

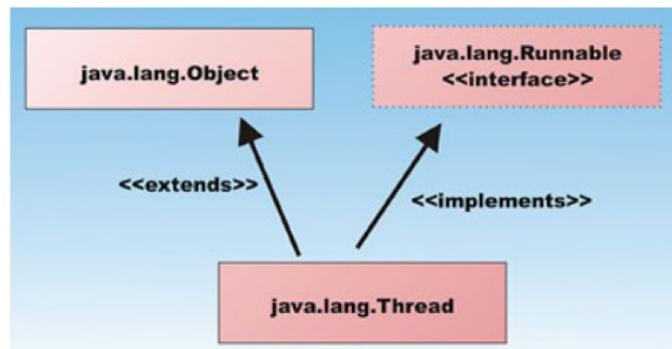
CREAZIONE ED ATTIVAZIONE DEI THREAD

- 1) la JVM crea un thread che invoca il metodo `main` del programma, quindi c'è sempre almeno un thread
- 2) Altri thread sono attivati automaticamente da Java (gestione eventi, interfaccia, garbage collector)
- 3) Ogni thread durante la sua esecuzione può creare ed attivare altri thread.

Primo metodo: Implements Runnable

- 1) Definire una classe R che implementi l'interfaccia `Runnable`, cioè implementare il metodo `run`. In questo modo si definisce un oggetto `Runnable`, cioè un task.
- 2) Allocare un'istanza T di R.
- 3) Allocare un oggetto thread, utilizzando il costruttore : `public Thread (Runnable target)`
- 4) Attivare il thread con una `start()`.

Gerarchia delle classi: Runnable



- Thread, memorizza un riferimento all'oggetto `Runnable`, passato come parametro, nella variabile `runnable`
- il metodo `run()` della classe `Thread` è definito come segue

```
public void run() {
    if (runnable != null)
        runnable.run();
}
```
- l'invocazione del metodo `start()` provoca la esecuzione del metodo `run()`, che, a sua volta, provoca l'esecuzione dei metodi `run()` della `Runnable`

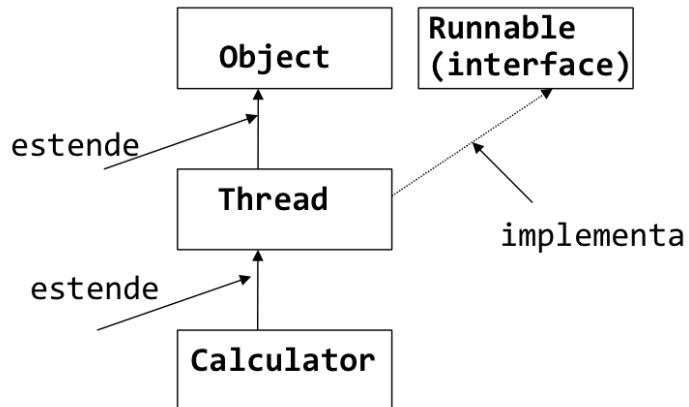
Metodo Start

- segnala allo scheduler (tramite la JVM) che il thread può essere attivato (invoca un metodo nativo)
- l'ambiente del thread viene inizializzato.
- restituisce immediatamente il controllo al chiamante, senza attendere che il thread attivato inizi la sua esecuzione.

Secondo Metodo : Extends thread

- 1) Creare una classe C che estenda Thread
- 2) Effettuare l'overriding del metodo run() definito da quella classe
- 3) Instantiazione di un oggetto T di quella classe
- 4) Invocare il metodo start() sull'oggetto instantiato

Generalità delle classi: overriding



- overriding del metodo run() all'interno della classe che estende Thread()
 - all'interno della classe Calculator
 - il comportamento del thread è definito dal metodo run di Calculator

OSSERVAZIONE: In Java una classe può estendere una sola altra classe (Eredità Singola)

Risultato: usare la versione "Extends thread" può risultare svantaggioso

Thread Demoni

- threads a bassa priorità
 - adatti per jobs non-critici da eseguire in background
 - servizi di background utili fino a che il programma è in esecuzione, generalmente creati dalla JVM, ad esempio per garbage collection
 - ma anche l'utente può dichiarare che un thread è un demone
- non appena tutti i thread non demoni del programma sono terminati, la JVM termina il programma, forzando la terminazione dei thread demoni
- un esempio di thread non-demon è il main()

Per indicare che c'è un thread demone

si usa `dt.setDaemon(true);`

Terminazione programmi concurrenti

- 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, fino alla loro terminazione.
 - il "quadratino" rosso di Eclipse rimane "rosso" anche se il main è terminato
- se uno dei thread usa l'istruzione `System.exit()` per terminare l'esecuzione, allora tutti i threads terminano la loro esecuzione

Gestire le interruzioni

JAVA mette a disposizione

- un meccanismo per interrompere un thread
- diversi meccanismi per intercettare l'interruzione
 - dipendenti dallo stato in cui si trova un thread, running, blocked
 - se il thread è sospeso l'interruzione solleva una `InterruptedException`
 - se è in esecuzione, può testare un flag che segnala se è stata inviata una interruzione.
- il thread decide comunque autonomamente come rispondere alla interruzione

```
public class SleepMain {
    public static void main (String args [ ]) {
        SleepInterrupt si = new SleepInterrupt();
        Thread t = new Thread (si);
        t.start ( );
        try
        {Thread.sleep(2000);}
        catch (InterruptedException x) {    };
        System.out.println("Interromo l'altro thread");
        t.interrupt( );
        System.out.println ("sto terminando..."); } }
```

```
public class SleepInterrupt implements Runnable
{
    public void run ()
    {
        try{System.out.println("dormo per 20 secondi");
            Thread.sleep(2000);
            System.out.println ("svegliato");}
        catch ( InterruptedException x )
        { System.out.println("interrotto");return;};
        System.out.println("esco normalmente");
    }
}
```

- in un istante compreso tra l'inizio e la fine della sleep (inizio e fine inclusi), al thread arriva una interruzione
- allora l'eccezione viene lanciata
- il metodo `interrupt()`
 - imposta a true un valore booleano nel descrittore del thread.
 - il flag vale true, se esistono interrupts pendenti
- per testare il valore del flag:
 - `public static boolean Interrupt ()`
(metodo statico, si invoca con il nome della classe `Thread.Interrupted()`)
 - `public boolean isInterrupted ()` → **Higlone!**
deve essere invocato su un'istanza di un oggetto di tipo thread
 - entrambi i metodi
 - restituiscono un valore booleano che segnala se il thread ha ricevuto un'interruzione
 - `interrupt()` rimette la flag a false, mentre `isInterrupted()` non cambia il valore

Proprietà di un thread

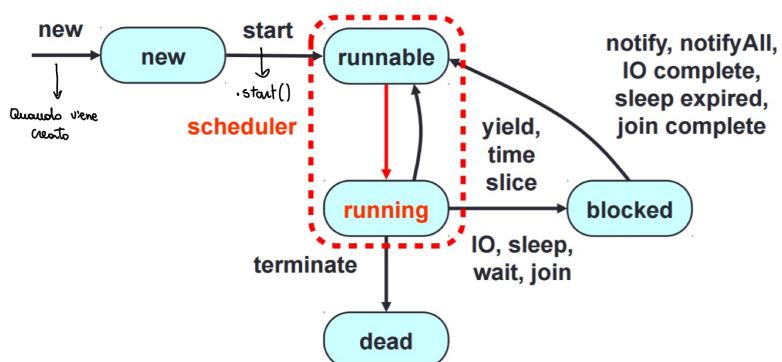
la classe `Thread` salva alcune info. sul thread:

- **ID**: identificatore del thread
- **nome**: nome del thread
- **priorità**: valore da 1 a 10 (1 priorità più bassa).
- **nome gruppo**: gruppo a cui appartiene il therad
- **stato**: uno dei possibili stati: `new`, `Runnable`, `blocked`, `waiting`, `time waiting` o `terminated`.

Per reperire o modificare le proprietà si usano i metodi `setter` e `getter`.

Esempio: `public final void setName(String newName),`
`public final String getName()`

Stati di un thread



Methods Join threads

Le thread che esegue la Join() si sospende in attesa della terminazione di t.

E' possibile specificare un tempo max di attesa (timeout di attesa)

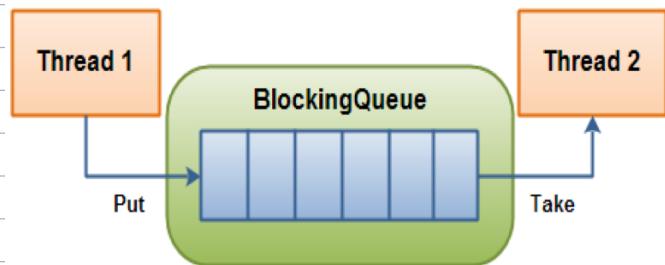
Se il thread sospeso sulla `join()` riceve una interruzione, viene sollevata una eccezione.

Blocking Queue (Interfaccia)

Java mette a disposizione una Coda thread safe per quanto riguarda inserimenti e rimozioni.

Se il produttore può inserire elementi se la coda non c'è piena, nessuno si blocca.

Il consumatore può estirare elementi se la coda non è vuota, nemmeno si blocca.



ha codice annette li metodi diversi per ogni operazione (insert, remove, examine):

lancia eccezione
↓ se non
riesce

restituisce
 ↙ booleans

Blocca il thread
/ se non
risponde

Si blocca per un tot
ne può riuscire e restituirlo.
un booleano

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

Formbili implementazioni:

- `ArrayBlockingQueue`
 - `DelayQueue`
 - `LinkedBlockingQueue`
 - `PriorityBlockingQueue`
 - `SynchronousQueue`

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

THREAD POOLS

Avere un thread per ogni task è svantaggioso perché:

Quindi conviene creare una Thread pool

cioè una struttura dati la cui dimensione max

può essere prefissata, che contiene i riferimenti a un insieme di threads.

Il vantaggio è che questi thread delle pool possono essere riutilizzati per l'esecuzione di più task, cioè, per esempio, il thread-1 può eseguire sia le task che calcola la tabellina dell'1, sia quella che calcola la tabellina del 10.

- L'utente crea il pool e stabilisce una politica di gestione dei thread interna:

- 1) Quando i thread vengono attivati (alla creazione del pool, su richiesta, all'arrivo di un task ecc...)
- 2) Se e quando è opportuno terminare l'esecuzione di un thread.
- 3) Sottomette i task per l'esecuzione del thread pool.

- Il supporto al momento della sottomissione del task può:

- 1) Utilizzare un thread attivato in precedenza, inattivo in quel momento.
- 2) Creare un nuovo thread
- 3) Memorizzare il task in una coda, in attesa dell'esecuzione
- 4) Restringere la richiesta di esecuzione del task.

Implementazione

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

Servizi che implementano le interfacce: (ThreadPoolExecutor, ScheduledThreadPoolExecutor,..)

Classe: la classe **Executor** opera come una factory in grado di generare oggetti di tipo **ExecutorService** con comportamenti predefiniti

N.B.: i tasks devono essere incapsulati in oggetti di tipo **Runnable** e passati a questi esecutori, mediante invocazione del metodo **execute()**

thread life cycle overhead

- overhead per la creazione/distruzione dei threads: richiede un'interazione tra JVM e sistema operativo
 - varia a seconda della piattaforma, ma non è mai trascurabile
 - per richieste di servizio frequenti e 'lightweight' può impattare negativamente sulle prestazioni dell'applicazione
- ### resource consumption
- molti threads idle quando il loro numero supera il numero di processori disponibili. Alta occupazione di risorse (memoria,...)
 - mette sotto stress sia il garbage collector che lo scheduler
- ### stability
- limitazione al numero di threads imposto dalla JVM/dal SO

Definizione di un server concorrente:

```
import java.util.concurrent.*;  
public class Server {  
    private ThreadPoolExecutor executor;  
    public Server() {  
        executor=(ThreadPoolExecutor) Executors.newCachedThreadPool();  
    }  
  
    public void executeTask(Task task){  
        System.out.printf("Server: A new task has arrived\n");  
        executor.execute(task);  
        System.out.printf("Server:Pool Size:%d\n",executor.getPoolSize());  
        System.out.printf("Server:Active Count:%d\n",executor.getActiveCount());  
        System.out.printf("Server:Completed Tasks:%d\n",  
            executor.getCompletedTaskCount());  
  
    }  
  
    public void endServer() { → Server Shutdown  
        executor.shutdown();}  
}
```

Comportamento del Pool

New Cached Thread Pool

Crea un pool con un comportamento predefinito.

