lists* (RTBLs), ad esempio: "sbl.spamhaus.org". Essi mantengono una lista di indirizzi IP che risultano, probabilmente, degli spammers e per identificare se un indirizzo IP corrisponde ad uno spammer si svolgono i seguenti passi:

- Inversione dei bytes dell'indirizzo IP
- Concatenazione del risultato a sbl.spamhaus.org
- Esecuzione di 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 se un indirizzo IPv4 è uno spammer noto, ad esempio una query DNS su 17.34.87.207.sbl.spamhaus.org ha successo se l'indirizzo è uno spammer.

```
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.getByAddress(query);
 return true;
 }
 catch (UnknownHostException e) {return false;}
 }
}
```

## Paradigma client/server

Un *servizio* è un software in esecuzione su una o più macchine che fornisce l'astrazione di un insieme di operazioni.

Un *client* è un software che sfrutta servizi forniti dal server:

- Web client: browser
- Email client: mail-reader

Un *server* è un'istanza particolare di un servizio in esecuzione su un host come ad esempio un server Web che invia la pagina web richiesta o un mail server che consegna la posta al client.

Occorre specificare l'host tramite indirizzo IP (la rete all'interno della quale si trova l'host + l'host all'interno della rete) e la porta individua un servizio tra i tanti servizi attivi su un host.

Ogni servizio individuato da una porta è un intero tra 1 e 65535 (per TCP e UDP), non è un dispositivo fisico ma un'astrazione per individuare i singoli servizi (processi).

Le porte da 1 a 1023 sono riservate per *well-known services*.

Una *socket* è uno standard per connettere dispositivi distribuiti, diversi ed eterogenei. Si può intendere anche come una presa "standard" a cui un processo si può collegare per spedire dati, ovvero un endpoint sull'host locale di un canale di comunicazione da/verso altri hosts.

Per usufruire di un servizio, il client apre una socket individuando host e porta che identificano il servizio e invia/riceve messaggi su/da uno stream.

In JAVA si usa `java.net.Socket` che usa codice nativo per comunicare con lo stack TCP locale:

`public Socket(InetAddress host, int port) throws IOException` e 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 un'eccezione di *IO*.

`public Socket (String host, int port) throws UnknownHostException, IOException` come il precedente, l'host è individuato dal suo nome simbolico e interroga automaticamente il DNS.

Un *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
 }
 }
 }
}
```

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 e impiega molto tempo.

Per ottimizzare il programma bisogna utilizzare un diverso costruttore, ovvero `public Socket(InetAddress host, int port) throws IOException` dove 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, una volta stabilita una connessione tra client e server devono scambiarsi dei dati. La connessione è modellata come uno stream.

Si associa 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. Si hanno dei 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 e ogni valore letto dallo stream viene prelevato dal *Receive Buffer* del livello TCP.

Per interagire con il server tramite socket si usa un client implementato in Java e un server in qualsiasi altro linguaggio. Si apre un socket *sock* sulla porta su cui è attivo il servizio e si utilizzano 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:

- Test
- JSON
- XML

È possibile conoscere il formato dei dati scambiati interagendo con il server tramite il protocollo *telnet*.

Si apre una connessione sulla porta 13, verso il servizio `time.nist.gov` (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; ecc...)
- 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)

```
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 { // rilascio esplicito delle risorse (da Java 7+ non necessaria)
 if (socket != null) {
 try {socket.close();}
 catch (IOException ex) {}
 }
 }
 }
}
```

Tramite un `try-with` una certa risorsa (ad esempio file, stream, reader o socket) è chiusa “automaticamente” dopo che è stata utilizzata, 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`. È possibile usare più risorse in un blocco *try with resources*, vengono chiuse in senso inverso rispetto all'ordine con cui sono state dichiarate.

```
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 BufferedInput
 int data = bufferedInput.read();
 while(data != -1) {
 System.out.print((char) data);
 data = bufferedInput.read();
 }
 }
 }
}
```

```
}
}
```

Il seguente codice risolve il problema delle “suppressed exceptions”, le 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*.

Tramite il comando `close()` è possibile chiudere una socket da entrambe le direzioni.

Le *half closure* sono le chiusure 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`.

È possibile costruire una socket senza connessione tramite un costruttore senza argomenti e con 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`.

Ed esistono i seguenti costruttori:

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

Per ottenere informazioni da una socket si usano i metodi getter:

- Indirizzo e porta host remoto:
  - `public InetAddress getInetAddress();`
  - `public int getPort();`
- Indirizzo e porta host locale:
  - `public InetAddress getLocalAddress();`
  - `public int getLocalPort();`

Vediamo un utilizzo:

```
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.get
 }
 }
 }
}
```

```

 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);}
 }
}
}

```

## Stream Sockets

Esistono due tipi di socket TCP, lato server:

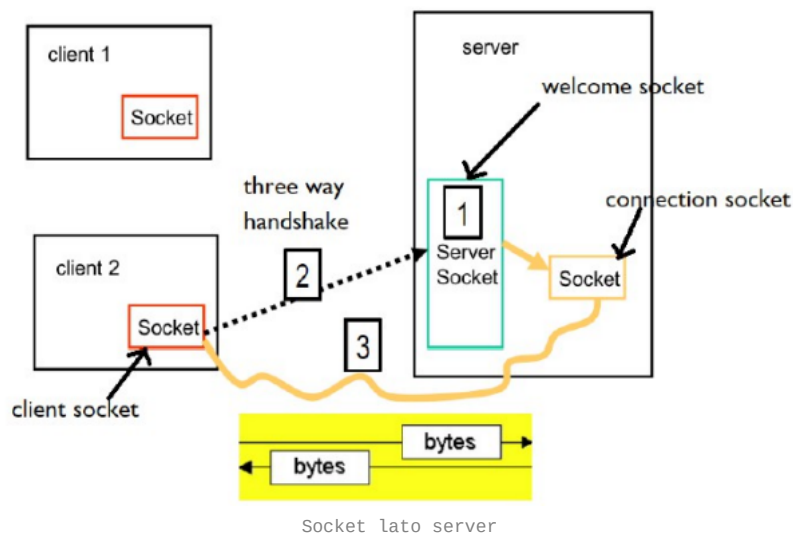
- *welcome (passive, listening) sockets*: utilizzate dal server per accettare le richieste di connessione.
- *connection (active) sockets*: connettono il server ad un particolare client e supportano lo streaming di byte tra di essi.

Il client crea una *active socket* per richiedere la connessione, il server accetta una richiesta di connessione sul *welcome socket* creando 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.

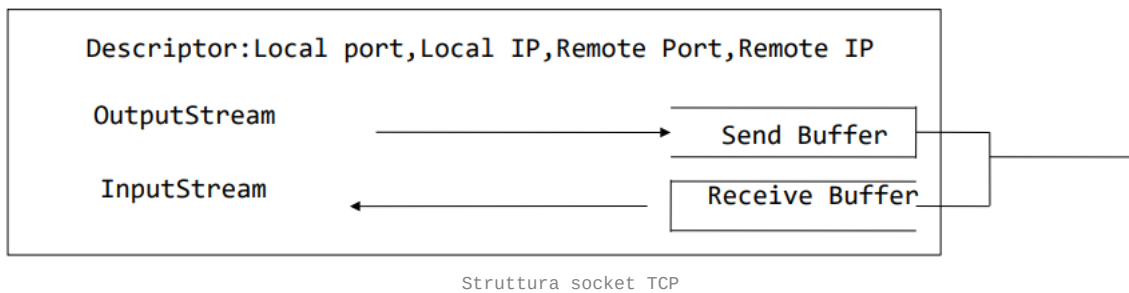
Il server pubblica un proprio servizio, gli associa una welcome socket, sulla porta remota PS all'indirizzo IPS e usa un oggetto di tipo `ServerSocket`.

Il client crea una `Socket` e la connette all'endpoint IPS+PS. La creazione della 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 e tutti i messaggi spediti dal client vengono diretti automaticamente sulla nuova socket creata.



Dopo che la richiesta di connessione viene accettata, client e server associano streams di bytes di input/output ai socket dedicati a quella connessione, poiché 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 con eventuale utilizzo di filtri associati agli stream.



Tramite `java.net.ServerSocket` importiamo i costruttori:

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

Entrambi i costruttori creano una *listening socket*, associandola alla porta *port*. Il parametro *length* è la lunghezza della coda in cui vengono memorizzate le richieste di connessione. Se la coda è piena, ulteriori richieste di connessione sono rifiutate.

Il costruttore `public ServerSocket(int port, int length, InetAddress bindAddress)` permette di collegare la 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 locale. Se voglio servire solo le richieste in arrivo dalla rete locale, associo il connection socket all'indirizzo IP locale.

Per accettare una nuova connessione dal connection socket si usa il metodo della classe `ServerSocket` `public Socket accept() throws IOException`.

Quando il processo server invoca il metodo `accept()`, pone il server in attesa di nuove connessioni in maniera *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 una nuova socket tramite cui avviene la comunicazione effettiva tra client e server.

```
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);}
 }
}
```

```
// 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();
```

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

public class DayTimeServer {
 public final static int PORT = 1313; // porte 0-1023 privilegiate
 public static void main(String[] args) {
 try (ServerSocket server = new ServerSocket(PORT)) {
 while (true) {
 try (Socket connection = server.accept()) { // si ferma qui ed aspetta, quando un client si connette restituisce un nuovo Socket
 // ...
 }
 }
 }
 }
}
```

```

 Writer out = new OutputStreamWriter(connection.getOutputStream());
 Date now = new Date();
 out.write(now.toString() + "\r\n");
 out.flush();
 connection.close(); // chiude la connessione e torna ad accettare nuove richieste
 }
 catch (IOException ex) {System.err.println(ex);} // try-with-resource: autoclose
}
}
catch (IOException ex) {System.err.println(ex);} // try-with-resource: autoclose
}
}
}

```

Nel codice di sopra, 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 e il server può gestire le richieste in modo più efficiente. L’unico problema sarebbe l’utilizzo eccessivo di risorse. Come soluzioni alternative abbiamo:

- Thread Pooling
- *ServerSocketChannels* di NIO

```

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()));
 }
 }
}

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);}
 }
}

```

```

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");
 return;
 }
 Scanner scanner = null;
 Scanner in = null;
 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());
 }
 }
 }
}

```

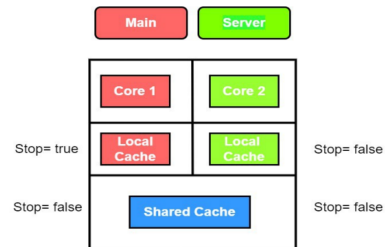
```

 }
 finally {scanner.close(); in.close();}
}
}

```

## Volatile e Atomic

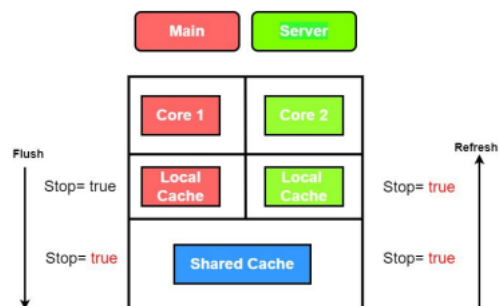
Supponendo di avere una variabile chiamata "stop" ed inizializzata a `true` nel Main, se volessimo utilizzarla all'interno di un altro programma server potremmo riscontrare qualche problema quando essa verrà modificato in uno dei due programmi.



Quando il Main aggiorna la variabile stop nella propria cache, la modifica potrebbe non venire riportata nella memoria condivisa.

Il problema riguarda la "visibilità" della modifica, non la sincronizzazione: read e write di un booleano sono atomiche.

Per risolvere questo problema modifichiamo la dichiarazione con la keyword `volatile boolean stop = false`.



L'aggiornamento ad una variabile *volatile* è sempre effettuato nella main memory e si ha un flush della cache. Il valore della variabile volatile è sempre letto dalla memoria.

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.

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.

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

```

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;}
}

```

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.

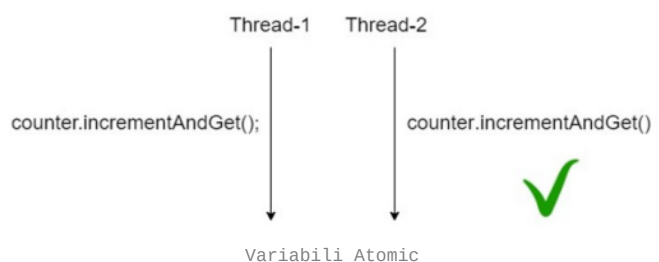
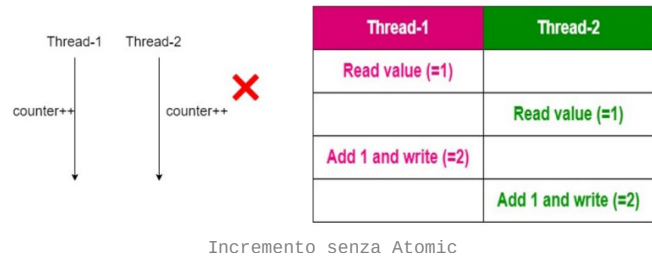
Un monitor garantisce sia sincronizzazione che visibilità.



Quindi bisogna 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"

L'incremento di una variabile (volatile o meno) non è *atomico* in quanto 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 ma una soluzione alternativa è quella di usare le variabili `Atomic`.



Per creare una variabile atomica si scrive:

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

Sono operazioni atomiche che non richiedono sincronizzazioni esplicite o lock, è la JVM che garantisce la atomicità:

- `incrementAndGet()`: incrementa di uno atomicamente
- `decrementAndGet()`: decrementa di uno atomicamente
- `compareAndSet(int expectedValue, int newValue)`

Ed esistono anche altre classi come `AtomicLong` o `AtomicBoolean`.