Se tutti i thread della pool sono occupati nell'esecuzione di altri task e c'è un nuovo task da eseguire, viene creato un nuovo thread

⇒ Nessun limite sulla dimensione della pool

Altimenti, se disponibile un thread già creato, viene riutilizzato per eseguire il nuovo task.

Se un thread rimane inutilizzato per 60 sec., la sua esecuzione termina.

```
public Server() {  
    executor=(ThreadPoolExecutor) Executors.newCachedThreadPool();  
    :  
}
```

New Fixed Thread Pool()

Crea un pool con un comportamento predefinito.

Vengono creati N thread, al momento della initializzazione del pool, riutilizzati per più task.

Quando viene sottomeno un nuovo task:

illimitata

Se tutti i thread sono occupati, T viene inserito in una coda gestita automaticamente dall'Executor Service.

Altimenti: viene utilizzato un thread inattivo in quel momento.

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

THREAD POOL EXECUTOR

```
import java.util.concurrent.*;  
public class ThreadPoolExecutor implements ExecutorService  
{public ThreadPoolExecutor  
(int CorePoolSize, → Dimensione minima pool (creati initialmente)  
    int MaximumPoolSize, → Dimensione max pool  
    long keepAliveTime, → Quantità di tempo  
    TimeUnit unit, → Unità di tempo  
    BlockingQueue <Runnable> workqueue).....} → Struttura per memorizzare  
task in attesa di exec.
```

È il costruttore più generale il quale permette di personalizzare la politica di gestione dei pool.

Gli thread core possono essere creati in 2 modi:

- 1) Prendendo `AllCoreThreads()` al momento della creazione del pool.
- 2) "on demand", cioè quando c'è un nuovo task.

Attenzione: Si dà priorità alla creazione di tutti i ^{CORE} thread, e non a quelli inattivi.

Come funziona:

Se tutti i thread core sono stati creati e viene notificato un nuovo task:

Casi:

- 1) Se un thread del core è inattivo, il task viene assegnato ad esso;
- 2) Altrimenti se la coda di attesa non è piena, esso viene inserito in codice;
- 3) Se la coda è piena e tutti i core thread stanno eseguendo task, si crea un nuovo thread attivando così un thread t.c.

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

- 4) Se la coda è piena e sono attivi `MaxPoolSize` threads, il task viene respinto.

OSS: Gli thread non core fermano l'esecuzione del task aspettano il "keepAliveTime", se nessun task viene notificato entro la scadenza, il thread termina.

Tipi di coda utilizzabili:

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

IMPLEMENTAZIONE NewFixed e NewCached con THREAD POOL EXECUTOR

newFixedThreadPool

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

newCachedThreadPool

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

TERMINAZIONE DI UN POOL

La terminazione può avvenire in 2 modi:

1) Shutdown():

- nessun task viene accettato dopo che la shutdown() è stata invocata.
- tutti i tasks sottomessi in precedenza e non ancora terminati vengono eseguiti, compresi quelli accodati, la cui esecuzione non è ancora iniziata
- successivamente tutti i threads del pool terminano la loro esecuzione

2) ShutdownNow():

- non accetta ulteriori tasks, ed elimina i tasks non ancora iniziati
- restituisce una lista dei tasks che sono stati eliminati dalla coda
- tenta di terminare l'esecuzione dei thread che stanno eseguendo i tasks inviando a loro una interruzione. Questo non garantisce la terminazione immediata perché se un thread non risponde all'interruzione, non si termina.

Metodi per gestire la terminazione di un pool:

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

Per capire se l'esecuzione del pool è terminata:

- **attesa passiva**: invoco la **awaitTermination()**
- **attesa attiva**: invoco ripetutamente la **isTerminated()**

Ciclo di vita di un Pool:

1) Running: Quando viene creata

2) Shutting Down: Quando viene chiamata una Shutdown() o ShutdownNow()

3) Terminated: Quando tutti i thread sono terminati

CALLABLE E FUTURE

L'interfaccia Callable ci serve per definire un task che restituisca un valore di ritorno e sollevare eccezioni.

public interface Callable <V> → tipo generico del valore di ritorno.

```
{ V call() throws Exception; }
```

Future ci serve per rappresentare il risultato di una computazione asincrona.

```
public interface Future <V>
{ V get( ) throws...; → si blocca fino a che il thread non ha prodotto il valore richiesto
  V get (long timeout, TimeUnit) throws...; → definisce un tempo max di attesa della
  void cancel (boolean mayInterrupt);
  boolean isCancelled( );
  boolean isDone( ); }
```

terminazione del task, dopo cui viene sollevata una **timeout exception**

- Il valore restituito dalla Callable, acceduto mediante un **oggetto di tipo <Future>**, rappresenta il risultato della computazione
- Se si usano i thread pools
 - sottomette direttamente l'oggetto di tipo Callable al pool mediante il metodo **submit**
 - la sottomissione restituisce **un oggetto di tipo <Future>**

CONDIZIONE DELLE RISORSE

In un programma concorrente, poniamo avere più thread che vogliono accedere ad una risorsa condivisa.

L'accesso non controllato ad essa potrebbe provocare situazioni di errore ed inconsistente. (**Race Conditions**)

Queste risorse condivise vengono chiamate **sezioni critiche**, cioè blocchi di codice dove deve essere effettuato l'accesso da un thread alla volta.

Per gestire la sincronizzazione fra i vari thread possono essere usati 2 meccanismi:

- 1) L'interfaccia lock (lock() e variabili di condizione)
- 2) Concetto di monitor (synchronized(), wait(), notify(), notifyAll())

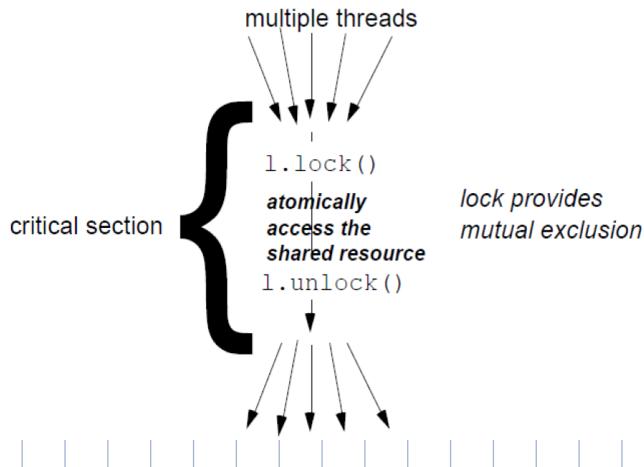
Lock

Una lock in Java è un oggetto che può trovarsi in 2 stati diversi:

1) locked

2) Unlocked

Gli stati vengono impostati con i metodi: lock(), unlock()



Y interfaccia:

```
interface Lock {  
    void lock(); → Prova a prenderlo, se non riesce si interrompe  
    void lockInterruptibly() → Acquisisce la lock fin quando non viene interrotta  
    boolean tryLock(); → Prova a prenderlo, se non riesce continua ad andare  
    boolean tryLock(long time, TimeUnit unit)  
    void unlock();  
    Condition newCondition(); }
```

Y implementazione: `private final Lock accountLock=new ReentrantLock();`

Dado lock:

- Thread(A) acquisisce Lock(X) e Thread(B) acquisisce Lock(Y)
- Thread(A) tenta di acquisire Lock(Y) e simultaneamente Thread(B) tenta di acquisire Lock(X)
- Entrambe i threads bloccati all'infinito, in attesa della lock detenuta dall'altro thread!

Si può usare il metodo "tryLock()" per provare ad acquisire la lock se libera, altrimenti il metodo termina immediatamente e restituisce il controllo al chiamante.

Il metodo restituisce inoltre un valore booleano, vero se è riuscito ad acquisire la lock(), falso altrimenti.

Performance: le lock introducono una performance penalty, quindi vanno usate con oculatezza.

Le lock in Java sono reentrant:

Per evitare di entrare in deadlock con se stessa

- incrementato ogni volta che un thread acquisisce la lock
- decrementato ogni volta che un thread rilascia la lock
- lock viene definitivamente rilasciata quando il contatore diventa 0
- un thread può acquisire più volte la lock su uno stesso oggetto senza bloccarsi

Read/write locks: Usate se c'è una struttura dati che puo' essere letta da più thread, ma scritta da solo un thread

interfaccia `ReadWriteLock`: mantiene una coppia di lock associate, una per le operazioni di lettura e una per le scritture.

- la read lock può essere acquisita da più thread lettori, purchè non vi siano scrittori.
- la write lock è esclusiva.

Implementation: `ReentrantReadWriteLock()`

Synchronized

Blocco Synchronized

```
public void somemethod ()  
synchronized (object) {  
    // a thread that is executing this code section  
    // has acquired the object intrinsic lock  
    // only a single thread may execute this  
    // code section at any given time  
}
```

Metodo Sincronizzato

```
public synchronized void f() {  
    .....  
}
```

Viene acquisita la lock su un oggetto,

di solito si usa un oggetto `Object` su

da ponere a `Synchronized`.

+ per ogni blocco di codice che

voglio eseguire in parallelo

```
Object mutex = new Object();  
...  
  
public void someMethod(){  
    nonCriticalSection();  
  
    synchronized (mutex){  
        criticalSection();  
    }  
  
    nonCriticalSection();  
}
```

Sincronizza l'intero metodo.

OSS:

i costruttori non devono essere dichiarati synchronized

- il compilatore solleva una eccezione
- solo il thread che crea l'oggetto deve avere accesso ad esso mentre l'oggetto viene creato

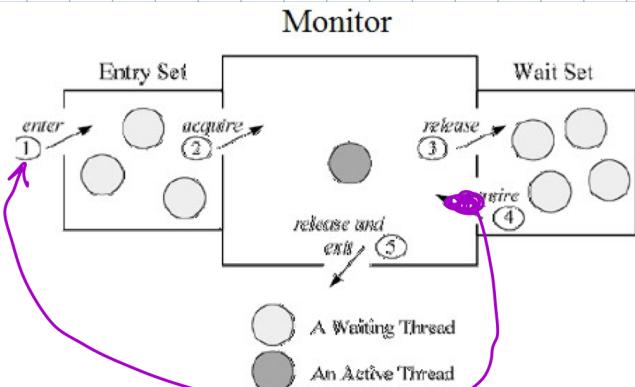
non ha senso specificare synchronized nelle interfacce

se una sottoclasse sovrscrive (override) un metodo synchronized della superclasse, il metodo della sottoclasse deve essere reso synchronized, non lo è per default

la lock è associata ad una istanza dell'oggetto, non alla classe, metodi su oggetti che sono istanze diverse della stessa classe possono essere eseguiti in modo concorrente!

Monitor = Meccanismo linguistico ad alto livello per la sincronizzazione

→ Un oggetto con metodi synchronized che incapsula lo stato di una risorsa condivisa.



Vantaggi e svantaggi:

- vantaggi:
 - costrutti strutturati. Evitare errori dovuti alla complessità del programma concorrente: deadlocks, mancato rilascio di lock,....
 - maggior manutenibilità del software
- svantaggi: poco flessibili

Metodi Wait e Notify

- 1) **Wait()**: Sospende il thread in attesa delle verifiche di una condizione
- 2) **Notify()**: Notifica di una condizione specificata a un thread qualiasi in wait
- 3) **NotifyAll()**: Notifica di una condizione specificata a tutti i thread in wait

OSS: I thread risvegliati "combattono" per la lock insieme a quelli in ready.

Attenzione: Mettere il wait sempre in un while!

LOCK FREE ATOMIC OPERATIONS

Java Concurrency include il package `java.util.concurrent.atomic`

- `AtomicBoolean`
- `AtomicInteger`
- `AtomicLong`

Sono classi che incapsulano variabili di

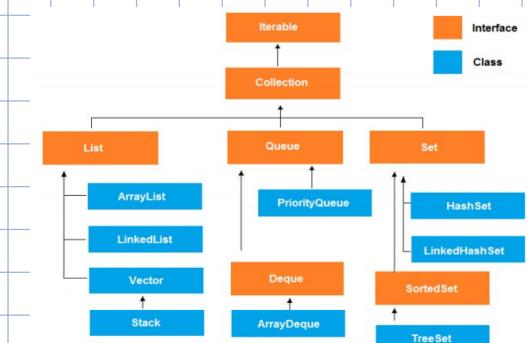
⇒ tipo primitivo e garantiscono l'atomicità delle op:

- `incrementAndGet()`: atomically increments by one the current value.
- `decrementAndGet()`: atomically decrements by one the current value.

Garantiscono l'atomicità delle operazioni per tipi di dato primitivi ed array senza l'uso di sincronizzazioni e lock.

JAVA COLLECTION FRAMEWORK

Un insieme di classi che consentono di lavorare con gruppi di oggetti, ovvero collezioni di oggetti:



l'interfaccia map (Implementation Hashmap)

+

le collezioni possono essere:

- collezioni che non offrono alcun supporto per il multithreading
- synchronized collections
- concurrent collections

Synchronized Collections

Si può usare il metodo "SynchronizedCollection(...)" per sincronizzare una collezione

```
Collection<Integer> synchCollection =  
    Collections.synchronizedCollection(new ArrayList<Integer>());  
synchCollection.addAll(Arrays.asList(1,2,3,4,5,6));  
System.out.println(synchCollection);
```

Metodi:

synchronizedList, synchronizedMap, synchronizedSet,....

Il metodo restituisce una collezione thread-safe:

- garantita con lock intrinseche sull'intera collezione
- ogni metodo sincronizzato sulla stessa lock
- implementata mediante metodi wrapper che contengono costrutti di sincronizzazione
- o qualcosa del genere
- degradazione di performance: un singolo thread alla volta sulla intera collezione

Attenzione: Creare metodi che includono più op. base della collezione potrebbero non essere thread-safe!

Quindi per rendere atomica una op. composta si esprime una sync esplicita!

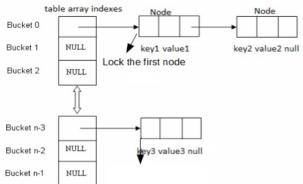
```
synchronized(synchList) {  
    if(!synchList.isEmpty())  
        synchList.remove(0);  
}
```

Concurrent collections: Sono un approccio alternativo alle Synchronized collections

- vantaggio:
 - ottimizzazione degli accessi, non una singola lock
 - thread-safeness con miglior performance
 - overlapping di operazioni di scrittura su parti diverse della struttura
 - letture in parallelo a modifiche della struttura
- svantaggio:
 - semantica "approssimata" di alcune operazioni size(), isEmpty()
 - weak consistency

→ le synchronized bloccano l'intera struttura!

Concurrent hash Map:



- HashMap: per default 16 buckets (incremento se load factor è alto)
- ConcurrentHashMap
 - lock striping: una lock per ogni bucket, associata al primo elemento
 - modifiche simultanee possibili, se modificano sezioni diverse della tabella
 - 16 o più threads possono operare in parallelo su bucket diversi
 - lettori possono accedere in parallelo a modifiche
 - richiede cambiamento della semantica degli iteratori

le op. usate

⇒

denevo sono atomiche!

```
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}
nuove funzioni per garantire atomicità ad operazioni composte
```

Iteratori:

Oggetti di supporto per accedere agli elementi di una collezione, uno alla volta e in sequenza.

Ogni iteratore è associato ad un oggetto collezione!

- schema generale per l'uso di un iteratore

```
// collezione di oggetti di tipo T che vogliamo scandire
Collection<T> c = ....
...
// iteratore specifico per la collezione c
Iterator<T> it = c.iterator()

// finché non abbiamo raggiunto l'ultimo elemento
while (it.hasNext()) {
    // ottieni un riferimento all'oggetto corrente, ed avanza
    T e = it.next();
    ...     // usa l'oggetto corrente (anche rimuovendolo)
}
```

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

Se una collezione viene modificata

mentre un altro thread sta iterando su

essa, gli iteratori possono comportarsi in 2 modi:

Fail-fast iterators

- sollevano una ConcurrentModificationException se c'è una modifica strutturale (inserzione, rimozione, aggiornamento) alla collezione su cui l'iteratore sta operando
- iteratori su ArrayList, HashMap, ed altre collezioni

Fail-safe iterators

- non sollevano una ConcurrentModificationException
 - creano un clone della collection e lavorano su quello
- iteratori su ConcurrentHashMap, CopyOnWriteArrayList, etc.

Input / Output Java

I/O coinvolge nel trasferire informazioni da una sorgente esterna o inviare ad una sorgente esterna.

1) File System: Files e directories

Java I/O API = Package I/O standard basato su stream

2) Connessioni di rete

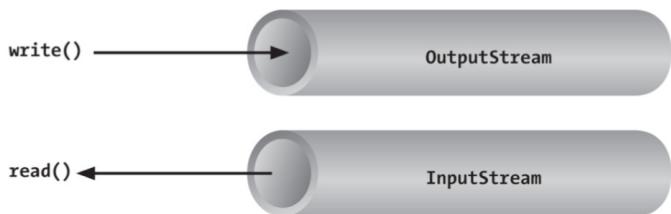
3) Keyboard

Java NIO API (New I/O) = Funzionalità simili a java.io, ma basato su canali e con comportamento non bloccante

4) In-memory Buffers (array)

Java NIO.2 = Alcuni miglioramenti rispetto a Java NIO

Sream = Uno stream rappresenta una connessione tra un programma Java ed un dispositivo esterno.



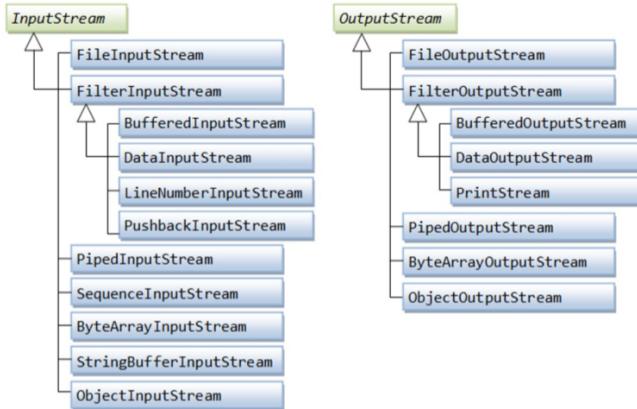
Caratteristiche:

- Accesso sequenziale
- Mantengono l'ordinamento FIFO
- One way: Read only oppure write only (se un programma ha bisogno di dati in input ed output)
- Bloccanti: Quando un'applicazione legge un dato dallo stream (o lo scrive) si blocca finché l'operazione non è completa.
- Non c'è richiesta una corrispondenza stretta fra lettura / scrittura

Java I/O classi Fondamentali: InputStream, OutputStream, Reader, Writer

\ / \ /
lavorano su bytes lavorano su caratteri

Classi Byte Stream



Input/OutputStream:

- forniscono operazioni base
 - possono essere "associati" ad ogni tipo di device di input/output
- implementazioni diverse per tipi diversi di I/O:
- console: `System.out`, `System.in`
 - files: `FileInputStream`, `FileOutputStream` per leggere/scrivere dati da un file
 - in-memory buffers: per trasferimento di dati da una parte all'altra di un programma JAVA: `ByteArrayOutputStream`, `ByteArrayInputStream`
 - utili per generare uno stream di byte, per poi trasferirlo in un pacchetto UDP.
 - connessioni TCP
 - `PipeInputStream`: pipes

- `int read()`
 - legge un byte dallo stream come un intero unsigned tra 0 e 255
 - restituisce -1 se viene individuata la "fine dello stream" (derivato dal C)
 - solleva una `IOException` se c'è un errore di I/O
 - bloccante
- `public int read(byte[] bytes)`
 - riempie il vettore passato in input con tutti i dati disponibili sullo stream fino alla capacità massima del vettore e restituisce il numero di byte letti
- `skip(long n)`, `available()`, `close()` e molti altri metodi
- `int waiting = System.in.available(); //funziona solo per FileStream`
- `if (waiting > 0) {`
 - `byte [] data = new byte [waiting];`
 - `System.in.read(data);`
 - `...}`
- diverse classi concrete implementano `InputStream`, `FileInputStream` legge un byte da un file, `System.in` legge byte a byte dalla keyboard

La classe FILE

un'istanza della classe File descrive:

- path per l'individuazione del file o della directory
- non una semplice stringa, ma offre metodi
 - per verificare l'esistenza del path
 - per restituire meta-informationi sul file,...
- quando si vuole stabilire una connessione (stream) con un file si può passare come parametro:
 - un oggetto di tipo File
 - una stringa
 -

TRY with resources

```

try {input = new FileInputStream("file.txt");
    int data = input.read();
    while(data != -1){
        System.out.print((char) data);
        data = input.read(); }
} finally {
    if(input != null){
        input.close(); } }

```

USANDO i TRY-CATCH CLASSICI VENNEBBERO PROPAGATE

due eccezioni e viene prop. nella stack

del chiamante solo quella delle finally!

Quindi si usano i try-catch dove specifichiamo le nostre I/O che vogliamo utilizzare.

```

try( FileInputStream input = new FileInputStream(new
File("immagine.jpg"));
BufferedInputStream bufferedInput = new
BufferedInputStream(input))
{
    int data = bufferedInput.read();
    while(data != -1){
        System.out.print((char) data);
        data = bufferedInput.read(); }
}

```

Java cloni filtri

Sono cloni che compiono trasformazioni sui dati a basso livello.

Filter Stream: trasformazioni effettuate

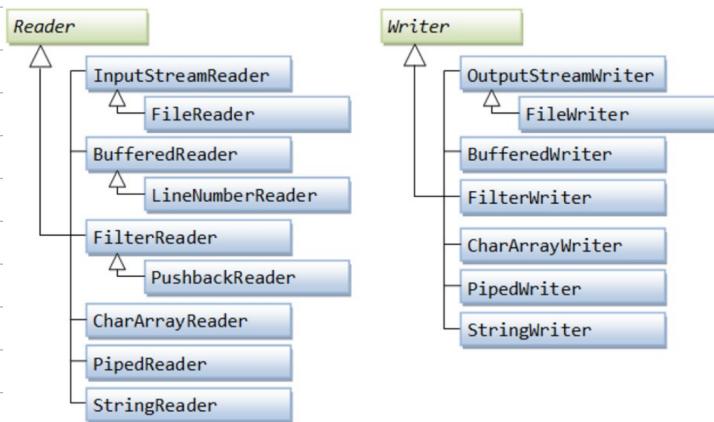
- crittografia
- compressione
- Buffering \Rightarrow i dati vengono scritti e letti in blocchi di bytes, invece che un solo blocco per volta (Migliora la performance)
- traduzione dei dati in un formato a più alto livello

Readers Writes

- orientati al testo e permettono di decodificare bytes in caratteri
- I filtri possono essere organizzati in catena. Ogni elemento della catena
- riceve dati dallo stream o dal filtro precedente
 - passa i dati al programma o al filtro successivo

```
FileOutputStream outputFile = new FileOutputStream("primitives.data");
BufferedOutputStream bufferedOutput = new BufferedOutputStream(outputFile);
```

CHARACTER STREAMS



Character Set

Coded Character Set: un character set in cui ad ogni carattere viene assegnato un valore numerico, o code point

Encoding: come i numeri che appartengono ad un coded character set sono rappresentati: 8 bit, una parola, una sequenza di bytes,...