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

public class AtomicIntExample {
 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();
 }

 public class CounterRunnable implements Runnable {
 AtomicInteger atomicInt;

 CounterRunnable(AtomicInteger atomicInt) {
 this.atomicInt = atomicInt;
 }

 @Override
 public void run() {
 System.out.println("Counter- " + atomicInt.incrementAndGet());
 }
 }
}
```

Tramite `java.util.concurrent.atomic` possiamo accedere alle seguenti classi:



## Serializzazione: JSON e native serailization

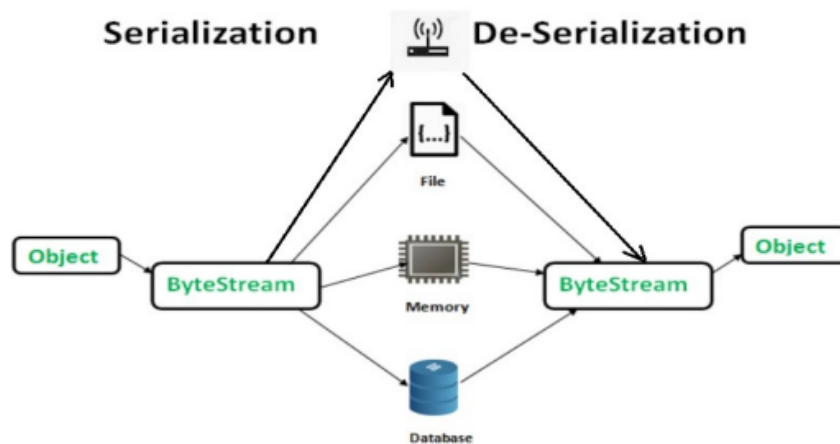
Gli oggetti esistono in memoria fino a che la JVM è in esecuzione, per la loro persistenza al di fuori della JVM, occorre creare una rappresentazione dell'oggetto indipendente dalla JVM usando meccanismi di *serializzazione*.

Ogni oggetto è caratterizzato da uno stato e da un comportamento:

- Stato: "vive" con l'istanza dell'oggetto
- Comportamento: specificato dai metodi della classe

La serializzazione effettua il *flattening* dello stato dell'oggetto, la deserializzazione ricostruisce lo stato dell'oggetto.

L'oggetto serializzato può quindi essere scritto su un qualsiasi stream di output.



L'*interoperabilità* è una caratteristica auspicabile di un formato di serializzazione che permette di non vincolare chi scrive e chi legge ad usare lo stesso linguaggio. La portabilità può limitare le potenzialità della rappresentazione.

I formati per la serializzazione dei dati che consentono l'interoperabilità tra linguaggi/macchine diverse sono:

- XML
- JSON (JavaScript Object Notation)

*JSON* è il formato nativo di JavaScript e ha il vantaggio di essere espresso con una sintassi molto semplice e facilmente riproducibile.

## JSON

JSON è un formato lightweight per l'interscambio di dati, indipendente dalla piattaforma poichè è testo, scritto secondo la notazione JSON. Non dipende dal linguaggio di programmazione ed è "self describing". semplice da capire e facilmente parsabile.

JSON è basato su due strutture:

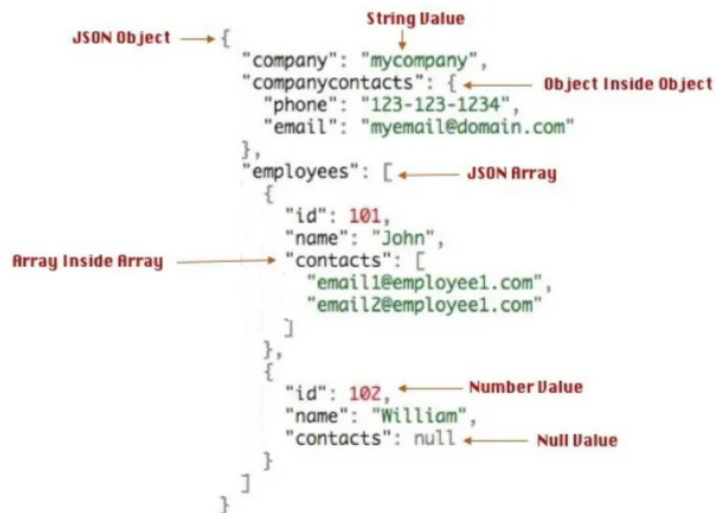
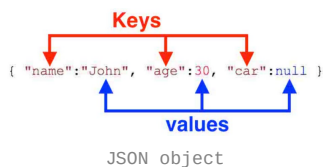
- Coppie (chiave: valore)
- Liste ordinate di valori

Una risorsa JSON ha una struttura ad albero (composizione ricorsiva di coppie e liste).

In una coppia le chiavi devono essere stringhe, mentre i tipi di dato per i valori sono:

- String
- Number (int o float)
- object (JSON object)
- Array
- Boolean
- null

Un JSON object è una serie non ordinata di coppie (nome, valore) delimitato da parentesi graffe dove le coppie sono separate da virgole.

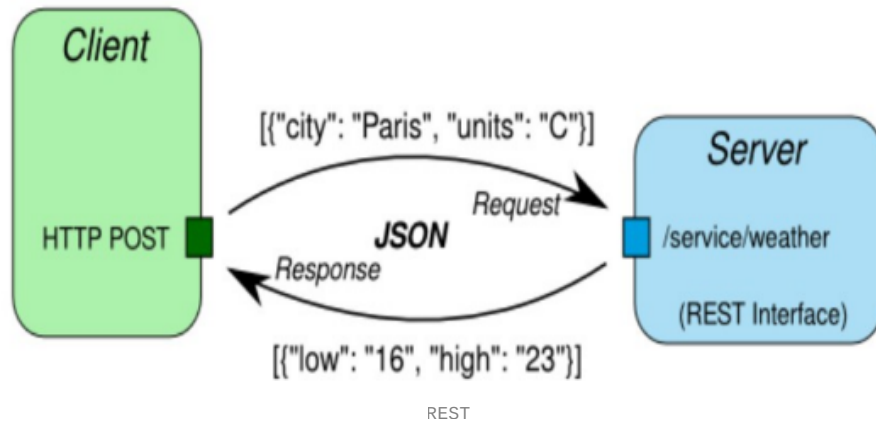


JSON object (struttura ricorsiva)

```
{
 "employees": [
 {
 "firstname": "John",
 "lastname": "Doe"
 },
 {
 "firstname": "Anna",
 "lastname": "Smith"
 },
 {
 "firstname": "Peter",
 "lastname": "Jones"
 }
]
}
```

Un JSON Array è una raccolta ordinata di valori, delimitato da parentesi quadre dove i valori sono separati da virgole. Un valore può essere di tipo String, un numero, un boolean, un JSON object o un array. Queste strutture possono essere annidate.

In un modello client server scritti in Java, client e server interagiscono mediante un'interfaccia chiamata *REST*. JSON è in genere il formato dei dati di scambio, quindi è necessaria una trasformazione da Java a JSON e viceversa.



Per la traduzione da Java a JSON esistono diverse librerie:

- GSON
- JACKSON
- JSON-Simple
- FastJSON
- ...

### GSON (Google GSON)

GSON è una libreria per serializzare/deserializzare oggetti Java da/in JSON.

Per esempio `toJson()` e `fromJson()` sono semplici metodi per la serializzazione e la deserializzazione (richiede reflection).

Si ha anche supporto per Java generics ed oggetti arbitrariamente complessi ed è possibile personalizzare la serializzazione.

Bisogna scaricare JAR ed inserirlo come libreria esterna nel progetto, scaricare GSON e importare la libreria.

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

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

I 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:** data una stringa in formato JSON restituisce un oggetto Java:
  - `fromJson(String json, Class<T> classOfT)`
  - `fromJson(JsonElement json, java.lang.reflect.Type typeOfT)`

```
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);
}
}

```

```

import com.google.gson.Gson;

public class Person {
 String name;
 int age;
 public Person(String name, int age) {
 this.name = name;
 this.age = age;
 }
}

public class ToGSON {
 public static void main(String[] args) {
 Person p = new Person("Alice", 59);
 Gson gson = new GsonBuilder().setPrettyPrinting().create(); // formatta l'output
 String json = gson.toJson(p);
 System.out.println.println(json);
 }
}

```

```

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);
 }
}

```

Adesso possiamo creare un file “restaurant.json” in cui memorizziamo la struttura JSON che stampa il codice sopra e vediamo come 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 jsonArrayOfVotes = fileObject.get("menu").getAsJsonArray();
 List <RestaurantMenuItem> menuItems = new ArrayList <RestaurantMenuItem>();
 for (JsonElement menuElement: jsonArrayOfVotes) {
 // 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);
 }
 } catch (IOException e) {
 e.printStackTrace();
 }
 }
}

```

```

 menuitems.add(restaurantel);
 }
 System.out.println("Items are " + menuitems);
}
catch (FileNotFoundException e) {e.printStackTrace();}
catch (Exception e) {e.printStackTrace();}
}
}

```

Nell'esempio qui sopra la deserializzazione avviene accedendo ai singoli campi dell'oggetto JSON ma in realtà è possibile deserializzare l'intera struttura JSON trasformandola in un solo passo nel corrispondente oggetto Java, solo che abbiamo bisogno di informazioni aggiuntive.

Occorre indicare a run time il tipo (la classe) utilizzato per la deserializzazione usando il meccanismo delle *Reflection*: hanno la capacità di analizzare ed interagire a run time con le classi, in particolare si fa utilizzo del tipo *Type* e della funzione `getType()` per determinare a run time il tipo della classe.

```

import com.google.gson.*;
import java.lang.reflect.*;
import com.google.gson.reflect.*;

public class RestaurantReflection {
 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}]]";

 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();}
 }
}

```

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 è potremmo utilizzare 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.

Si hanno quindi due possibili scenari:

- Il client/server invia al server/client un oggetto JSON che rappresenta una singola entità
- Il client/server invia al server/client un oggetto JSON che contiene la rappresentazione di uno stream di entità

```

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());
 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) {e.printStackTrace();}
 }
}

```

```
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) {e.printStackTrace();}
 }
}
```

- `JsonReader`
- `JsonWriter`

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

39

```

 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());}
}
}

```

## Serializzazione Java

Per rendere un oggetto “persistente”, l’oggetto deve implementare l’interfaccia `Serializable` che è una *marker interface*: nessun metodo, solo informazione su un oggetto per il compilatore e la JVM.

Si ha però un controllo limitato sul meccanismo di linearizzazione dei dati. Tutti i tipi di dato primitivo sono serializzabili.

Se gli oggetti implementano `Serializable` allora sono serializzabili.

L’interfaccia `Externalizable` estende `Serializable` e consente di creare un proprio protocollo di serializzazione ottimizzando la rappresentazione serializzata dell’oggetto e implementando i metodi:

- `readExternal`
- `writeExternal`

```

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;}
}

```

```

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();}
 // 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());
 }
}

```

Il metodo `readObject()` legge la sequenza di bytes memorizzati in precedenza e crea un oggetto che è l'esatta replica di quello originale. Inoltre può leggere qualsiasi tipo di oggetto, è necessario però 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) e si percorre l'albero delle superclassi fino alla prima superclasse non serializzabile. Per quella classe viene invocato il costruttore.

Gli oggetti contenenti riferimenti specifici alla JVM o al SO (Java native class) non sono serializzabili.

Thread, OutputStream, Socket e File non possono essere ricreati, perchè contengono riferimenti specifici al particolare ambiente di esecuzione.

Altri elementi che non possono essere serializzati sono:

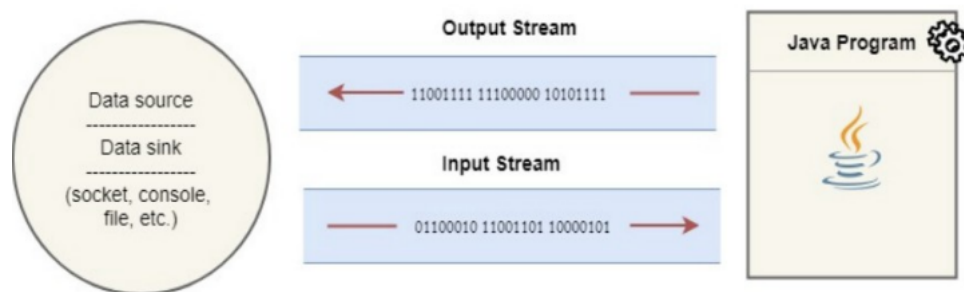
- Le variabili marcate come `transient`: ad esempio le variabili che non devono essere scritte per questioni di privacy.
- Le variabili statiche: sono associate alla classe e non alla specifica istanza dell'oggetto che si sta serializzando.

Tutti i componenti di un oggetto devono essere serializzabili: se ne esiste uno non serializzabile e non transient si solleva una `notSerializableException`.

## Java NIO

Come abbiamo visto fino ad ora, tramite Java Stream Oriented IO, i dati sono scritti/letti su/da uno stream (ogni stream è unidirezionale e bloccante). I byte sono scritti/letti sullo stream un byte alla volta ma è possibile una bufferizzazione dei dati scritti/letti su/dallo stream, tramite:

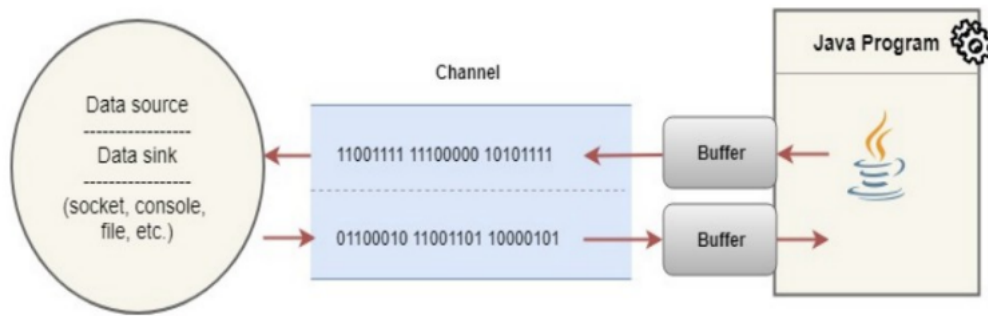
- `BufferedInputStream` / `BufferedOutputStream`: un buffer allocato nello heap della JVM da cui la JVM preleva i dati e poi li passa alla applicazione (gestito dalla JVM).
- *Array di byte*: a carico del programmatore, allocato sullo heap.



Java Stream Oriented IO

Tramite *Java NIO Channels* i dati sono trasferiti sul dispositivo mediante un canale e vengono scritti/letti in un buffer (interfaccia tra programma e canale), il programma opera sul buffer non sul canale.

I canali sono bidirezionali e possono essere non bloccanti.



Java NIO Channels

I *Channel* sono bidirezionali e ognuno di essi può leggere dal dispositivo e scrivere sul dispositivo (più vicino alla implementazione reale del sistema operativo).

Tutti i dati sono gestiti tramite oggetti di tipo *Buffer*: non si scrive/legge direttamente su un canale ma si passa da un buffer. Essi possono essere bloccanti o meno (utili soprattutto per comunicazioni in cui i dati arrivano in modo incrementale). Hanno minore importanza per letture da file, *FileChannel* sono bloccanti.

#### Vantaggi:

- Definizione di primitive "più vicine" al livello del sistema operativo, con aumento di performance.
- In generale, migliori prestazioni

#### Svantaggi:

- Primitive a più basso livello di astrazione: perdita di semplicità ed eleganza rispetto allo stream-based IO.
- Maggior difficoltà nella messa a punto del programma.
- 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.

#### NIO (Java 1.4):

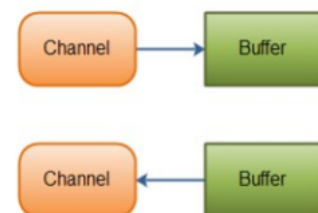
- **Buffers**
- **Channels**
- **Selectors**

#### NIO.2 (Java 1.7, implementato in alcuni package contenuti nel package NIO):

- new File System API
- asynchronous I/O
- update

Come già detto, i costruttori base di NIO sono:

- Canali e Buffers: L'IO standard è basato su stream di byte o di caratteri, con filtri. Con NIO tutti i dati da e verso dispositivi devono passare da un canale (simile ad uno stream di `java.io`) e tutti i dati inviato a o letti da un canale devono essere memorizzati in un buffer.



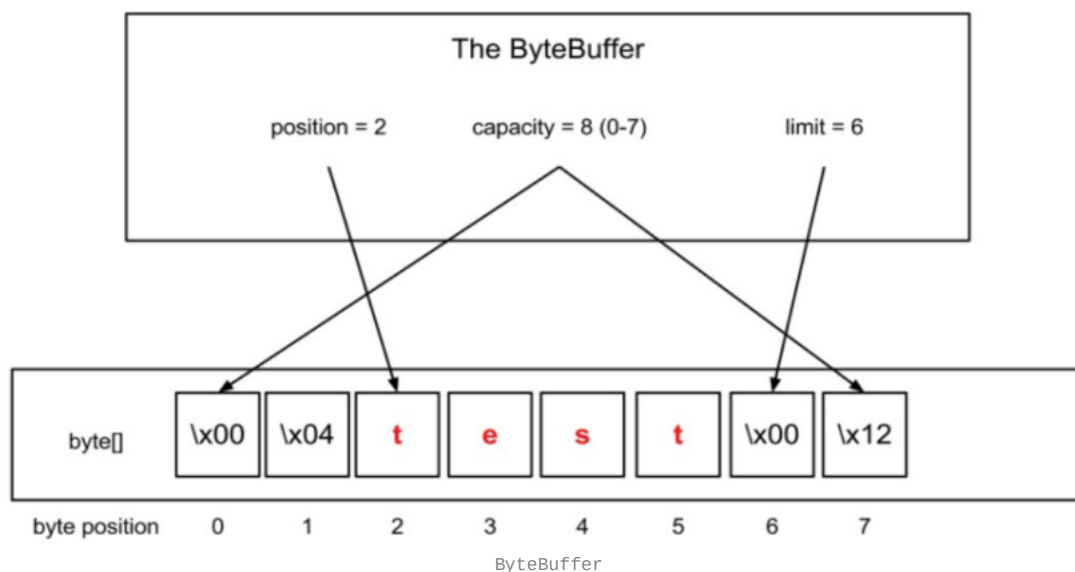
- Selector: oggetto in grado di monitorare un insieme di canali. Intercetta eventi provenienti da diversi canali: dati arrivati, apertura di una connessione, ecc... Un selector fornisce la possibilità di monitorare più canali con un unico thread.

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

I *Channel* collegano da/verso i dispositivi esterni e sono bidirezionali. A differenza degli stream, non si scrive/legge mai direttamente da un canale. Le interazioni con i canali sono le seguenti:

- Trasferimento dati dal canale nel buffer, il programma accede al buffer.
- Il programma scrive nel buffer, il contenuto del buffer viene trasferito nel canale.

## Buffers



Un oggetto di tipo *Buffer* è composto da uno spazio di memorizzazione (byte buffer) e un insieme di variabili di stato. Un `ByteBuffer` "backed" da un `byte array`.

Supponiamo di eseguire il seguente codice:

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

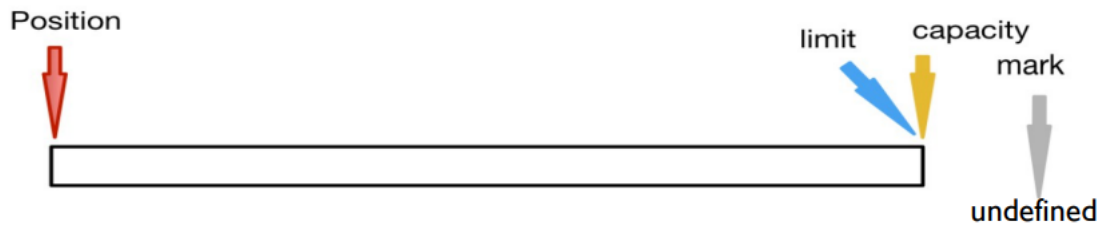
Otteniamo due diversi `ByteBuffer` che si riferiscono al solito `bytearray` ma il loro contenuto è diverso.

Le variabili di stato di un `ByteBuffer` sono:

- **Capacity**: è il massimo numero di elementi del Buffer. Viene definita al momento della creazione del Buffer e non può essere modificata. Viene lanciata `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. Viene aggiornato implicitamente dalle operazioni sul buffer effettuate dal programma o dal canale.
- **Position**: come un file pointer per un file ad accesso sequenziale, è la posizione in cui bisogna scrivere o da cui bisogna leggere. Viene aggiornata implicitamente dalle 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` viene lanciata `java.nio.InvalidMarkException`.

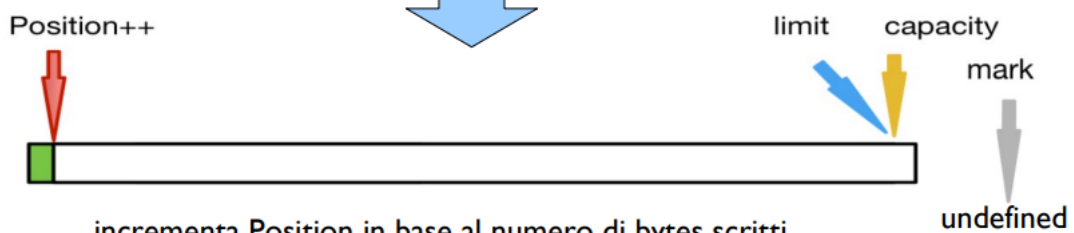
Valgono sempre le seguenti relazioni:

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



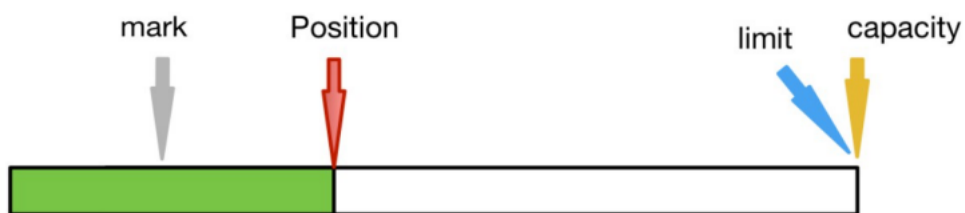
stato iniziale del Buffer: Limit=Capacity-1, Position=0, Mark=undefined

SCRITTURA



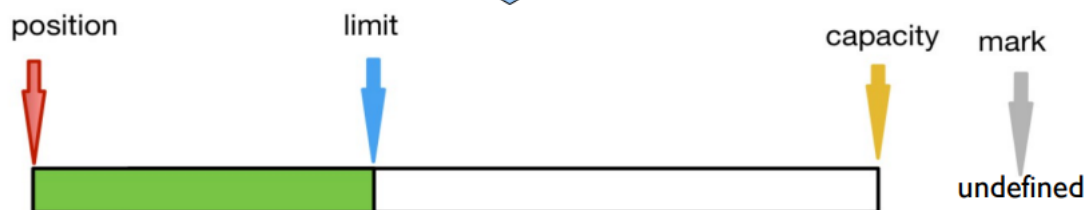
incrementa Position in base al numero di bytes scritti