coded character sets comuni:

- ASCII (128 caratteri, 7 bits encoding all'interno di un byte)
- ISO 8859-1 (caratteri identici as ASCII da 0 a 128, altri caratteri da 160 a 255, 8 bit encoding, utilizzato da sistemi UNIX-based)
- Windows 1252 (CP 1252) (come ASCII da 0 a 128, come ISO 8859 da 160 a 255 + altri caratteri, 8 bit encoding, utilizzato in Windows).
- Unicode Universal character set, gestito dall'Unicode Consortium (unicode.org)

INPUT STREAM READER

deve essere istanziato su un InputStream

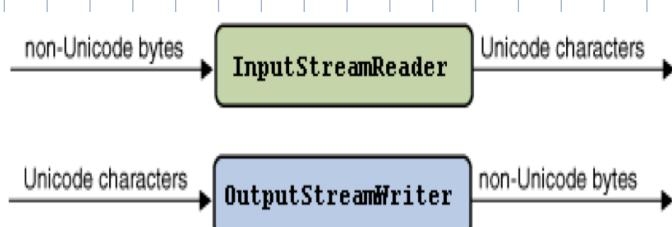
prende come parametro la codifica dei caratteri presenti sullo stream di byte

permette di specificare diversi encoding (UTF-8, UTF-16,...)

default charset

- settato dalla JVM al momento dello start-up
- dipendente dal sistema operativo

```
System.out.println("Codifica"+Charset.defaultCharset().displayName());
> windows-1252"
```

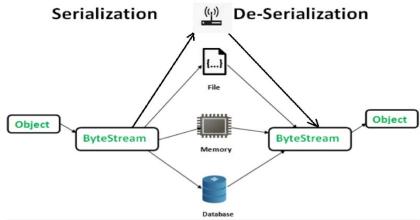


SERIALIZZAZIONE DI OGGETTI

Corsiste nell'andare ad effettuare il flattening dello stato dell'oggetto.

La de-serializzazione invece carica nel ricreare lo stato dell'oggetto.

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



Era viene utilizzata

1) Per inviare oggetti : a) Su uno Stream su connessione TCP

b) Come parametri di metodi: invocati via

Remote Method Invocation

2) Per generare pacchetti: UDP , si salva l'oggetto serializzato su uno stream di byte e poi si genera un pacchetto UDP

Come effettuarla (Serializzazione)

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

```

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

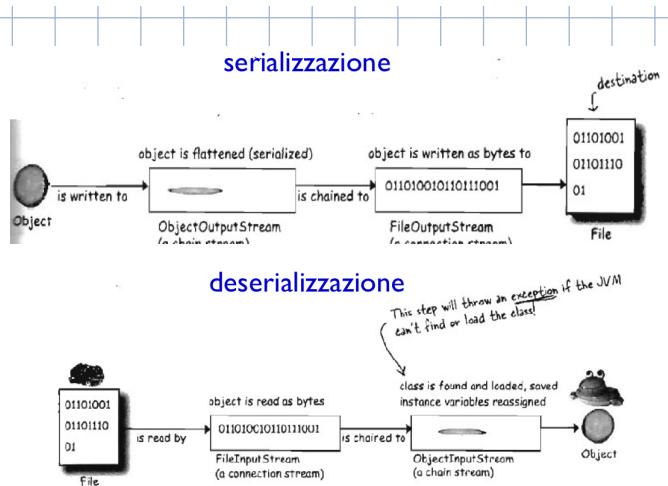
- la serializzazione vera e propria è gestita dalla classe `ObjectOutputStream`
- tale stream deve essere concatenato con uno stream di bytes, che può essere un `FileInputStream`, uno stream di bytes associato ad un socket, uno stream di byte generato in memoria....

```

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

in rosso le parti relative alla serializzazione

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



- La serializzazione è un processo ricorsivo, l'oggetto serializzato può contenere altri oggetti
- quando un oggetto viene serializzato, si percorre la gerarchia delle superclassi e si salva lo stato di ogni classe, fino a che non si trova la prima classe non serializzabile

DESERIALIZZAZIONE

```
public class InflateTime
{public static void main(String [] args)
{String filename = "time.ser";
if(args.length > 0) {filename = args[0]; }
PersistentTime time = null; FileInputStream fis = null;
ObjectInputStream in = null;
try(
    FileInputStream fis = new FileInputStream(filename);
    ObjectInputStream in = new ObjectInputStream(fis);
    {time = (PersistentTime)in.readObject();})
catch(IOException ex)
{ex.printStackTrace(); }
catch(ClassNotFoundException ex)
{ex.printStackTrace();}
```

in rosso le parti relative alla **deserializzazione**

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

Output ottenuto:

Flattened time: Mon Mar 12 19:11:55 CET 2012
Current time: Mon Mar 12 19:16:24 CET 2012

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

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

Cosa non è serializzabile?

- 1) Oggetti contenenti riferimenti specifici alle JVM o al SO (`Thread, OutputStream, Socket, File`)
- 2) le var. mancate come transient
- 3) le variabili statiche (sono associate alla classe e non all'oggetto)

OSS: tutte le componenti di un oggetto devono essere serializzabili *senza* \Rightarrow Eccezione `notSerializableException`

Mecanismi serializzazione standard Java

1) Caching

- ogni volta che un oggetto viene serializzato e inviato ad una `ObjectOutputStream`, un suo riferimento viene memorizzato in una "identity hash table"
- se l'oggetto viene scritto nuovamente sull' `OutputStream`, non viene nuovamente serializzato, viene inserito un puntatore all'oggetto precedente
 - minimizzazione e scrittura e risoluzione di relazioni circolari tra oggetti
- comportamento analogo, quando si legge da uno stream: l'oggetto letto viene memorizzato in una "identity hash table", la prima volta
 - lettura future fanno riferimento allo stesso oggetto
- possibili inconsistenze quando lo stato dell'oggetto viene modificato
 - la modifica viene persa, perché il riferimento all'oggetto rimane il solito, anche se il suo stato è stato modificato.
 - problema nel caso di invio di uno stream di oggetti su una connessione di rete

2) Controllo delle versioni

- per deserializzare un oggetto occorre conoscere
 - i byte che rappresentano l'oggetto serializzato
 - il codice della classe che descrive la specifica dell'oggetto
- la deserializzazione può avvenire in un ambiente diverso, ad esempio
 - mediante l'utilizzo di un compilatore diverso
 - a distanza di tempo rispetto al momento in cui è stata effettuata la serializzazione
 - su un computer diverso, a cui è stato mandato l'oggetto serializzato tramite una connessione di rete
- cosa succede se la classe utilizzata per la serializzazione cambia quando l'oggetto viene deserializzato?

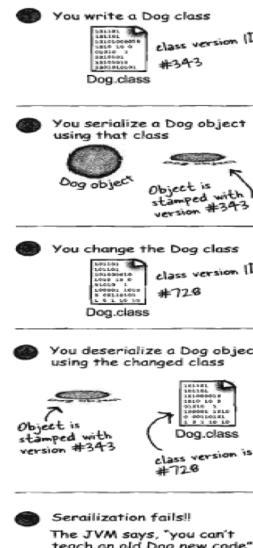
```
public class Employee {
    private String id;
    private String name;
    private int age;}
```

- supponiamo di serializzare i dati di un insieme di impiegati utilizzando la classe precedente
 - successivamente la classe viene modificata come segue
- ```
public class Employee {
 private String id;
 private String name;
 private Date dateOfBirth; }
```
- si può utilizzare la classe modificata per deserializzare un oggetto che è stato serializzato con la classe prima della modifica?

## SerialVersionUID (SUID)

- identificatore unico che identifica una classe
- utilizzato per verificare che la compatibilità tra le classi usate per la serializzazione e la deserializzazione, in fase di deserializzazione
- può essere gestito in due modi diversi
- identificatore implicito:** generato dal compilatore, non specificato dall'utente
  - 64-bit hash (SHA) generato a partire dalla struttura della classe, durante la serializzazione (nome della classe, nomi delle interfacce, metodi e campi)
  - in alcuni casi, compilatori diversi possono generare identificatori diversi per la stessa classe
- identificatore esplicito:** il programmatore gestisce esplicitamente la compatibilità delle classi, gestendo i loro identificatori. Ma come si generano gli identificatori?
  - dichiarato a scelta dal programmatore
  - usando l'IDE Eclipse
  - con un generatore a linea di comando

## SUID IMPLICITO



- la classe usata per la serializzazione può essere modificata, ma essere sempre **backward-compatible**
  - aggiungere attributi e/o oggetti
  - trasformare attributi transient in non-transient
  - cambiare una variabile di istanza in static
  - e molti altri cambiamenti.....
- in fase di deserializzazione, il meccanismo di default semplicemente imposta valori dei campi mancanti con valori di default
- il programmatore può "fixare" i valori dei campi aggiunti all'oggetto deserializzato
- modifiche che rendono la classe non backward-compatible
  - rimuovere attributi
  - trasformare attributi non-transient in transient

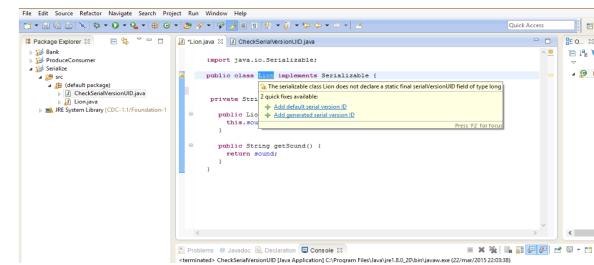
## SUID ESPlicito

- il programmatore può generare esplicitamente `serialVersionUID` per le classi interessate alla serializzazione/deserializzazione
  - classi compatibili: stesso `serialVersionUID` in entrambe le classi
  - classi incompatibili: diverso `serialVersionUID` per la classe modificata
- il supporto:
  - controlla se l'utente ha dichiarato esplicitamente il `serialVersionUID` ed, in questo caso, usa questo valore.
  - altrimenti genera un identificatore implicito
- generazione esplicita di identificatori
  - mediante tool automatici di JAVA, dato il nome della classe restituiscono un identificatore unico della classe
- dichiarazione identificatori espliciti
 

```
private static final long serialVersionUID = 42L;
```

generazione di `serialVersionUID` unico mediante l'algoritmo utilizzato da JAVA:

- in Eclipse puntare il mouse sul nome di una classe `Serializable` (`addGeneratedSerialVersionUID`)
- in altri ambienti: usare tool di generazione specifici (`serialver`)



- infine JAVA suggerisce di indicare esplicitamente un `SerialVersionUID`
- da JAVADOC: "the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization"
- spesso si ottiene una eccezione anche se le classe utilizzate in fase di serializzazione e deserializzazione sono in realtà le stesse.
  - compilatori diversi generalmente lo stesso serialVersionUID per la stessa classe
- per questo è consigliato di specificare comunque esplicitamente un `serialVersionUID`

## Svantaggi: Serializzazione

La serializzazione registra un descrittore dell'oggetto di dimensione significativa

- i "magic data"
  - `STREAM_MAGIC = "acde"`
  - `STREAM_VERSION = versione della JVM`
- i metadati che descrivono la classe associata all'istanza dell'oggetto serializzato
  - la descrizione include il nome della classe, il serialVersionUID della classe, il numero di campi, altri flag.
- i metadati di eventuali superclassi, fino a raggiungere `java.lang.Object`
- i valori associati all'oggetto istanza della classe, partendo dalla super classe a più alto livello
- i dati degli oggetti eventualmente riferiti dall'oggetto istanza della classe, iniziando dai metadati e poi registrando i valori. (Le istanze della classe Figlio, nell'esempio precedente).
- non si registrano i metodi della classe

- oggetto serializzato aumenta molto di dimensione rispetto all'oggetto originario
- limitata interoperabilità
  - utilizzabile solo quando sia l'applicazione che serializza l'oggetto che quella che lo deserializza sono scritte in JAVA
- per aumentare l'interoperabilità occorre utilizzare altre soluzioni:
  - il formato standard JSON (presentato in una prossima lezione...)
  - serializzazione in XML
  - ...altri formati....