Scrittura

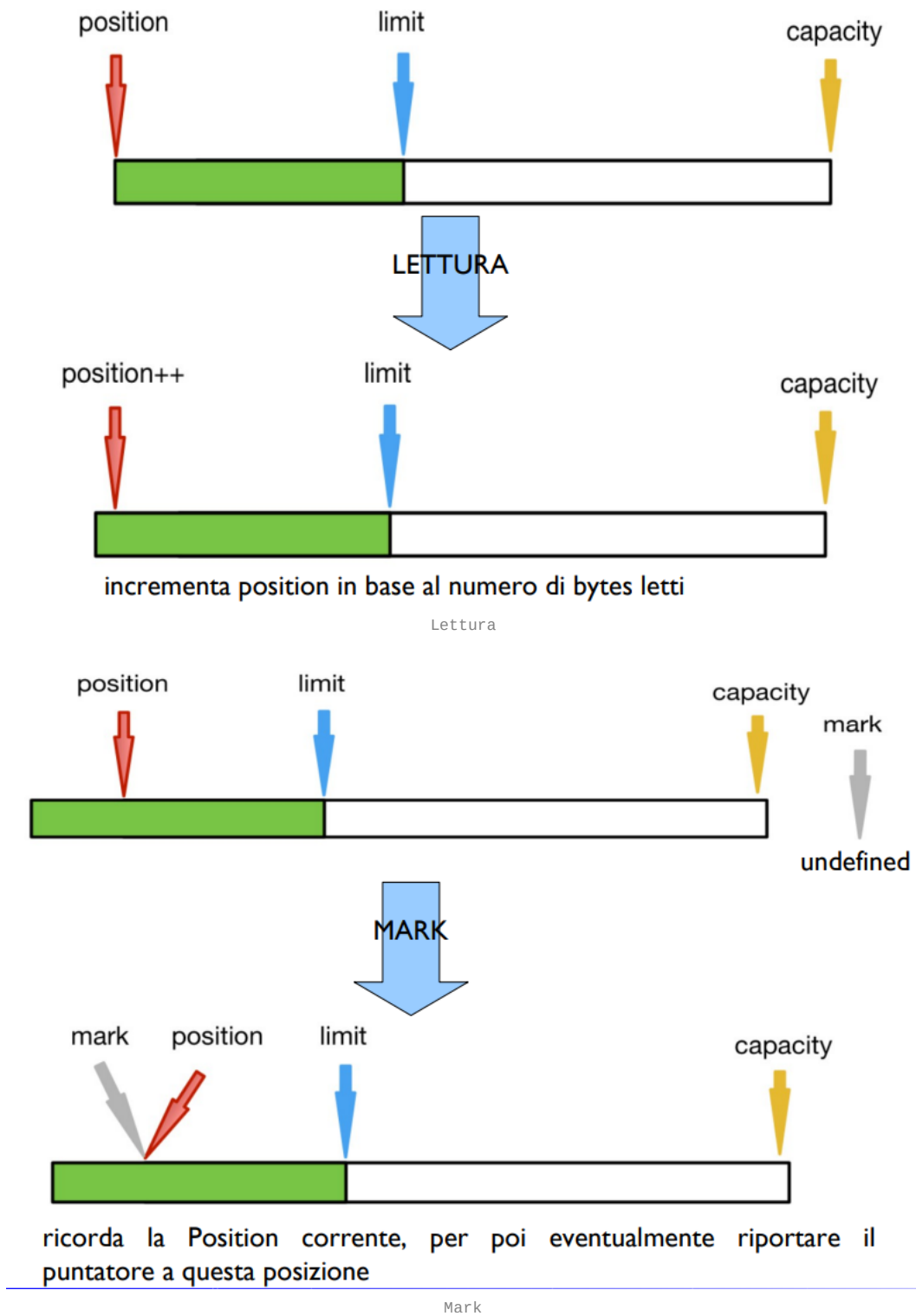


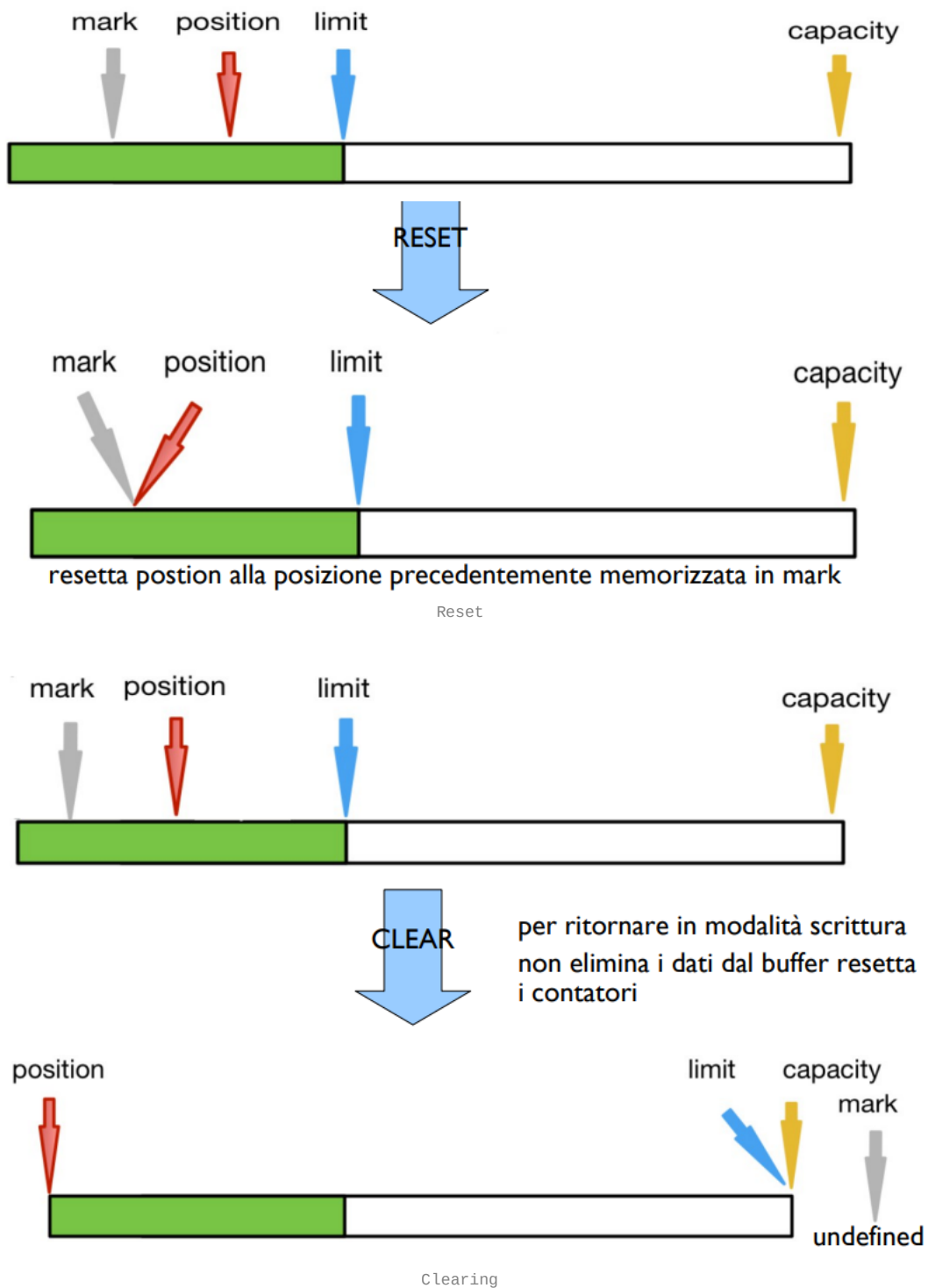
FLIP

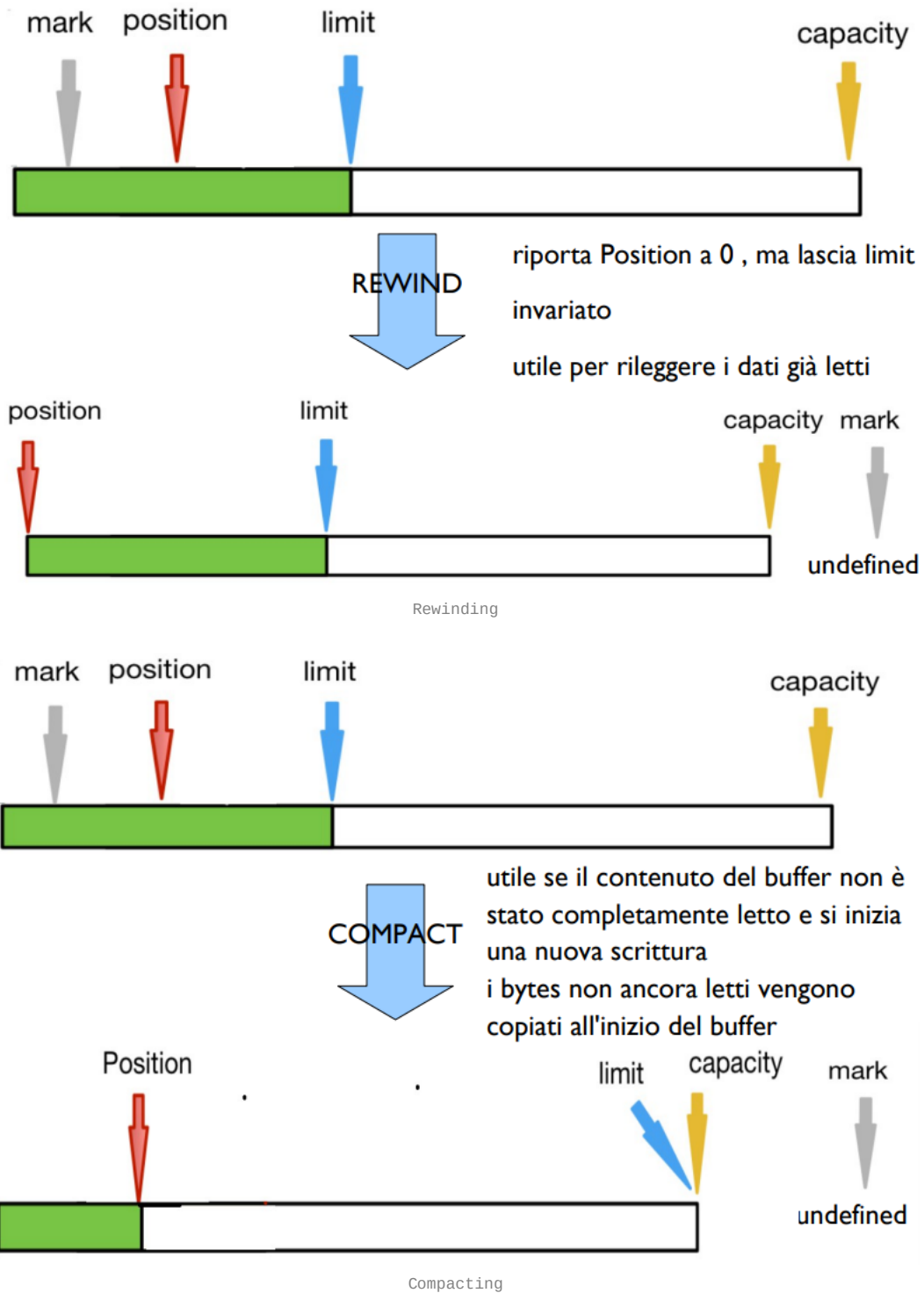
predisporre il buffer alla lettura,  
dopo la scrittura



Flipping

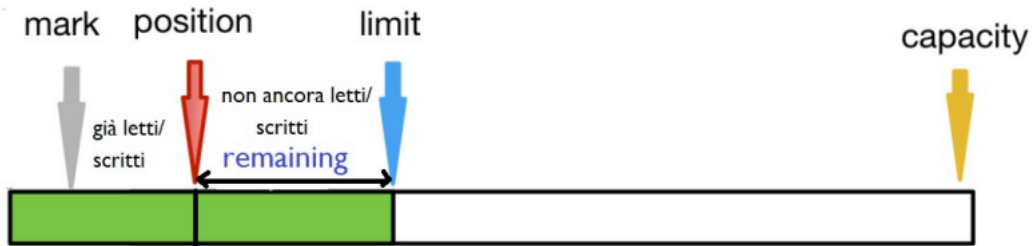






Altri metodi utili sono:

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



```
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]

 System.out.println(byteBuffer1.getChar()); // a
 System.out.println(byteBuffer1); // 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();
 System.out.println(byteBuffer1); // java.nio.HeapByteBuffer[pos=0 lim=8 cap=10]

 System.out.println(byteBuffer1.getInt()); // 1
 System.out.println(byteBuffer1.getInt()); // 2
 System.out.println(byteBuffer1); // java.nio.HeapByteBuffer[pos=8 lim=8 cap=10]

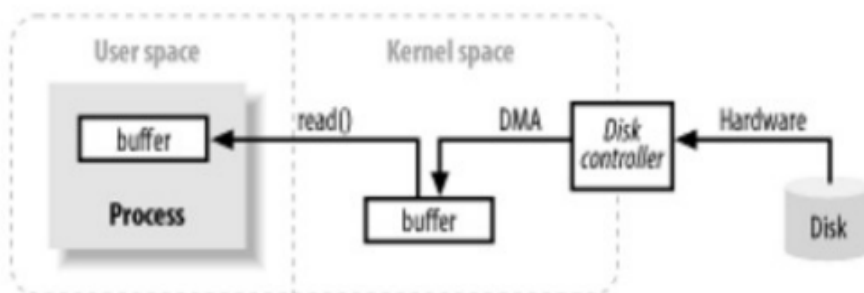
 byteBuffer1.rewind(); // rewind prepara a rileggere i dati che sono nel buffer, ovvero resetta position a 0 e non modifica limit
 System.out.println(byteBuffer1); // java.nio.HeapByteBuffer[pos=0 lim=8 cap=10]

 System.out.println(byteBuffer1.getInt()); // 1

 byteBuffer1.mark();
 System.out.println(byteBuffer1.getInt()); // 2
 System.out.println(byteBuffer1); // java.nio.HeapByteBuffer[pos=8 lim=8 cap=10]

 byteBuffer1.reset();
 System.out.println(byteBuffer1); // java.nio.HeapByteBuffer[pos=4 lim=8 cap=10]

 byteBuffer1.clear();
 System.out.println(byteBuffer1); // java.nio.HeapByteBuffer[pos=0 lim=10 cap=10]
 }
}
```



Interazione JVM/Sistema operativo

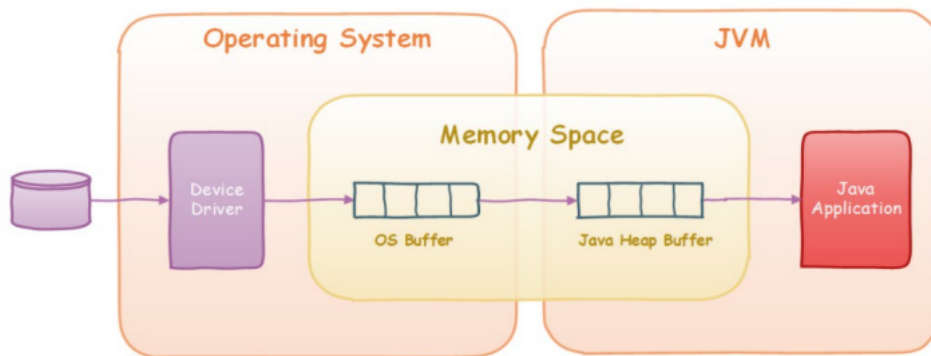


La JVM esegue una `read()` da stream o canale e provoca una system call (native code). Il kernel invia un comando al disk controller, il disk controller, via DMA (senza controllo della CPU) scrive direttamente un blocco di dati nel kernel space e i dati sono copiati dal kernel space nello user space (all'interno della JVM).

La gestione ottimizzata di questi buffer può comportare un notevole miglioramento della performance dei programmi!

Un *NON-Direct Buffer* crea sullo heap un oggetto Buffer, quindi si ha una doppia copia dei dati:

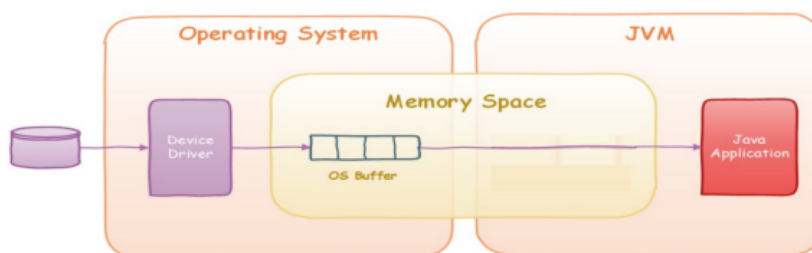
- Nel buffer del kernel
- Nel buffer sullo heap della JVM



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

NON-Direct Buffer

Un *Direct Buffer* serve per 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 del sistema operativo. I vantaggi sono la migliore performance, gli svantaggi sono il maggiore costo di allocazione/deallocazione e se il buffer non è allocato sullo heap il Garbage Collector non può recuperare memoria.



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

Direct Buffer

## Channels

Per quanto riguarda la scrittura, se il canale è utilizzato solo in output, possiamo crearlo partendo da un `FileOutputStream`, usando classi "ponte" tra stream e channels, nel seguente modo:

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

Successivamente si crea il Buffer sul canale:

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

Per scrivere il messaggio nel buffer:

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

Per scrivere sul canale:

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

Notare che occorre predisporre il Buffer in lettura, dopo che i dati sono stati trasferiti.

Per quanto riguarda la lettura, se il canale è utilizzato solo in input, possiamo crearlo partendo da un `FileInputStream`, usando classi "ponte" tra stream e channels, nel seguente modo:

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

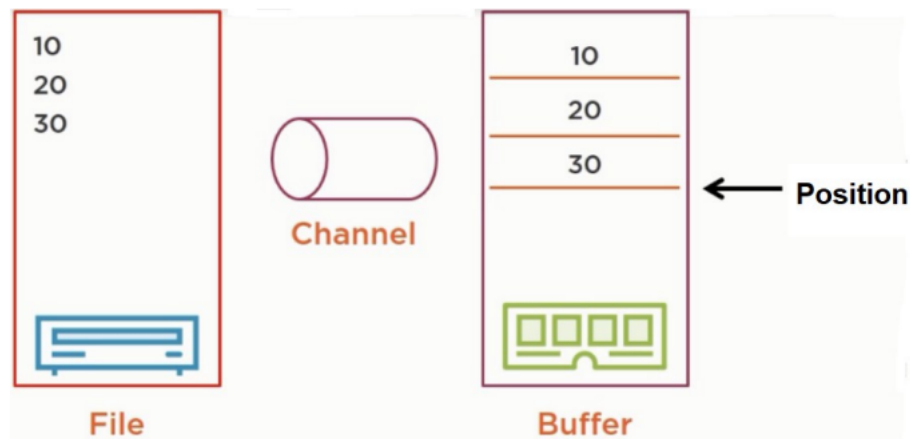
Successivamente si crea il Buffer sul canale:

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

E per leggere dal canale al buffer basta semplicemente:

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



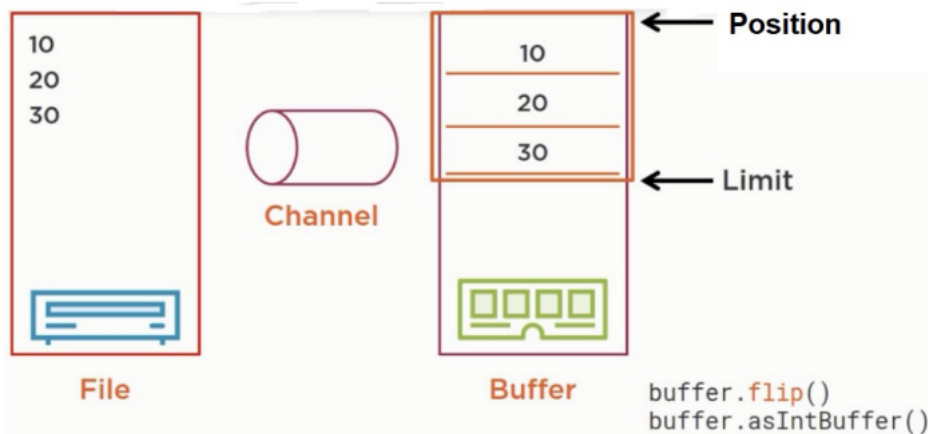
Questa foto raffigura la lettura di tre interi dal File al Buffer tramite il Channel, dopo la lettura dal file, Position indica l'ultima posizione letta dal file. Ora il programma deve leggere i dati dal Buffer.

Una possibilità (errata) sarebbe usare la `rewind()` che riporta il cursore all'inizio del Buffer ma occorre tenere traccia di dove si trovava Position prima della `rewind()`.

```
buffer.rewind();
buffer.asIntBuffer(); // interpreta i byte del Buffer come interi
```

Ma l'operazione corretta è la `flip()` che setta la `Limit` alla `Position` corrente, viene memorizzata quale è la parte significativa del `Buffer`, quindi si comporta come la `rewind()` e riporta `Position` a 0.

```
buffer.flip();
buffer.asIntBuffer();
```



I channel sono connessi a descrittori di file/socket gestiti dal sistema operativo. L'API per i channel utilizza molte interfacce Java, le implementazioni utilizzando principalmente codice nativo.

Channel 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. È possibile un "trasferimento diretto" da Channel a Channel se almeno uno dei due è un Channel.

Gli oggetti di tipo `FileChannel` possono essere creati direttamente utilizzando `FileChannel.open`, dichiarando il tipo di accesso al channel (READ/WRITE o entrambi):

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

`FileChannel` API è a basso livello, si hanno solo metodi per leggere e scrivere bytes (lettura e scrittura richiedono come parametro un `ByteBuffer`).

I `FileChannel` sono bloccanti e thread safe, infatti più thread possono lavorare in modo consistente sullo stesso channel e alcune operazioni possono essere eseguite in parallelo (esempio: read), altre vengono automaticamente serializzate.

Su `SocketChannel` sono possibili operazioni non bloccanti.

```
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 [] args) 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();
 }

 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 ancor // scritti nel file di output
 buffer.flip();
 while (buffer.hasRemaining()) {dest.write (buffer);}
 }
 }

 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()
 }
 }
}

```

Il metodo `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.