## Java NIO

**Block-oriented:** Ogni operazione produce o consuma dei blocchi di dati

- obiettivi:
  - incrementare la performance dell'I/O, senza dover scrivere codice nativo
  - input ad alte prestazioni da file, network socket, piped I/O
  - aumentare l'espressività delle applicazioni
- vantaggi
  - miglioramento di performance: definizione di primitive "più vicine" al livello del sistema operativo
- svantaggi
  - risultati dipendenti dalla piattaforma su cui si eseguono le applicazioni
  - primitive a più basso livello di astrazione: perdita di semplicità ed eleganza rispetto allo stream-based I/O
  - ma anche primitive espressive, ad esempio per lo sviluppo di applicazioni che devono gestire un alto numero di connessioni di rete.

## Costituti di base

**canali:** - Sono analoghi ad uno Stream in Java.Io, tutti i dati inviati o letti

su un canale devono essere memorizzati in un buffer

- Collega da/verso dispositivi esterni (è bidirezionale)
- interazione con i canali
  - trasferimento dati dal canale nel buffer, quindi programma legge il buffer
  - il programma scrive nel buffer, quindi trasferimento dati dal buffer al canale

**Buffer:** Contenitore di dati

**Selector:** Oggetto che monitora un insieme di canali

## leggere da un canale

il canale è associato ad un FileInputStream

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

creazione di un ByteBuffer

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

lettura dal canale al Buffer

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

### Osservazioni:

- non è necessario specificare quanti byte il sistema operativo deve leggere nel Buffer
- necessarie delle variabili interne all'oggetto Buffer che mantengano lo stato del Buffer

## Scrivere sul canale

il canale è associato ad un FileOutputStream

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

creazione del Buffer per scrivere sul canale

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

copia del messaggio nel Buffer

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

per indicare quale porzione del Buffer è significativa occorre modificare le variabili interne di stato (vedi lucidi successivi), quindi si scrive sul canale  
buffer.flip();  
fc.write( buffer );

## Variabili di stato

- Capacity
  - massimo numero di elementi del Buffer
  - definita al momento della creazione del Buffer, non può essere modificata
  - java.nio.BufferOverflowException, se si tenta di leggere/scrivere in/da una posizione > Capacity
- Limit
  - indica il limite della porzione del Buffer che può essere letta/scritta
    - per le scritture= capacity
    - per le letture delimita la porzione di Buffer che contiene dati significativi
  - aggiornato implicitamente dalla operazioni sul buffer effettuate dal programma o dal canale
- Position
  - come un file pointer per un file ad accesso sequenziale
  - posizione in cui bisogna scrivere o da cui bisogna leggere
  - 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
  - tentativi di resettare un mark undefined sollevano
   
`java.nio.InvalidMarkException.`

**OSSERVAZIONE:**  $0 \leq \text{mark} \leq \text{position} \leq \text{limit} \leq \text{capacity}$

## Metodi:

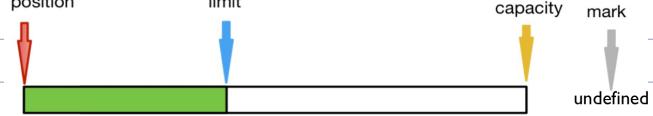
1) `mark()`: Ricorda la posizione corrente



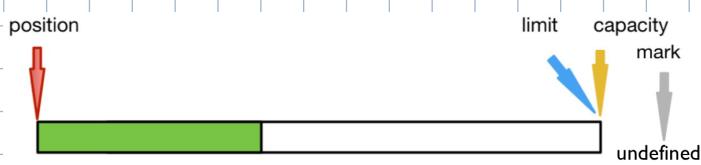
2) `reset()`: Restetta position alla posizione segnata da mark



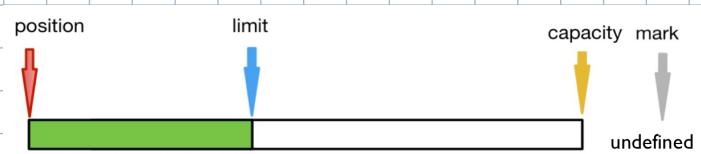
3) `flip()`: Predisponde il buffer alle letture dopo la scrittura



4) `clear()`: Non elimina i dati nel buffer e resetta i contatori

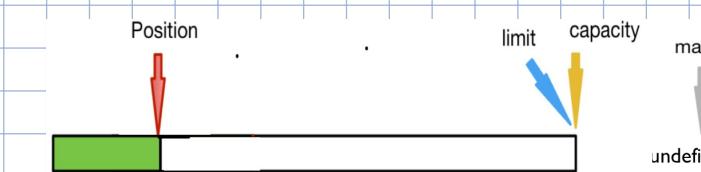


5) `Rewind()`: Riporta position a 0, ma lascia limit invariato  
(Utile per rileggere dati già letti.)



6) `compact()`: I bytes non letti vengono copiati ad inizio buffer

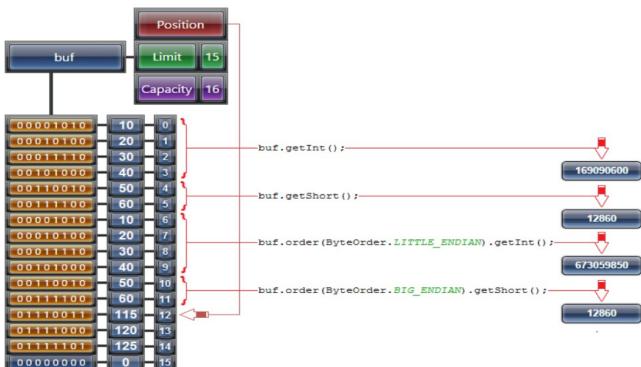
utile se il contenuto del buffer non è stato completamente letto e si inizia una nuova scrittura



7) `remaining()`: Restituisce il numero di elementi nel buffer

8) `hasRemaining()`: Restituisce true se `remaining() > 0`

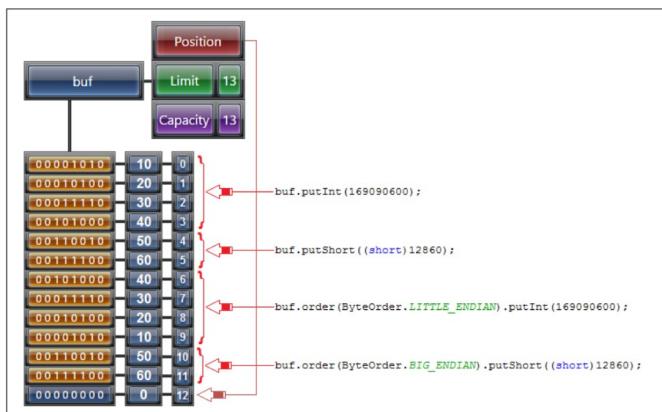
Leggere dati di tipo primitivo



```
int fileSize = byteBuffer.getInt();
```

- estrae quattro bytes dal buffer, iniziando dalla posizione corrente li combina per comporre un intero a 32-bits
- metodi analoghi per gli altri tipi di dato primitivi

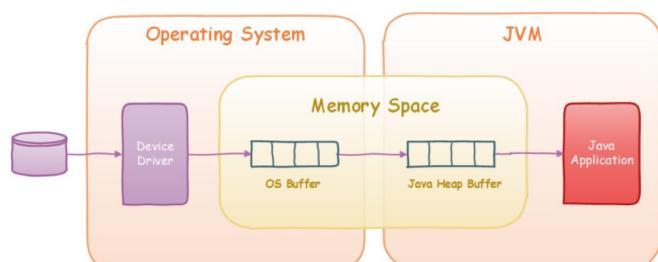
Scrivere dati di tipo primitivo



- inserisce nel buffer un valore intero
- metodi analoghi per gli altri tipi di dato primitivi

Tipi di buffer:

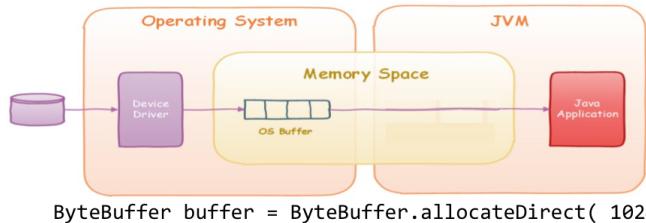
Nou direct Buffers



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

- crea sullo heap un oggetto Buffer, che incapsula una struttura per memorizzare gli elementi + variabili di stato
- doppia copia dei dati

## Direct Buffers



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

## Wrapping Buffers

ByteBuffer. wrap() => Metodo statico che crea un buffer, ma non alloca memoria per gli elementi.

Usa un vettore precedentemente allocato come Backing Storage

- l'oggetto Buffer è distinto dalla memoria utilizzata per i suoi elementi
- ogni modifica al buffer è visibile nell'array e viceversa.

## View Buffers

Potrete specificare il tipo dei dati di un buffer

IntBuffer intBuf = buf.asIntBuffer(); => Dove buf è un byteBuffer.

## Canali:

L'interfaccia 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.

## Caratteristiche:

- sono bidirezionali
- i dati sono gestiti mediante buffers
- possono essere bloccanti e non bloccanti

## FileChannels

Apri un canale verso un file e lavorare su byte (lettura e scrittura richiedono un byte buffer).

Sono bloccanti e thread safe:

- più thread possono lavorare in modo consistente sullo stesso channel
- alcune operazioni possono essere eseguite in parallelo (esempio: read), altre vengono automaticamente serializzate
- ad esempio le operazioni che cambiano la dimensione del file o il puntatore sul file vengono eseguite in mutua esclusione

Create un file channel

1) Scrittura : 

```
WritableByteChannel dest =
 Channels.newChannel (new FileOutputStream("out.txt"));
```

2) Lettura : 

```
ReadableByteChannel source =
 Channels.newChannel(new FileInputStream("in.txt"));
```

Differenze tra Stream e buffer:

Stream:

- reader.readLine()
  - quando restituisce il controllo al chiamante, una linea di testo è stata letta
  - si blocca fino a che la linea è stata completamente letta
- ad ogni passo, il programma sa quali dati sono stati letti
- dopo aver letto dei dati, non si può tornare indietro sullo stream
- lettura di uno o più bytes alla volta
- meccanismo di bufferizzazione a livello di applicazione: possibile con byteArray
- caching possibile a livello del supporto  
(BufferedReader, BufferedInputStream,...)

Buffer:

- int bytesRead =  
 inChannel.read(buffer);
- l'applicazione deve verificare se sono stati letti abbastanza dati
- verifica ripetuta anche diverse volte

```
ByteBuffer buffer = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buffer);

while(! bufferFull(bytesRead)) {
 bytesRead = inChannel.read(buffer);
}
```

- buffer è pieno: si procede alla elaborazione
- buffer non pieno: si può decidere di elaborare o meno i dati letti, si controlla iterativamente lo stato del buffer
- buffering di dati a livello della applicazione
- gestione del buffer a carico del programmatore:
  - controllo della disponibilità dei dati richiesti
  - controllo che nuovi dati non sovrascrivano dati non ancora elaborati.

## Direct channel transfer

E' possibile connettere due canali e trasferire direttamente dati dall'uno all'altro.  
Uno dei due channel deve essere un FileChannel.

### Esempio:

```
import java.nio.channels.FileChannel;
import java.nio.channels.WritableByteChannel;
import java.nio.channels.Channels;
import java.io.FileInputStream;

public class ChannelTransfer
{ public static void main (String [] argv) throws Exception
 { if (argv.length == 0) {
 System.err.println ("Usage: filename ...");
 return; }
 catFiles (Channels.newChannel (System.out), argv);
 // Concatenate the content of each of the named files to the given
 channel. A very dumb version of 'cat'.
 }
}
```

```
private static void catFiles (WritableByteChannel target,
 String [] files) throws Exception
{ for (int i = 0; i < files.length; i++)
 { FileInputStream fis = new FileInputStream (files [i]);
 FileChannel channel = fis.getChannel();
 channel.transferTo (0, channel.size(), target);
 channel.close();
 fis.close();
 }
}
Input:
i file di testo primo.txt contenente "questo corso di Laboratorio di Reti "
e secondo.txt contenente "è veramente bello!"
Output prodotto:
questo corso di Laboratorio di Reti è veramente bello!
```

## JSON

JSON è un formato light weight per l'intercambio di dati. Un file JSON ha una struttura ad albero.