Il metodo `flip()` converte il buffer da modalità scrittura a modalità lettura.

Il metodo `write()` preleva alcuni dati dal buffer e li scarica sul canale. Non necessariamente scrive tutti i dati presenti nel Buffer sul canale.

Il metodo `hasRemaining()` verifica se esistono elementi nel buffer nelle posizioni comprese tra `position` e `limit`.

Esistono anche altri due metodi:

- `FileChannel.transferTo()`
- `FileChannel.transferFrom()`

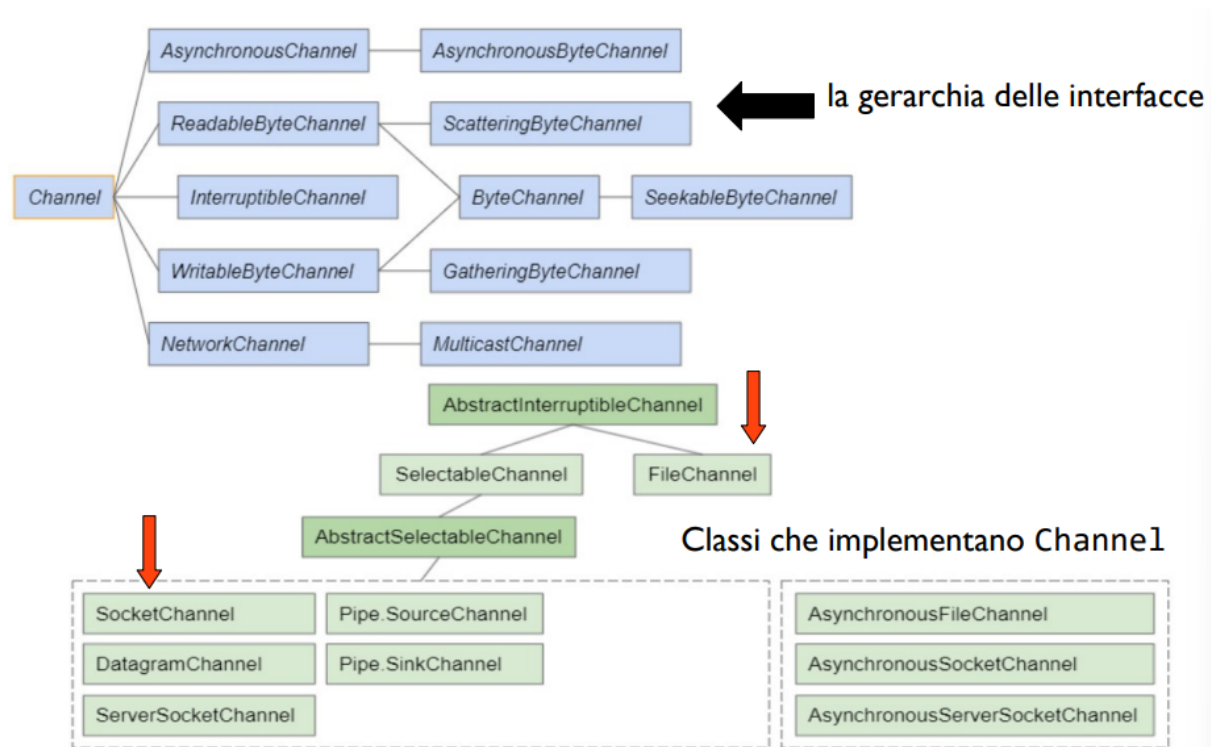
Ad esempio:

```

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);

```

## Channel Multiplexing

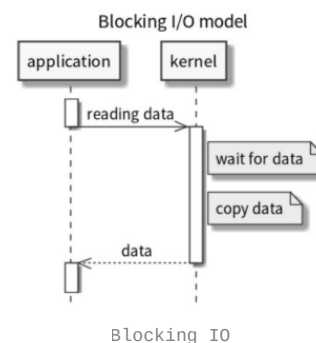


Il modello *blocking IO* permette di compiere operazioni bloccanti su stream: l'applicazione esegue una chiamata di sistema e si blocca fino a che tutti i dati sono ricevuti nel kernel, e copiati dal kernel space alla memoria dell'applicazione.

Il metodo `read()` si blocca fino a quando non è stato letto un byte, un vettore di byte, un intero, ecc...

Il metodo `accept()` si blocca fino a che non viene stabilita una nuova connessione.

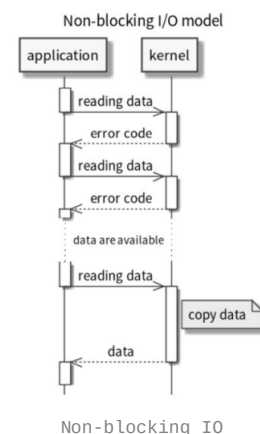
Il metodo `write(byte[] buffer)` si blocca fino a che tutto il contenuto del buffer non è stato copiato sulla periferica di I/O.



Nel modello *non-blocking IO* la chiamata di sistema restituisce il controllo all'applicazione prima che l'operazione richiesta sia stata "pienamente soddisfatta". Ci sono due scenari possibili:

- Vengono restituiti i dati disponibili, o una parte di essi
- L'operazione di I/O non è possibile: codice errore o valore `null`

Per completare l'operazione bisogna effettuare system-call ripetute, fino a che l'operazione può essere effettuata. Il non-blocking IO è possibile per channel associati a socket.



Un channel associato ad una socket (*socket channel*) TCP “combina” una socket con un canale di comunicazione bidirezionale:

- Scrive e legge da una socket TCP.
- Estende la classe `AbstractSelectableChannel` e da questa mutua la capacità di passare dalla modalità bloccante a quella non bloccante.
- In modalità bloccante il funzionamento è simile a quello delle stream socket, ma con interfaccia basata su buffers.

Le due classi di socket channel sono: `SocketChannel` e `SocketServerChannel`.

Ognuno di essi è associato ad un oggetto Socket della libreria `java.net`.