E' basato su 2 strutture:

- 1) Copie (chiave: valore) → tipi di dato ammissibili.
- 2) liste ordinate di valori.

una raccolta ordinata di valori

["Ford", "BMW", "Fiat"]

delimitato da parentesi quadre e i valori sono separati da virgola.

- un valore può essere di tipo string, un numero, un boolean, un oggetto o un array.
- queste strutture possono essere annidate.

- String
- Number (int o float)
- object (JSON object, la struttura può essere ricorsiva)
- Array
- Boolean
- null

## JSON Object

Sono una serie di coppie non ordinate delimitata da parentesi graffe e separate da virgolette.

```
{
 "name": "John",
 "age": 30,
 "car": null
}
```

OSS: puoi avere

oggetti annidati:

```
{ "name": "John",
 "age": 30,
 "cars": {
 "car1": "Ford",
 "car2": "BMW",
 "car3": "Fiat"
 } }
```

## Jackson

Jackson è una libreria che permette di serializzare e deserializzare oggetti Java in/da JSON.

### Object Mapper :

- prende in ingresso un file o stringa JSON e crea un oggetto o un grafo di oggetti (deserializzazione di oggetti Java da JSON).
- usato anche per la serializzazione

`WriteValue()` e `ReadValue()`: Per convertire oggetti Java a/da JSON.

(Sono metodi dell' Object Mapper )

```
<T> T readValue(Reader src, Class<T> valueType)
<T> T readValue(String content, Class<T> valueType)
void writeValue(Writer w, Object value)
void writeValueAsString(Object value)
```

`ReadTree()`: Quando non si conosce il tipo eretto di oggetto, il parsing restituisce un oggetto `JsonNode`.

```
JsonNode readTree(File file)
JsonNode readTree(InputStream in)
JsonNode readTree(String content)
```

## NETWORK APPLICATIONS

### Applicazione di rete:

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

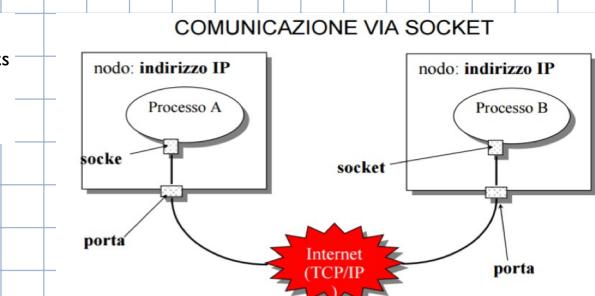
Socket: Standard per connettere dispositivi distribuiti, diversi, eterogeni.

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

Tipi di connessione:

#### 1) Connection-oriented :

- Connessione modellata come stream
- Asimmetrici
- Client side : `Socket` class
- Server side :
  - `ServerSocket` class
  - `Socket` class



#### 2) Connectionless:

Dati su un socket ma per client  
che per server (simmetrici)

## Identificazione di un processo

- la rete all'interno della quale si trova l'host su cui è in esecuzione il processo
- l'host all'interno della rete
- il processo in esecuzione sull'host

Comunicazione: è un 5-upla:

- il protocollo (TCP o UDP)
- l'indirizzo IP del computer locale  
(client sky3.cm.deakin.edu.au, 139.130.118.5)
- la porta locale esempio: 5101
- l'indirizzo del computer remoto  
(server res.cm.deakin.edu.au 139.130.118.102),
- la porta remota: 5100

{tcp, 139.130.118.102, 5100, 139.130.118.5, 5101}

## Osservazioni:

1) Un server può offrire più servizi su porte diverse

2) Le porte sono suddivise in:

0-1023: Well-known

1024-49151: Registered

49152 - 65535: Not Registered

## Classe INET ADDRESS

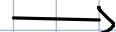
La classe INET ADDRESS rappresenta un indirizzo IP, il record contiene:

- String hostName: una stringa che rappresenta il nome simbolico di un host
- byte[] address: un vettore di bytes che rappresenta l'indirizzo IP dell'host

Non ha nessun costruttore, ma offre tre metodi statici:

```
public static InetAddress getByname (String hostname)
 throws UnknownHostException
public static InetAddress getAllByName(String hostname)
 throws UnknownHostException
public static InetAddress.getLocalHost()
 throws UnknownHostException
```

- reperire nome simbolico ed indirizzo IP del computer locale



- cerca l'indirizzo IP corrispondente all'host il cui nome è passato come parametro e restituisce un oggetto di tipo InetAddress
- richiede una interrogazione del DNS per risolvere il nome dell'host
  - l'host deve essere connesso in rete
  - può utilizzare una cache locale, in questo caso ricerca prima nella cache
- può sollevare UnknownHostException, se non riesce a risolvere il nome dell'host
- reverse lookup: passo un indirizzo IP come parametro, nella forma dotted quad

per hosts che possiedano più indirizzi (es: web servers)

Getters classe INET ADDRESS senza collegamenti con DNS:

```
public String getHostName ()
public byte [] getAddress ()
public String getHostAddress ()
```

Metodi sovraccaricati della classe object:

- equals(): due oggetti InetAddress sono uguali se e solo se
  - hanno lo stesso indirizzo IP
  - non necessariamente devono avere lo stesso hostname
- hashCode()
  - converte 4 bytes dell'indirizzo IP in un int
  - coerente con equals, non considera hostname
- toString()
  - restituisce nome dell'host/indirizzo dotted quad
  - se non esiste il nome, stringa vuota + indirizzo dotted quad

## Caching: Prima di interrogare il DNS, controlla la cache

permanenza dati nella cache:

- 10 secondi se la risoluzione non ha avuto successo, spesso il primo tentativo di risoluzione fallisce a causa di un time out...
- tempo illimitato altrimenti. Problemi: indirizzi dinamici.

`java.security.Security.setProperty` imposta il numero di secondi in cui una entrata nella cache rimane valida,

```
java.security.setProperty
("networkaddress.cache.ttl","0");
```

↓  
non vengono salvati  
in cache

## Network Interface

Un'interfaccia che collega allo host una rete. Ema è caratterizzata da un indirizzo IP.

```
public static NetworkInterface getByAddress (InetAddress address)
throws SocketException
```



restituisce un oggetto NetworkInterface che rappresenta la network interface collegata ad un indirizzo IP ( o null)

## Indirizzo di loopback

### 127.0.0.1 Indirizzo di loopback

- utilizzabile per testare applicazioni
- quando si usa un indirizzo di loopback si possano eseguire client e server in locale, sullo stesso host
- attivati da due shell diverse, o due progetti diversi Eclipse
- ogni dato spedito utilizzando l'indirizzo di loopback in realtà non lascia l'host locale
- il dato viene restituito all'host locale stesso, sulla porta opportuna

## Paradigma Client-SERVER

### Socket lato Server:

- **welcome (passive, listening) sockets:** utilizzati dal server per accettare le richieste di connessione
- **connection (active) sockets:** supportano lo streaming di byte tra client e server

### Socket lato Client: Active Socket per richiedere una connessione.

quando il server accetta una richiesta di connessione,

- crea a sua volta **un proprio active socket AS** 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

*Collegamento:*

a) il server pubblica un proprio servizio

- associato al listening socket, creato sulla porta remota PR, all'indirizzo IP SA
- usa un oggetto di tipo ServerSocket

b) il client che intende usufruire del servizio deve conoscere

- l'indirizzo IP del server, e la porta remota, PR, a cui è associato il servizio
- usa un oggetto di tipo Socket

c) la creazione del socket effettuata dal client produce in modo atomico la richiesta di connessione al server

- completamente gestito dal supporto
- three way handshake
- se la richiesta viene accettata,
- il server crea un socket dedicato per l'interazione con il client
- tutti i messaggi spediti dal client vengono diretti automaticamente sul nuovo socket creato.

## Stream Socket API : lato Client

```
public Socket(InetAddress host, int port) throws IOException
 • crea un active socket e tenta di stabilire, tramite esso, una connessione con l'host individuato da InetAddress, sulla porta port.
 • se la connessione viene rifiutata, lancia una eccezione di IO

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

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

### Altri costruttori della Classe java.net.Socket

```
public Socket(String H, int P, InetAddress IA, int LP)
```

tenta di creare una connessione

- verso l'host H,
- sulla porta P.
- dalla interfaccia locale IA
- dalla porta locale LP

## Stream Socket API : lato Server

```
java.net.ServerSocket: costruttori
public ServerSocket(int port) throws BindException, IOException
public ServerSocket(int port, int length) throws BindException, IOException
 • costruisce un listening socket, associandolo alla porta p.
 • length: lunghezza della coda in cui vengono memorizzate le richieste di connessione.
 se la coda è piena, ulteriori richieste di connessione sono rifiutate

public ServerSocket(int port, int length, InetAddress bindAddress).....
 • permette di collegare il socket ad uno specifico indirizzo IP locale.
 • utile per macchine dotate di più schede di rete, ad esempio un host con due indirizzi IP, uno visibile da Internet, l'altro visibile solo a livello di rete locale
 • se voglio servire solo le richieste in arrivo dalla rete locale, associo il connection socket all'indirizzo IP locale
 accettare una nuova connessione dal connection socket
```

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

metodo della classe ServerSocket.

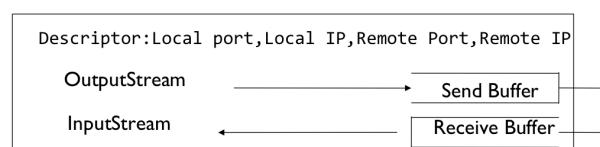
- quando il processo server invoca il metodo accept(), pone il server in attesa di nuove connessioni.
- se non ci sono richieste, il server si blocca (possibile utilizzo di time-outs)
- se c'è almeno una richiesta, il processo si sblocca e costruisce un nuovo socket S tramite cui avviene la comunicazione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

## Modellare una connessione mediante Stream: ( Bloccante )

- associare uno stream di input o di output ad un socket:

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

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



Struttura del Socket TCP

## Modellare la connessione tramite channels (Non bloccante):

La chiamata di sistema restituisce il controllo all'applicazione prima che l'operazione richiesta sia stata "pienamente soddisfatta".

Possibili scenari:

- restituiti i dati disponibili, o una parte di essi
- operazione I/O non possibile, restituito un codice errore o valore null

Si associano i channels ai socket:

un channel associato ad un socket TCP

- "combinazione" di un socket e canale di comunicazione bidirezionale
- scrive e legge da un socket TCP
- estende la classe AbstractSelectableChannel e da questa mutua la capacità di passare dalla modalità bloccante a quella non bloccante
- in modalità bloccante funzionamento simile a quello degli stream socket, ma con interfaccia basata su buffers
- ogni SocketChannel è associato ad un oggetto Socket della libreria java.net
  - il socket associato può essere reperito mediante il metodo socket()
- blocking: come ServerSocket, ma con interfaccia buffer-based
- non blocking: permette multiplexing di canali

Lato Server

Associazione un ServerSocket a un serverSocketChannel:

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
ServerSocket socket = serverSocketChannel.socket();
```

Rendere ServerSocketChannel non bloccante:

```
serverSocketChannel.configureBlocking(false);
```

Si usa la classe InetSocketAddress al posto di InetAddress:

```
socket.bind(new InetSocketAddress(9999));
```

InetSocketAddress: rappresenta l'indirizzo di un socket descritto da indirizzo IP e numero di porta, alternativo rispetto InetAddress

Lato client

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

modalità blocking/non blocking:

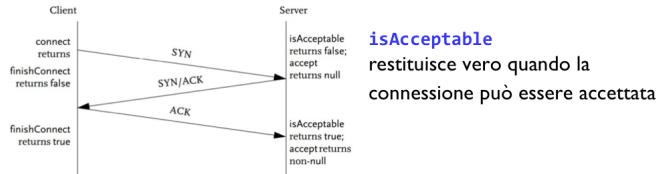
```
SocketChannel.configureBlocking(false);
```

non blocking, lato client, significativa ad esempio nel caso in cui:

- un'applicazione lato client che deve gestire l'interazione con l'utente, mediante GUI e contemporaneamente, gestire uno o più sockets

## Nou blocking connect (lato client)

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



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

```

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

```

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

se si toglie

```

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

```

viene sollevata `java.nio.channels.NotYetConnectedException`

Utile se un client deve gestire

→ sia la Gui che un socket

### Riassunto Blocking e non Blocking:

- **Blocking accept:** si blocca finché non arriva una richiesta di connessione
- **Non-Blocking accept:** controlla se c'è una richiesta da accettare (se c'è accetta) e ritorna
- **Blocking write:** si blocca finché la scrittura dei dati nel buffer non è completata
- **Non-Blocking write:** tenta di scrivere i dati nella socket, ritorna immediatamente, anche se i dati non sono stati completamente scritti
- **Blocking read:** si blocca in attesa di byte da leggere
- **Non-blocking read:** ritorna immediatamente e restituisce il numero di byte letti (anche 0)

## Modelli di Servers

Criteri per la valutazione delle prestazioni di un server:

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

### UN SINGOLO THREAD

Un solo thread per tutti client:

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

### UN THREAD PER OGNI CONNESSIONE

- **scalabilità:** possibile servire diversi clienti in maniera concorrente, fino al massimo numero di thread previsti per ogni processo
  - ogni thread aloca il proprio stack: memory pressure
  - impossibile predirne il numero massimo di client: dipende da fattori esterni e può essere molto variabile
- **accept latency:** tempo tra l'accettazione di una connessione e la successiva è in genere basso rispetto a quello di interarrivo delle richieste
- **reply latency:** bassa, le risorse del server condivise tra connessioni diverse
  - ragionevole uso di CPU e RAM per centinaia di connessioni, se aumenta, il tempo di reply può non essere accettabile
- **efficiency:** bassa
  - ogni thread può essere bloccato in attesa di IO, ma utilizza risorse come la RAM
  - attivazione di un thread per ogni connessione, de-attivazione a fine servizio
  - quando un server monitora un grande numero di comunicazioni:
    - problemi di scalabilità: il tempo per il cambio di contesto può aumentare notevolmente con il numero di thread attivi
    - maggior parte del tempo impiegata in context switching

### UN NUMERO FISSO DI THREAD

Un numero costante di thread: utilizza Thread Pool

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

Fixed  
thread  
(Pool)

## MULTIPLEXED I/O

I/O non bloccante con modifiche bloccanti.

Dice al kernel quali operazioni gli interverranno sui canali e successivamente li monitora:

- 1) Chiama una System call bloccante.
- 2) Fin quando non c'è un'operazione di I/O "pronta" aspetta (Richiesta di connessione, lettura / scrittura ecc..)
- 3) A quel punto effettua l'operazione di I/O in modo non bloccante.

Questo avviene tramite un **selettore**:

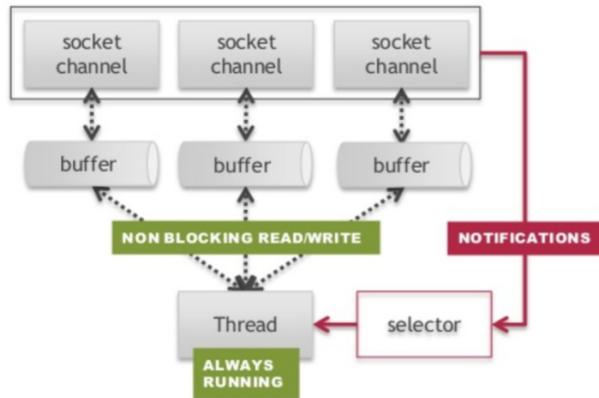
selettore un componente che esamina uno o più NIO Channels, e determina quali canali sono pronti per leggere/scrivere

Quindi avrei un unico thread che gestisce più connessioni di rete, questo permette di ridurre:

- thread switching overhead
- uso di risorse per thread diversi

**Multiplexing:**

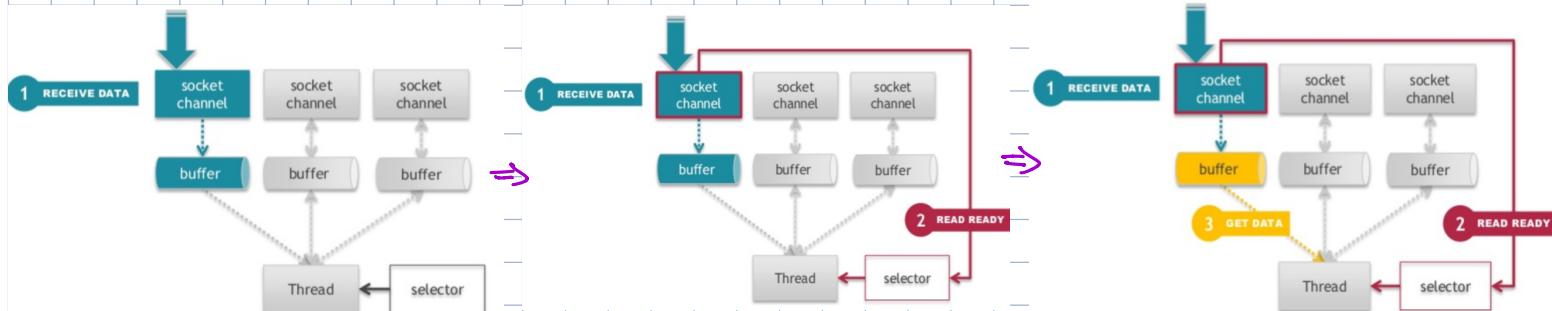
### NIO (non-blocking)



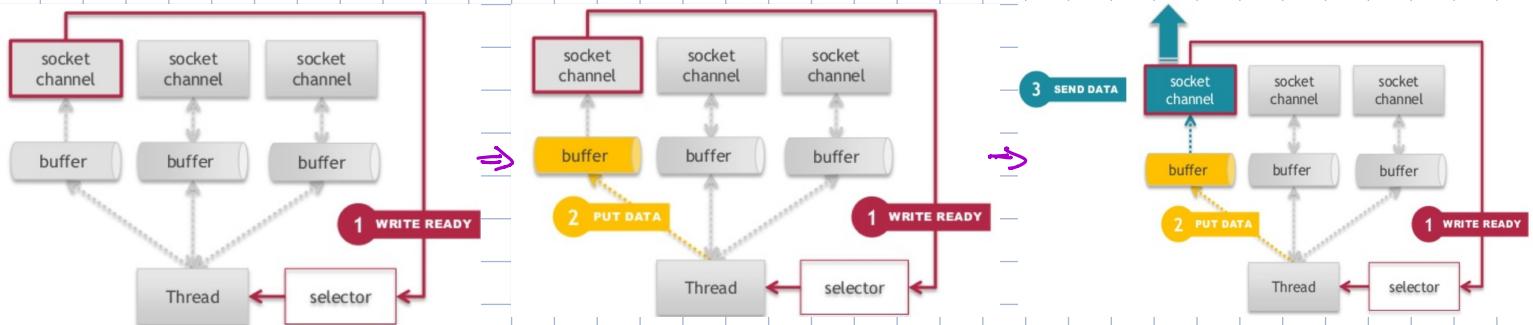
### I/O multiplexing

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

### Read



## WRITE



## Oggetto selector

Permette di selezionare un SelectableChannel che è pronto per operazioni di rete.

- accept, write, read, connect
- stesso thread che gestisce più eventi che possono avvenire simultaneamente

## Selectable channels:

- ServerSocketChannel
- SocketChannel
- DatagramChannel
- Pipe.SinkChannel
- Pipe.SourceChannels
- file I/O non inclusa

## Registrazione dei Canali (Selection Keys)

4 canali devono essere registrati su un selettore per op. specifiche.

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

Ogni registrazione di un canale su un selettore restituisce una chiave che la rappresenta.

La chiave è un oggetto di tipo **SelectionKey**. Valida fino a che non cancellata esplicitamente.

Lo stesso canale può essere registrato con più selettori (key diversa per ogni registrazione)

## 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
  - definisce le operazioni, del canale associato, su cui si deve fare il controllo di "readiness", la prossima volta che il metodo `select` verrà invocato per monitorare i canali del selettore
- il ready set
  - dopo la invocazione della `select`, contiene gli eventi che sono pronti su quel canale
  - si può leggere dal canale, si può scrivere, c'è una richiesta di connessione,...
- un allegato, attachment
- uno spazio di memorizzazione associato a quel canale

## Classe SelectionKey

```
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();
}
```

## Interest Set (bit Mask)

- rappresentato da una **bitmask** che codifica le operazioni per cui si registra un interesse su quel canale
- attualmente sono supportati 4 tipi di operazioni
  - connect
  - accept
  - read
  - write
- non tutte le operazioni valide per tutti i `SelectableChannel`, ad esempio `SocketChannel` non supporta `accept()`
- 4 costanti predefinite nella classe `SelectionKey`, rappresentano le operazioni, ognuna **corrisponde ad una bitmask**

```
1 SelectionKey.OP_CONNECT
2 SelectionKey.OP_ACCEPT
3 SelectionKey.OP_READ
4 SelectionKey.OP_WRITE
```

- Interest Set è manipolato con gli operatori JAVA |, &, ^, ~ che eseguono operazioni bit a bit su operandi interi o booleani
- in fase di registrazione del canale con il Selector si imposta il valore iniziale dell'Interest Set
 

```
Selector selector = Selector.open();
channel.register(selector,SelectionKey.OP_READ | SelectionKey.OP_WRITE);
```
- l'interest set è reperibile e può essere manipolato tramite gli operatori
 

```
int interestSet = selectionKey.interestOps();
boolean isInterestedInAccept =
(interestSet & SelectionKey.OP_ACCEPT) == SelectionKey.OP_ACCEPT
```
- oppure `selectionkey.interestOps(SelectionKey.OP_READ);`

| 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               |

## Ready Set (bit Mask)

- aggiornato quando si esegue una operazione di monitoring dei canali, mediante una `select` (vedi slide successive)
- identifica le chiavi per cui il canale è "pronto", per l'esecuzione
  - sottoinsieme dell'interest set
  - `interest set={read, write} ready set={read}`
- inizializzato a 0 quando la chiave viene creata
- non può essere modificato direttamente
- restituito dal metodo `readyOps()` invocato su una `SelectionKey`

```
if ((key.readyOps() & SelectionKey.OP_READ) != 0)
{ myBuffer.clear();
key.channel().read (myBuffer);
doSomethingWithBuffer (myBuffer.flip());}
```
- **shortcuts**
  - `key.isReadable()` equivale a `key.readyOps() & SelectionKey.OP_READ != 0`
  - analogo per le altre operazioni

# Registrazione di un canale (Pattern Generale)

```
// Crea il socket channel e configuralo come non bloccante

ServerSocketChannel server = ServerSocketChannel.open();
server.configureBlocking(false);
server.socket().bind(new java.net.InetSocketAddress(host, 8000));
System.out.println("Server attivo porta 2001");

// Crea il 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               |

## Metodo Select :

```
int selector.select();
```

- bloccante, seleziona, tra i canali registrati sul selettore selector, quelli pronti per almeno una delle operazioni di I/O dell'interest set.
- si blocca finché 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 chiavi dei canali pronti

```
int select(long timeout)
```

- si blocca fino a che non è trascorso il timeout, oppure valgono le condizioni precedenti