La socket può essere reperita mediante il metodo `socket()`, applicato al 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...
}
```

Ogni `SocketChannel` è associato ad un oggetto di tipo `Socket`. La creazione di un `SocketChannel` può avvenire in due modi:

- Implicita: creato se si accetta una connessione su un `ServerSocketChannel`
- Esplicita: lato client, quando si apre una connessione verso un server, mediante l'operazione di `connect()`:

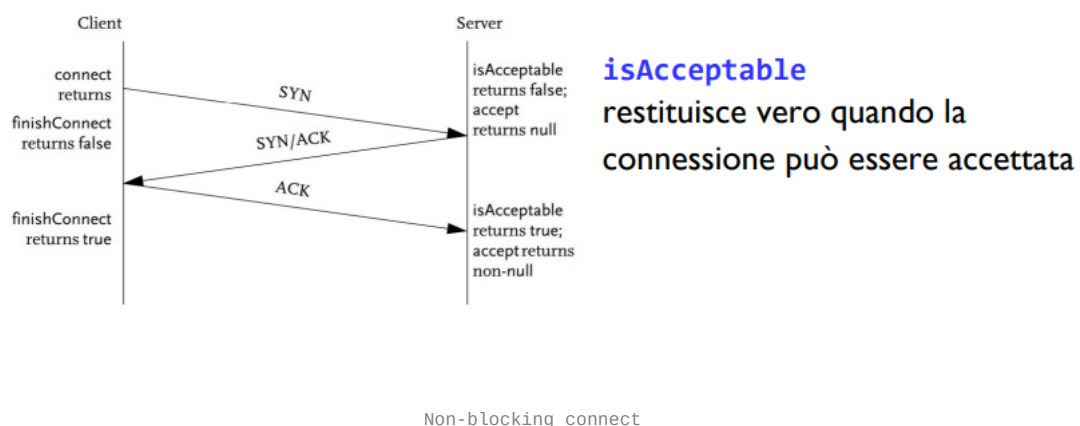
```
SocketChannel socketChannel = SocketChannel.open();
socketChannel.connect(new InetSocketAddress("www.google.it", 80));
```

Per selezionare la modalità blocking/non-blocking si usa il seguente comando:

`SocketChannel.configureBlocking(false/true)`.

La modalità non-blocking, lato client, è significativa ad esempio nel caso in cui un'applicazione deve gestire l'interazione con l'utente, mediante gui, e l'apertura della socket.

Nella modalità non-blocking, la connessione avviene nel seguente modo:



Può restituire il controllo al chiamante prima che venga stabilita la connessione. Si usa `finishConnect()` per controllare la terminazione dell'operazione:

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

Se l'ultima fase del three way handshake non è completo quando il client effettua la read, la read restituirà 0 valori nel buffer.

Se si toglie il while dall'esempio precedente, viene sollevata

`java.nio.channels.NotYetConnectedException`.

| <u>Accept()</u>                                                              | <u>Write()</u>                                                                                                                    | <u>Read()</u>                                                                        |
|------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| Blocking: si blocca finché non arriva una richiesta di connessione           | Blocking: si blocca finché la scrittura dei dati nel buffer non è completata                                                      | Blocking: si blocca in attesa di byte da leggere                                     |
| Non-blocking: controlla se c'è una richiesta da accettare e ritorna comunque | Non-blocking: tenta di scrivere i dati nella socket, ritorna immediatamente, anche se i dati non sono stati completamente scritti | 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).

Se si ha un solo thread per tutti client:

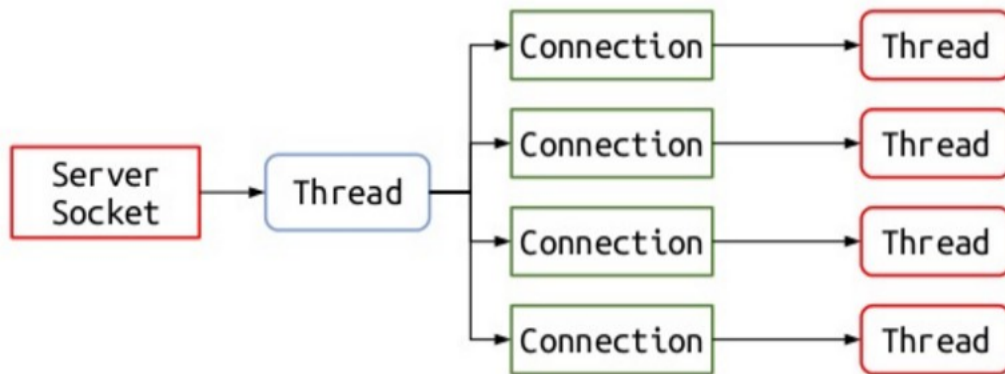
- Scalabilità: nulla, in ogni istante, solo un client viene servito.
- Accept latency: alta, il "prossimo" cliente deve attendere fino a che il primo cliente termina la connessione.
- Reply latency bassa: tutte le risorse a disposizione di un singolo client.
- Efficiency: buona, il server utilizza esattamente le risorse necessarie per il servizio dell'utente.

Adatto quando il tempo di servizio di un singolo utente è garantito che rimanga basso.

Se si ha 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.

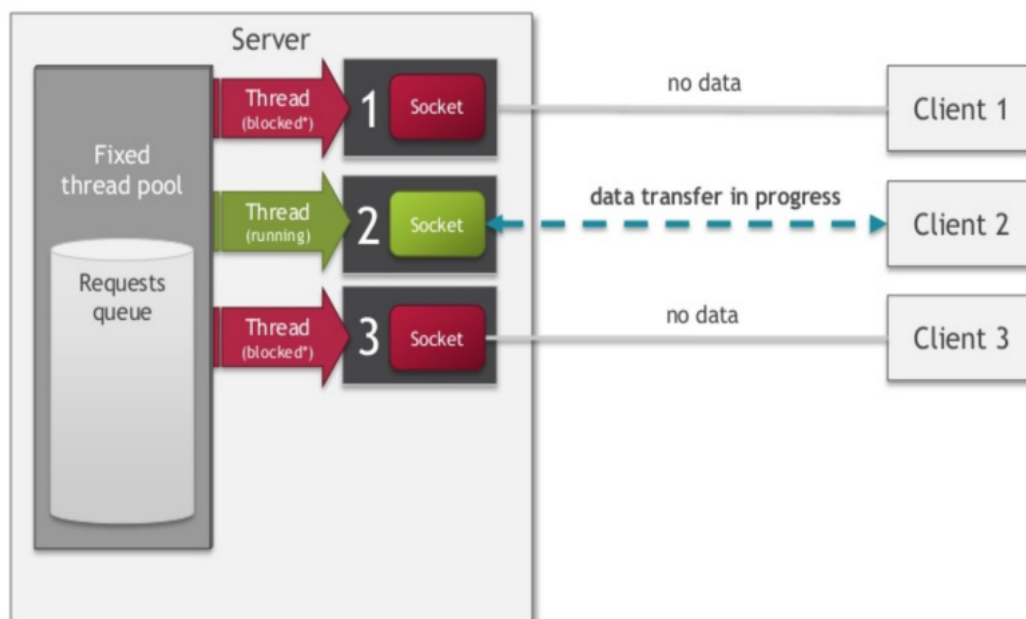
- Ragionevole uso di CPU e RAM per centinaia di connessioni, se aumenta, il tempo di reply può non essere accettabile
- Efficiency: bassa.
  - Ogni thread può essere bloccato in attesa di I/O, ma utilizza risorse come la RAM



Un thread per ogni connessione

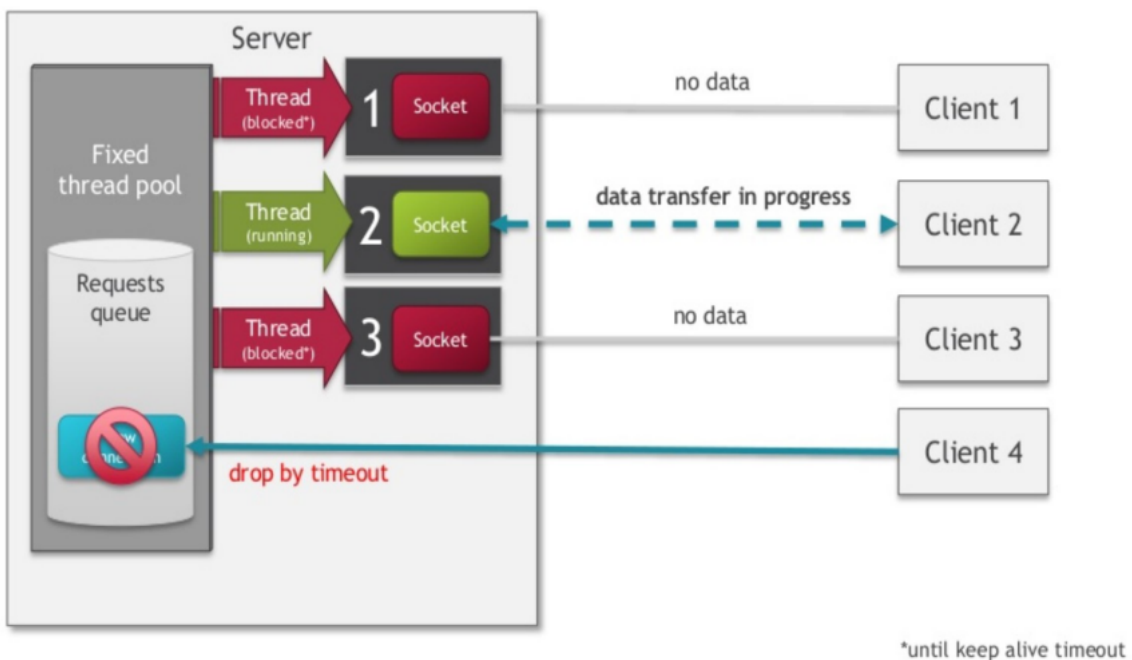
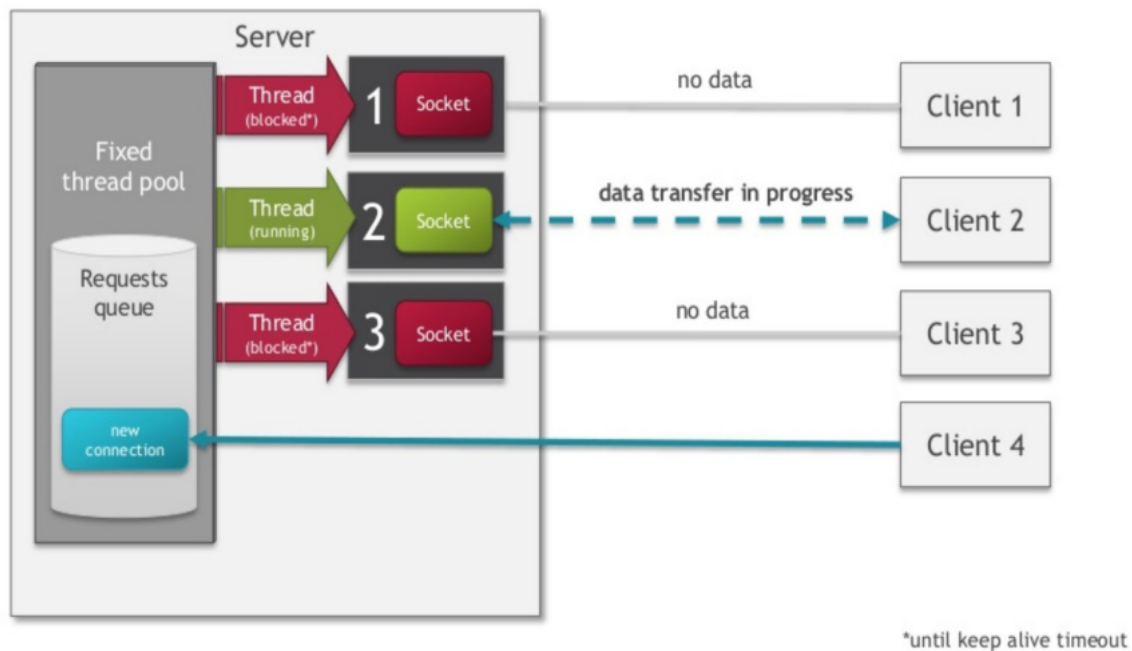
Attivazione di un thread per ogni connessione, de-attivazione a fine servizio. Quando un server monitora un grande numero di comunicazione si hanno problemi di scalabilità: il tempo per il cambio di contesto può aumentare notevolmente con il numero di thread attivi (maggior parte del tempo impiegata in context switching).

Per esempio, usando una `FixedThreadPool`:



\*until keep alive timeout



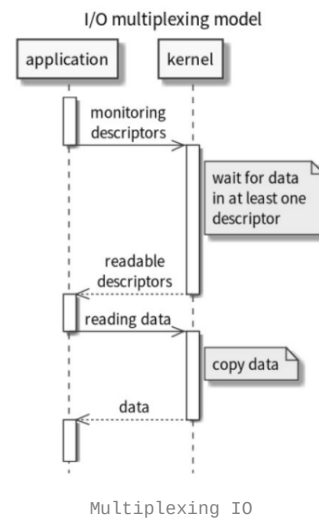


Con un numero costante di thread, usando una Thread Pool:

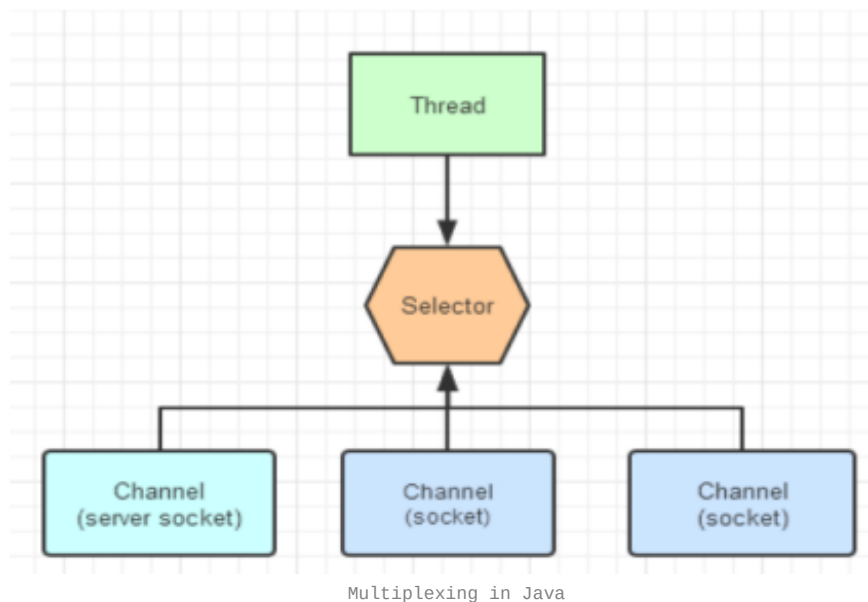
- Scalabilità: limitata al numero di connessioni che possono essere supportate.
- Accept latency: bassa fino ad un certo numero di connessioni.
- Reply latency: bassa fino al numero massimo di thread fissato, degrada se il numero di connessioni è maggiore.
- Efficiency: trade-off rispetto al modello precedente.

Il modello *multiplexed IO* è simile al non-blocking IO ma con notifiche bloccanti. L'applicazione registra "descrittori" delle operazioni di I/O a cui è interessato ed 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



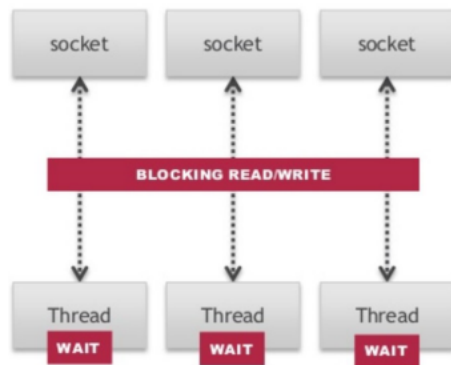
In Java il multiplexing funziona nel seguente modo:



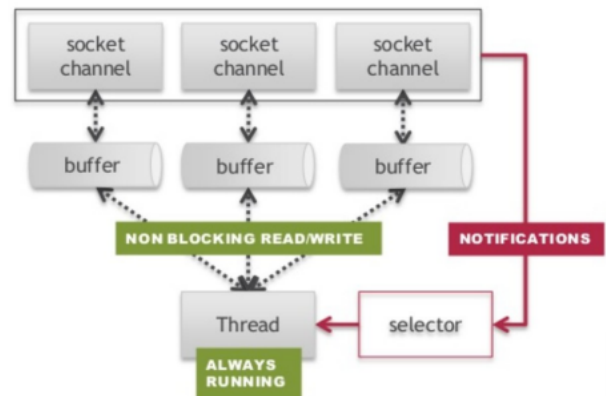
Un *selettore* (Selector) è un componente che esamina uno o più NIO Channels, e determina quali canali sono pronti per leggere e scrivere. Più connessioni di rete sono così gestite mediante un unico thread e consente di ridurre il thread switching overhead e l'uso di risorse per thread diversi. Si può usare anche insieme al multithreading.

Si usa quindi un solo thread che gestisce un numero arbitrario di sockets. Non si ha un thread per connessione, ma un numero ridotto di threads (numero basso anche con migliaia di sockets, il caso limite è 1). Si ottiene quindi un miglioramento delle performance e della scalabilità ma l'architettura è più complessa da capire e da implementare.

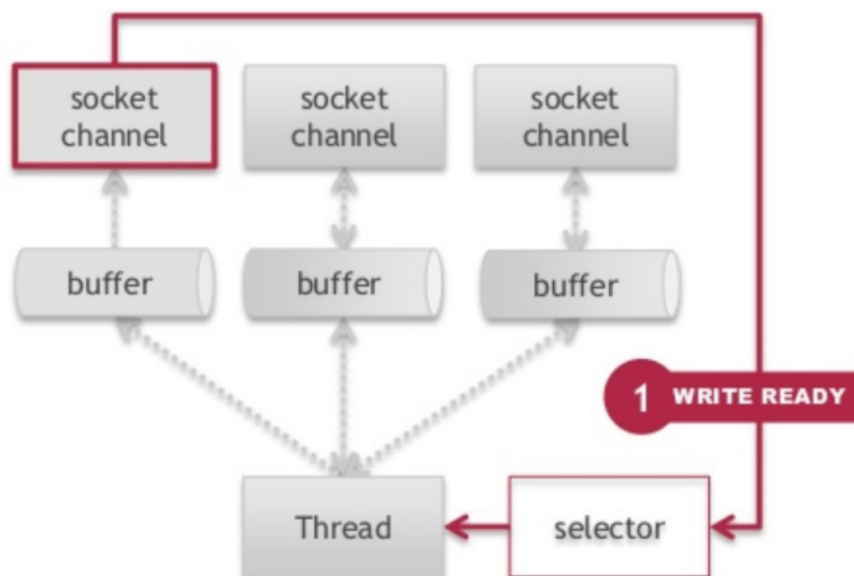
## IO (blocking)

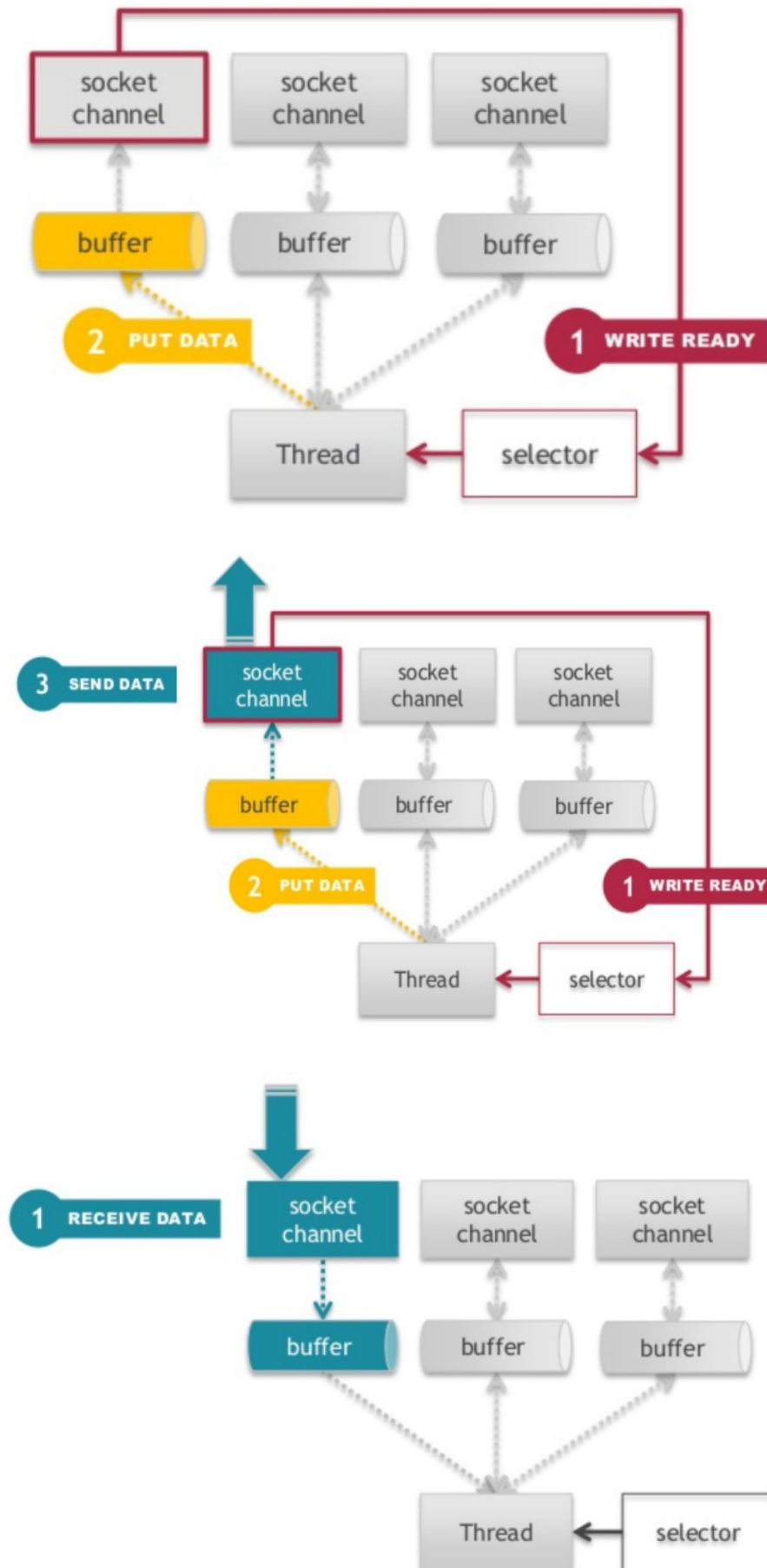


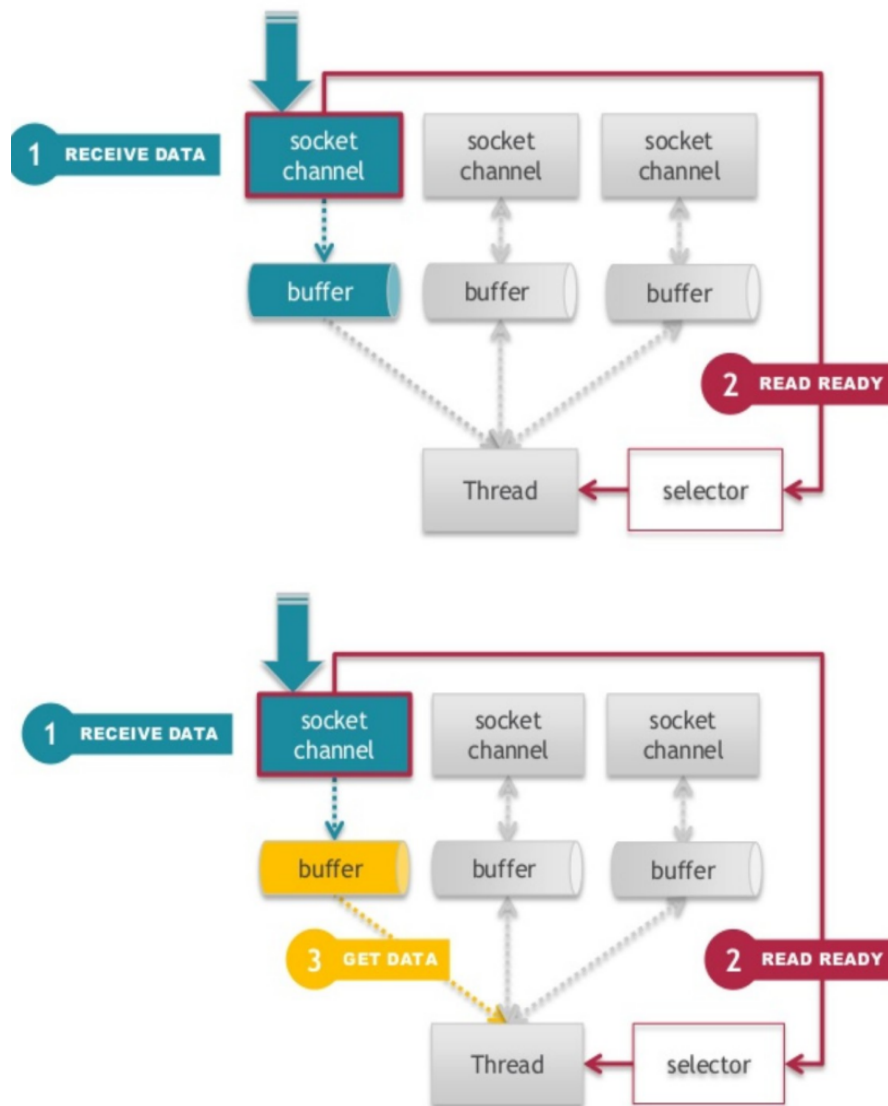
## NIO (non-blocking)



Per esempio:







Il componente base per il multiplexing è quindi il Selector che ha la seguente sintassi:

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

Permette di selezionare un `SelectableChannel` che è pronto per operazioni di rete: accept, read, write, connect. Lo stesso thread gestisce più eventi che possono avvenire simultaneamente.

I `SelectableChannel` sono:

- `ServerSocketChannel`
- `SocketChannel`
- `DatagramChannel`
- `Pipe.SinkChannel`
- `Pipe.SourceChannels`
- File non inclusi

La registrazione di un canale su un selettore avviene nel seguente modo:

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

Il canale deve essere in modalità non bloccante e non si possono usare `FileChannel` con i `Selector`.

Il secondo parametro "ops" è l'interest set: indica quali eventi si è interessati a monitorare su quel canale (per esempio `SelectionKey.OP_READ`). Il terzo parametro "attach" è un buffer associato al canale.

L'interest set viene rappresentata come una bitmask di 8 bit (un intero) che 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` 00000001
- accept: `OP_WRITE-4` 00000100
- read: `OP_ACCEPT-16` 00010000
- write: `OP_READ | OP_WRITE-5` 00000101

Manipolato con gli operatori Java "`|`", "`&`", "`^`", "`~`", che eseguono operazioni bit a bit su operandi interi o booleani. Non tutte le operazioni sono valide per tutti i `SelectableChannel`, ad esempio `SocketChannel` non supporta `accept()`.

Nella classe `SelectionKey`, 4 costanti predefinite che corrispondono alle bitmask:

- `SelectionKey.OP_CONNECT`
- `SelectionKey.OP_ACCEPT`
- `SelectionKey.OP_READ`
- `SelectionKey.OP_WRITE`

Sono utilizzabili in fase di registrazione del canale con il `Selector` per impostare il valore iniziale dell'interest set.

Per ottenere l'interest set basta scrivere il seguente comando:

```
int interestSet = SelectionKey.interestOps()
```

Ogni registrazione di un canale su un selettore restituisce una chiave, un "token" che la rappresenta. È un oggetto di tipo `SelectionKey` valida fino a che non viene cancellata esplicitamente.

Lo stesso canale può essere registrato con più selettori con una chiave diversa per ogni registrazione.

L'oggetto `SelectionKey` 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

```
// 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 selettore e registra il server al Selector

Selector selector = Selector.open();
server.register(selector,SelectionKey.OP_ACCEPT, null);
```

L'eventuale allegato

| 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               |

Il metodo `int selector.select()` è 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()` e costruisce un insieme contenente le `SelectionKeys` dei canali pronti.

Il metodo `int select(long timeout)` si blocca fino a che non è trascorso il timeout, oppure vale una delle condizioni precedenti.

Il metodo `int selectNow()` è non bloccante. Nel caso nessun canale sia pronto restituisce il valore 0.

Il *ready set* viene 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}

Viene inizializzato a 0 quando la chiave viene creata e non può essere modificato direttamente. Si usano le operazioni su bitmask per verificare se si è verificato un evento.

Può essere ottenuto mediante il metodo `readyOps()` invocato su una `SelectionKey`. Supponiamo che "key" sia una `SelectionKey`, per testare se ci sono dati pronti per essere letti bisogna scrivere:

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

`key.isReadable()` equivale a `key.readyOps() & SelectionKey.OP_READ != 0`, esistono shortcut analoghe anche per le altre operazioni.

```
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() {};
}

```

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).

La `select` svolge i seguenti passi:

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":
  - a. 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.
  - b. Altrimenti: resetta il ready set e lo imposta con la chiave della operazione pronta e aggiunge la chiave al Selected Key Set.

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.

Per resettare il ready set bisogna rimuovere la chiave dall'insieme delle chiavi selezionate.

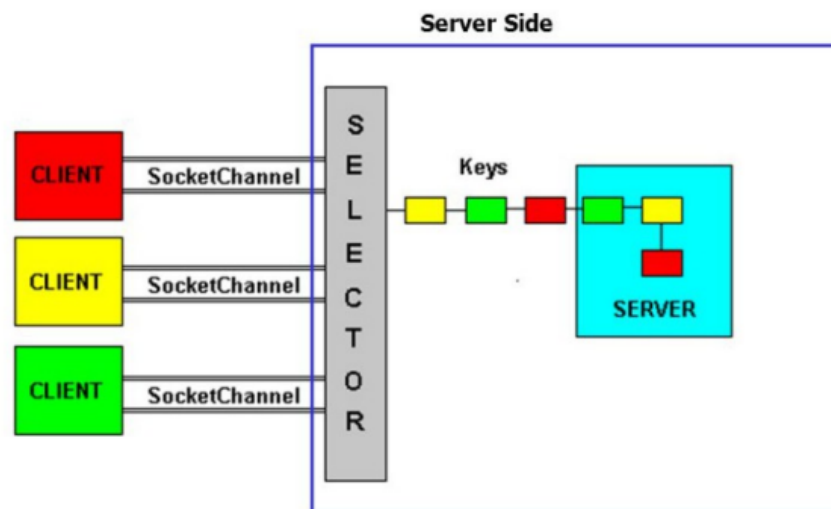
```
Selector selector = Selector.open();
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
while (true) {
 int readyChannels = selector.selectNow();
 if (readyChannels == 0) continue;
 Set<SelectionKey> selectedKeys = selector.selectedKeys();
 Iterator<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();
 }
}

```



Si attua un'interazione sull'insieme delle chiavi per individuare i canali pronti. Dalla chiave si può ottenere un riferimento al canale su cui si è verificato l'evento.

`keyIterator.remove()` deve essere invocato, poichè il Selector non rimuove le chiavi.



L'*attachment* è un riferimento ad un generico Object. Risulta utile quando si vuole accedere ad informazioni relative al canale (associato ad una chiave) che riguardano il suo stato pregresso. È 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 o memorizzare il numero di bytes che si devono leggere in totale.

## UDP DatagramPacket e DatagramSocket

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, ma:

- Aggiunge un ulteriore livello di indirizzamento a quello offerto dal livello IP, quello delle porte
- Offre un servizio di scarto dei pacchetti corrotti

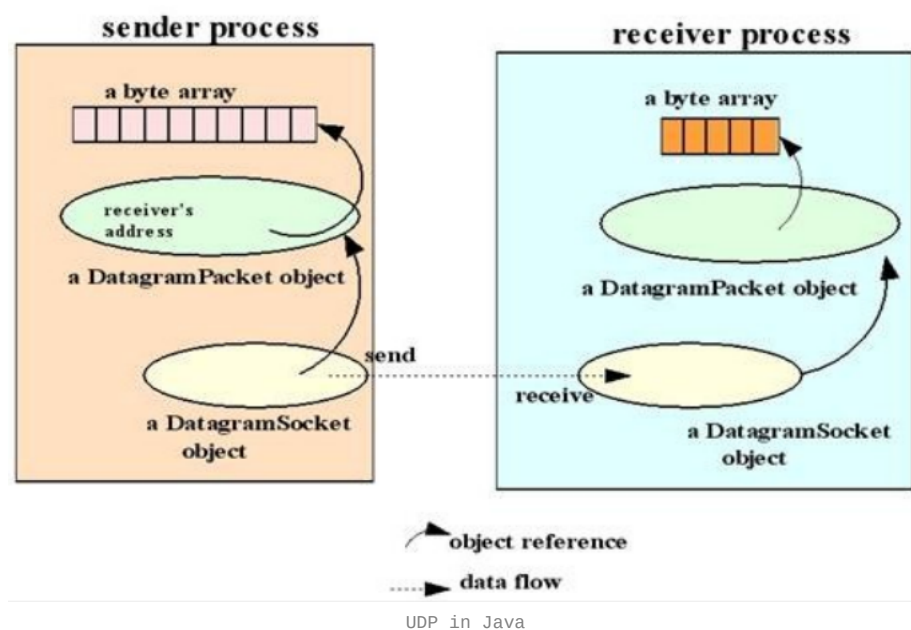
Conviene usare UDP nei seguenti casi:

- 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 keepalive 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 che si basano su UDP sono le prime versioni di NFS, TFTP e alcuni protocolli peer-to-peer.

Mentre con il TCP usavamo delle Stream Sockets, aprendo una connessione per collegare il client al server, con l'UDP si usano le Datagram Sockets che non richiedono un collegamento prima di inviare una lettera ma è piuttosto necessario specificare l'indirizzo del destinatario per ogni lettera spedita.

Con UDP una lettera è un "pacchetto", ogni pacchetto, chiamato *DatagramPacket*, è indipendente dagli altri e porta l'informazione per il suo instradamento.



Per sviluppare un DayTime client procediamo nel seguente modo:

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);
```

```

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

public class DayTimeUDPClient {
 // RFC-867
 private static final 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();}
 }
}

```

Per sviluppare un DayTime server procediamo nel seguente modo:

1. Aprire un DatagramSocket su una porta “nota” (“well known port”) 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

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

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);
```

```

import java.net.*;
import java.util.Date;
import java.io.*;

public class DayTimeUDPServer {
 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);
 }
 }
 }
 }
}

```

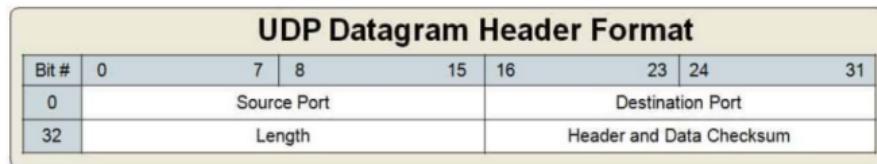
```

 }
 catch (IOException | RuntimeException ex) {ex.printStackTrace();}
 }
 }
 catch (IOException ex) {ex.printStackTrace();}
 }
}

```

La dimensione massima teorica di un pacchetto è 65537 bytes, dove l'IP header ha dimensione di 20 bytes e l'UDP header ha dimensione di 8 bytes. Molte piattaforme limitano la dimensione massima a 8192 bytes.

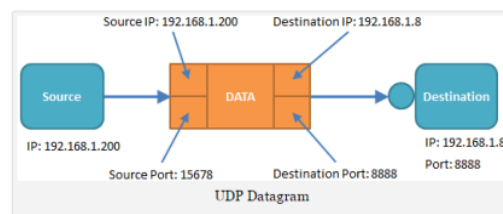
In Java un *datagram UDP* è rappresentato come una istanza della classe DatagramPacket.



Il mittente deve inizializzare:

- Il campo DATA
- Destination IP e Destination Port

Source IP è inserito automaticamente mentre Source Port può essere effimera.



Datagram UDP

Esistono in totale 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 i 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.

Il costruttore `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.

Il costruttore `public DatagramPacket(byte[] buffer, int length, InetAddress remoteAddr, int remotePort)` 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` e se il byte buffer contiene più di `length` bytes, questi non vengono copiati.

`Destination` + `port` individuano il destinatario.

Per essere memorizzato nel buffer, il messaggio deve essere trasformato in una sequenza di bytes. Per generare vettori di bytes si usa:

- Il metodo `getBytes()`
- La classe `java.io.ByteArrayOutputStream`

`DatagramSocket()` crea una socket e la collega ad una qualsiasi porta libera disponibile sull'host locale.

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

Utilizzato generalmente lato client, per spedire datagrammi.

Il metodo `getLocalPort()` serve 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 una socket e la collega alla porta specificata, sull'host locale. Il server crea una socket collegata 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.

Quando il socket viene chiuso, la porta viene utilizzata per altre connessioni.

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 Exceptio {
 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 generalmente troncati.

Esistono 6 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 port)`: 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);
```

Esistono 6 metodi get:

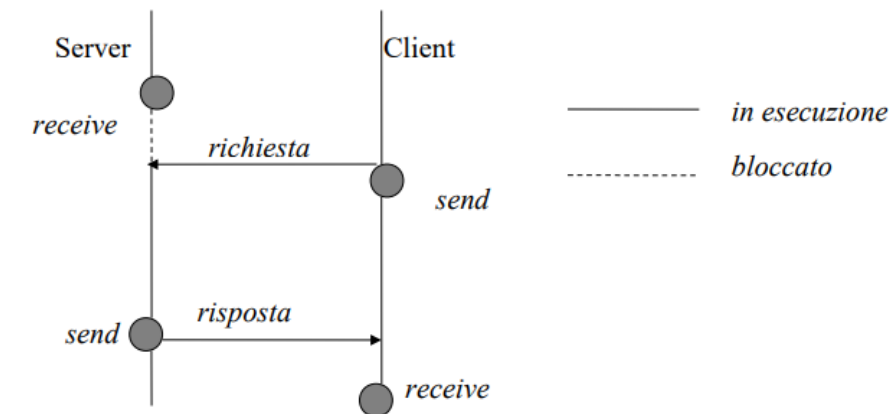
- `InetAddress getAddress()`: restituisce l'indirizzo IP della macchina a cui il Datagram è stato inviato oppure della macchina da cui è stato spedito.
- `int getPort()`: restituisce il numero di porta sull'host remoto a cui il Datagram è stato inviato, oppure della macchina da cui è stato spedito.
- `byte[] getData()`: restituisce un byte array contenente i dati del buffer associato al Datagram, ignora offset e lunghezza.
- `int[] getLength()`: restituisce la lunghezza del Datagram da inviare o da ricevere.
- `int[] getOffset()`: restituisce l'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.

L'invio di pacchetti avviene tramite `sock.send(dp)`, sock è la socket attraverso la quale voglio spedire il Datagram dp.

La ricezione di pacchetti avviene tramite `sock.receive(dp)`, sock è la socket attraverso la quale ricevo il Datagram dp. Riempie il buffer associato al socket con i dati ricevuti. Il Datagram ricevuto contiene anche indirizzo IP e porta del mittente.

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

La *receive* è 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`.



```

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();}
 }
}

```

```

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);
 }
 }
}

```

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.

SO\_TIMEOUT è una proprietà associata al socket, indica l'intervallo di tempo, in millisecondi, di attesa di ogni receive eseguita su quella 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.

Per gestire il time out si usa `public synchronized void setSoTimeout(int timeout) throws SocketException`.

I dati inviati mediante UDP devono essere rappresentati come vettori di bytes. Alcuni metodi per la conversione stringhe/vettori di bytes sono:

- `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` prelevando `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.

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

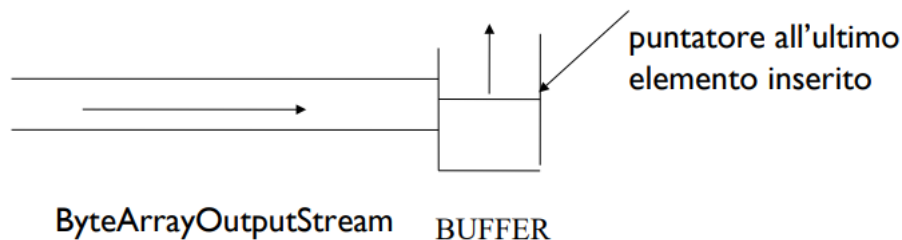
Gli oggetti istanze di questa classe rappresentano stream di bytes. Ogni dato scritto sullo stream viene riportato in un buffer di memoria a dimensione variabile (dimensione di default = 32 bytes).

```
protected byte buf[]
```

```
protected int count
```

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

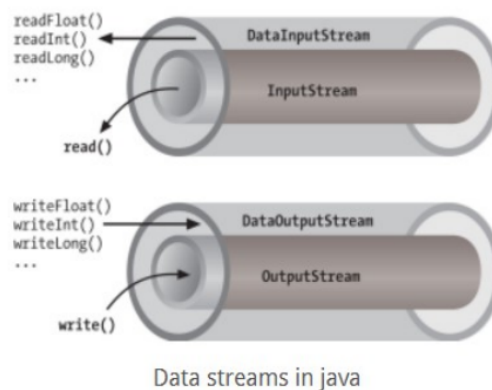
Quando il buffer si riempie la sua dimensione viene raddoppiata automaticamente.



È possibile collegare ad un `ByteArrayOutputStream` un altro filtro:

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

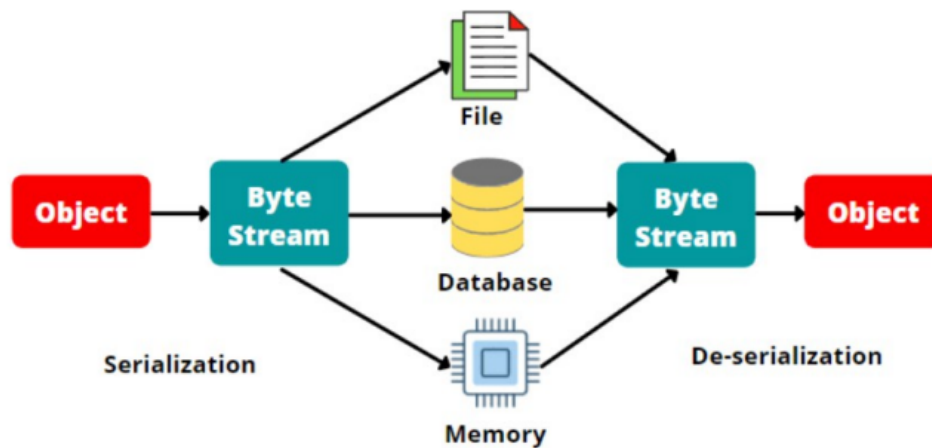
`DataOutputStream/DataInputStream` consentono di scrivere dati primitivi sullo stream, la trasformazione in bytes è effettuata automaticamente



Per inviare oggetti in pacchetti si usa la serializzazione per generare uno stream di byte. Si collega l'outputstream generato ad un `ByteArrayOutputStream`.

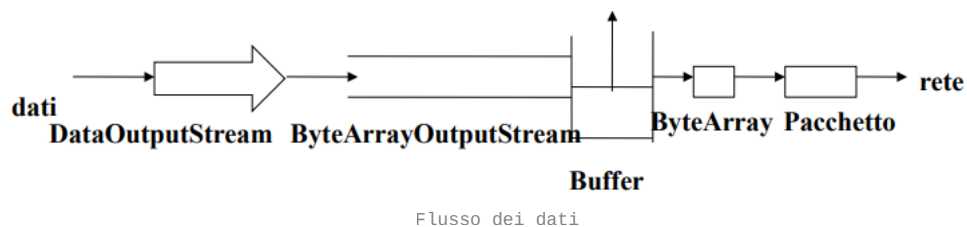
```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(baos);
```





I dati presenti nel buffer B associato ad un ByteArrayOutputStream baos possono essere copiati in un array di bytes.

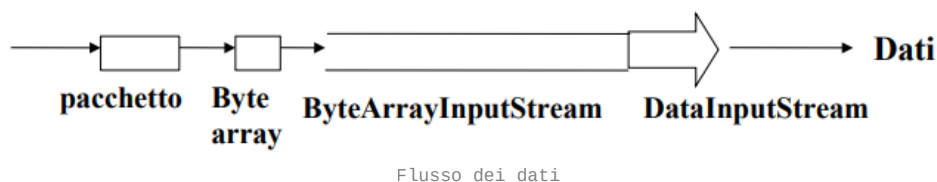
```
byte[] barr = baos.toByteArray();
```



```
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.



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.
- `public synchronized byte toByteArray ()`: restituisce un vettore in cui sono stati copiati tutti i bytes presenti nello stream. Non modifica count e non svuota il buffer.

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

public class multidatastreamsender {
 public static void main(String args[]) throws Exception {
 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);
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();
}
}
}

```

```

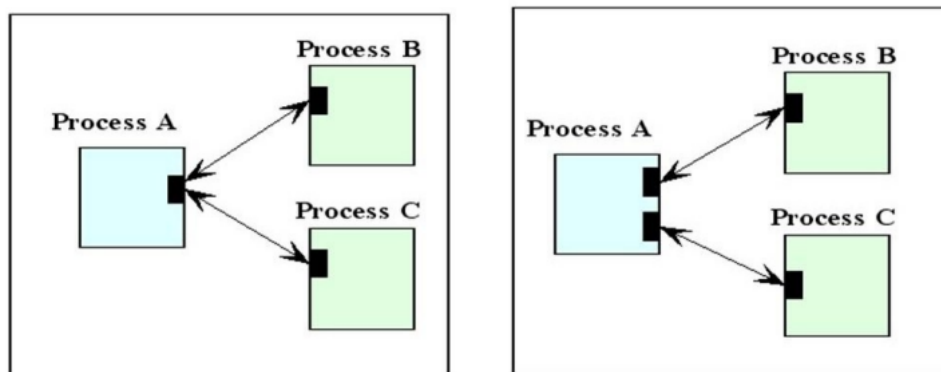
import java.io.*;
import java.net.*;
public class multidatastreamreceiver {
 public static void main(String args[]) throws Exception {
 FileOutputStream fw = new FileOutputStream("text.txt");
 DataOutputStream dr = new DataOutputStream(fw);
 int port = 13350;
 DatagramSocket ds = new DatagramSocket(port);
 byte[] buffer = new byte[200];
 DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
 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);
 }
 }
}

```

In una trasmissione connectionless gli stream utilizzati per la generazione dei pacchetti:

- `ByteArrayOutputStream`, consentono la conversione di uno stream di bytes in un vettore di bytes da spedire con i pacchetti UDP.
- `ByteArrayInputStream`, converte un vettore di bytes in uno stream di byte.

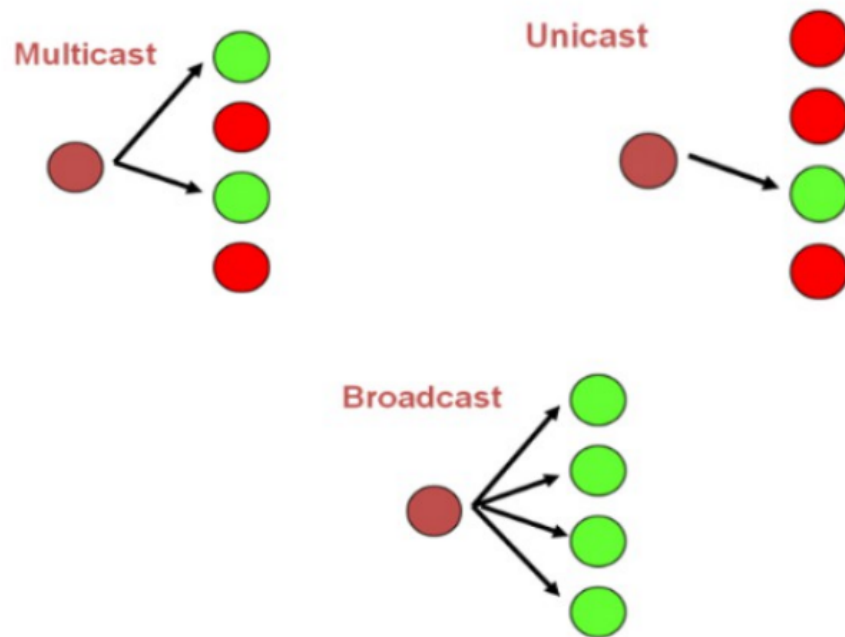
È possibile usare lo stesso socket per spedire pacchetti verso destinatari diversi. Processi (applicazioni) diversi possono spedire pacchetti sullo stesso socket, in questo caso l'ordine di arrivo dei messaggi è non deterministico, in accordo con il protocollo UDP ma è anche possibile utilizzare diversi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



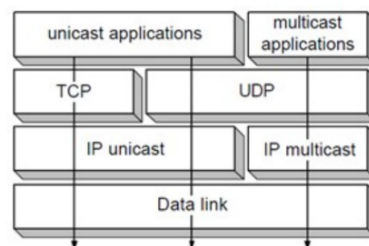
## Multicasting

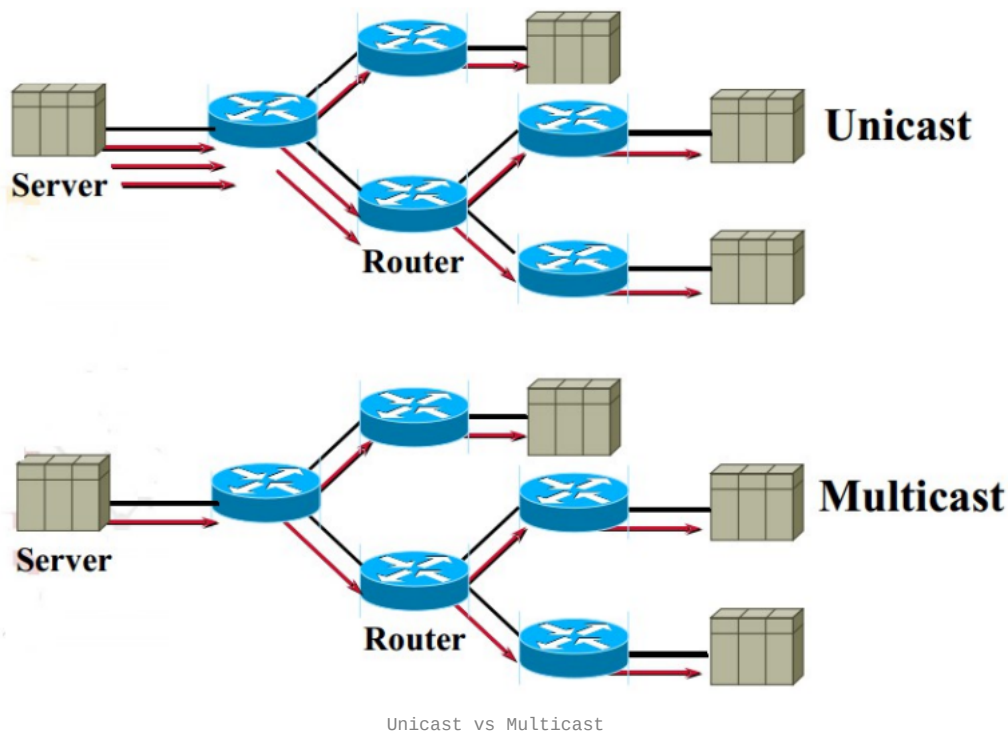
Il *multicasting* permette di inviare dati da un host ad un insieme di altri nodi (non tutti gli host della rete: solo quelli che hanno espresso interesse ad unirsi ad un gruppo di multicast).

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 multicast.



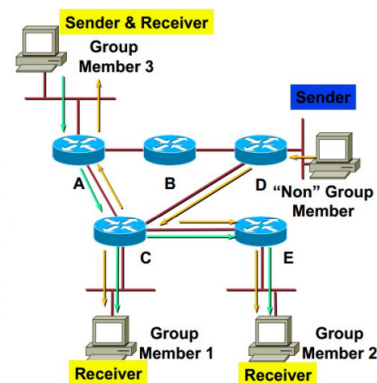
IP multicasting utilizza UDP e non esiste alcun tipo di multicast TCP.





L'IP multicast è 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

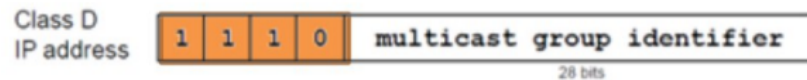
Il supporto deve fornire:

- 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.

Lo schema di indirizzamento multicast è 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 |||| |||| in binario.

Il multicast 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*.

La comunicazione *connectionless* è adatta per il tipo di applicazioni verso cui è orientato il multicast, come ad esempio la 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.

`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");}
 }
}
```

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

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

`joinGroup` è 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.

Viene sollevata una `IOException` se l'indirizzo di multicast è errato.

```
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 `provenmulticast` 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.

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 nel seguente modo:

```
try {sock.setReuseAddress(true);}
catch (SocketException se) {...}
```

### 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).