```
int selectNow()
```

- non bloccante, nel caso nessun canale sia pronto restituisce il valore 0

COSA FA LA SELECT

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

## Insieme di chiavi mantenute dal selettore:

- **Key Set**: contiene le SelectionKeys dei canali registrati con quel selettore.
  - restituite dal metodo keys()
- **Selected Key Set**: dopo che una select() consente di accedere ai canali pronti per l'esecuzione di qualche operazione
  - restituiti dal metodo selectedKeys(), invocato sul selettore
  - Selected Key Set è l'insieme di chiavi precedentemente registrate e per cui una delle operazioni nell'interest set è anche nel ready set della chiave
- **Cancelled Key Set** contiene la chiavi invalidate, quelle su cui è stato invocato il metodo cancel(), ma non ancora de-registrate
  - comportamento cumulativo" della selezione
- una chiave aggiunta al selected key set, può essere rimossa solo con una operazione di rimozione esplicita
- il ready set di una chiave inserita nel selected key set, non viene mai resettato, ma viene aggiornato incrementalmente
- scelta di progetto: assegnare al programmatore la responsabilità di aggiornare esplicitamente le chiavi
- per resettare il ready set
  - rimuovere la chiave dall'insieme delle chiavi selezionate

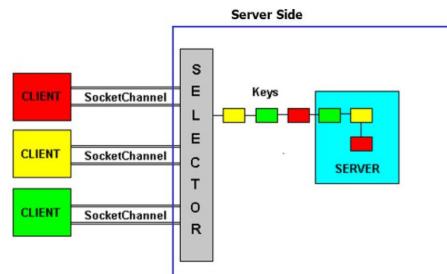
## Selezione (Pattern Generale) :

```

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

```

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



## UDP

UDP viene usato per comunicazioni veloci e dove un piccolo errore fa poca differenza.

UDP è connection-less ed utilizza i DatagramSocket  $\Rightarrow$  Non serve avere connessioni ad

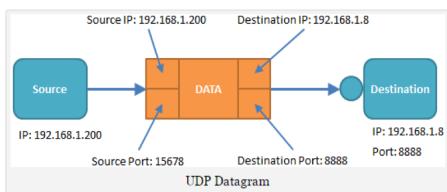
un altro socket prima di avere utilizzato.

connessione orientata ai messaggi

In UDP ogni messaggio chiamato Datagram ed è indipendente dagli altri e porta l'informazione per il suo indirizzamento.

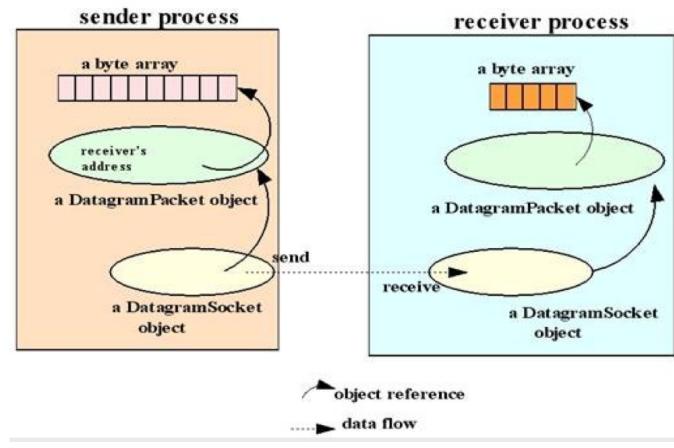
## Struttura del Datagram

**Datagram:** un messaggio indipendente, self-contained in cui l'arrivo ed il tempo di ricezione non sono garantiti, modellato in JAVA come un DatagramPacket



- il mittente deve inizializzare
  - il campo DATA
  - destination IP e destination port
- source IP inserito automaticamente, source port scelta automaticamente in modo casuale

## Come funziona UDP in Java



## Java Datagram Socket API

- **DatagramSocket** per creare i sockets.
- **DatagramPacket** per costruire i datagram

### Inviare un datagramma

- creare un **DatagramSocket** e collegarlo ad una porta, anche effimera
- creare un oggetto di tipo **DatagramPacket**, in cui inserire
  - un riferimento ad un byte array contenente i dati da inviare nel payload del datagramma
  - indirizzo IP e porta del destinatario nell'oggetto creato
- inviare il **DatagramPacket** tramite una send invocata sull'oggetto **DatagramSocket**

### Ricevere un datagramma

- creare un **DatagramSocket** e collegarlo ad una porta pubblicata (che corrisponde a quella specificata dal mittente nel pacchetto)
- creare un **DatagramPacket** per memorizzare il pacchetto ricevuto. Il **DatagramPacket** contiene un riferimento ad un byte array che conterrà il messaggio ricevuto.
- invocare una receive sul **DatagramSocket** passando il **DatagramPacket**

## CARATTERISTICHE SOCKET UDP

### LATO MITTENTE

un processo può utilizzare lo stesso socket per spedire pacchetti verso destinatari diversi

### LATO DESTINATARIO

- 1) processi (applicazioni) diverse possono spedire pacchetti sullo stesso socket allocato dal destinatario: in questo caso l'ordine di arrivo dei messaggi è non deterministico, in accordo con il protocollo UDP
- 2)...ma anche utilizzare socket diversi per comunicazioni diverse

## DATA GRAM SOCKETS CLASSI E COSTRUTTORI

### LATO MITTENTE

```
public class DatagramSocket extends Object
 public DatagramSocket() throws SocketException
```

- crea un socket e lo collega ad una porta **anonima** (o effimera), il sistema sceglie una porta **non utilizzata** e la assegna al socket.
- utilizzato generalmente lato client, per spedire datagrammi
- per reperire la porta allocata utilizzare il metodo `getLocalPort()`

Allocata  
temporaneamente

### LATO DESTINATARIO

```
public class DatagramSocket extends Object
 public DatagramSocket(int p) throws SocketException
```

- utilizzato in genere lato server.
- crea un socket sulla porta specificata (**int p**). PORTA ALLOCATA → PERM. AL SERVIZIO
- solleva un'eccezione quando la porta è già utilizzata, oppure se si tenta di connettere il socket ad una porta su cui non si hanno diritti.

## DATA GRAM PACKET CLASSI E COSTRUTTORI

### LATO MITTENTE

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

### LATO SERVER

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

costruttore per un `DatagramPacket` da inviare

`length` indica il numero di bytes che devono essere copiati dal byte buffer nel pacchetto IP, a partire dal byte 0 o da offset, se indicato

- solleva un'eccezione se `length` è maggiore di `buffer.length`
- il byte buffer può contenere più di `length` bytes, ma questi non verranno spediti sulla rete

`destination + port` individuano il destinatario

molti altri costruttori sono disponibili

notare che, per essere memorizzato nel buffer, il messaggio deve essere trasformato in una `sequenza di bytes`. Per generare vettori di bytes:

- il metodo `getBytes()` → Per le Stringhe
- la classe `java.io.ByteArrayOutputStream` → Per degli oggetti più complessi

- definisce la struttura utilizzata per memorizzare il pacchetto ricevuto.
- il buffer viene passato vuoto alla receive che lo riempie al momento della ricezione di un pacchetto, con il payload del pacchetto
- se settato offset, la copia avviene nelle posizioni individuate da esso
- 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
  - se la lunghezza del pacchetto è maggiore di `length`, quando `length` bytes sono stati copiati
  - `getLength` restituisce il numero di bytes effettivamente copiati

OSS: Oggi socket ha associati 2 buffer, uno per la ricezione ed uno per la spedizione, GESTITI DALLA JVM. Sono diversi dai buffer passati nei costruttori.

Per visualizzare la dim. si usa:

```
int r = dgs.getReceiveBufferSize();
int s = dgs.getSendBufferSize();
```

## METODI GET del DATAGRAM PACKET

```
byte[] getData()
```

- restituisce un riferimento all'intero buffer associato più recentemente al `DatagramPacket`

a) Mediante costruzione

b) Mediante `setData()`

```
public int getLength()
```

- restituisce la lunghezza dei dati ricevuti

## METODI SET del DATAGRAM PACKET

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

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

- i dati nel pacchetto da spedire possono essere inseriti mediante il costruttore o mediante il metodo `SetData`

• utile quando si deve mandare una grossa 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);
```

## RICEZIONE DEI PACCHETTI

- `sock.receive(buffer)`

dove `sock` è il socket attraverso il quale ricevo il pacchetto e `buffer` è la struttura in cui memorizzo il pacchetto ricevuto

la ricezione è bloccante cioè il

processo si blocca fino al momento in cui viene ricevuto il pacchetto

Posso impostare un `timeout`: `ds.setSoTimeout(30000)`

che lancia una `InterruptedException`

## INVIO DEI PACCHETTI

```
sock.send(dp)
```

dove: `sock` è il socket attraverso il quale voglio spedire il pacchetto `dp`



la send è non bloccante cioè

il programma continua senza attendere

l'arrivo del pacchetto a destinazione

## DATI SFRUTTATORI IN PACCHETTI UDP

Per inviare e ricevere dati strutturati si usano:

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

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

### ByteArrayOutputStream

Gli oggetti di questa classe rappresentano stream di bytes

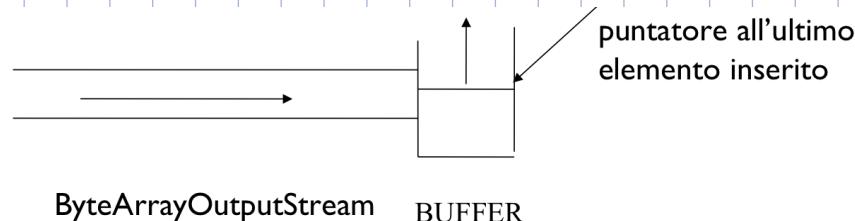
e hanno 2 variabili principali:

`protected byte buf []`

`protected int count`

**Buf:** Buffer di memoria dove vengono riportati i dati scritti sullo stream. Il buffer ha dimensione variabile (default = 32 byte) che raddoppia ogni volta che viene riempito.

**Count:** Indica quanti byte sono memorizzati in `buf`



### OSS:

1) ad un `ByteArrayOutputStream` può essere collegato un altro filtro

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

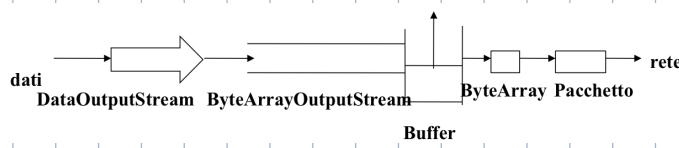
2) i dati presenti nel buffer B associato ad un `ByteArrayOutputStream` baos possono essere copiati in un array di bytes

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

### Metodi:

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

### Flusso dei dati:

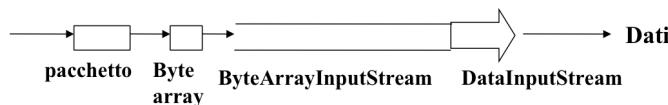


# Byte Array InputStream

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

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

Flusso dei dati:



Riassunto:

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

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

## UDP channels

### Read / Put non bloccanti

```
ByteBuffer buffer = ByteBuffer.allocate(8192);
DatagramChannel channel= DatagramChannel.open();
channel.configureBlocking(false);
channel.socket().bind(new InetSocketAddress(9999));
SocketAddress address = new InetSocketAddress("localhost",7);
buffer.put(...);
buffer.flip();
while (channel.send(buffer, address) == 0);
 //do something useful ...
buffer.clear();
while ((address = channel.receive(buffer))==null);
 //do something useful ...
```

- se la send restituisce 0, ciò indica che il send-buffer è pieno
- se la receive restituisce null, non è stato ricevuto alcun Datagram
- il programma può eseguire altre operazioni, nell'attesa che il canale sia disponibile per l'operazione

### Multiplexing

- possibilità di registrare il canale con un selettore
- invocazione del metodo `Selector.select` per testare la "readability" o "writability" del canale.

| Operation | Meaning                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| OP_READ   | Data is present in the socket receive-buffer or an exception is pending.                                                                                                                                                                                                                                                                                                                                                                                         |
| OP_WRITE  | Space exists in the socket send-buffer or an exception is pending. In UDP, OP_WRITE is almost always ready except for the moments during which space is unavailable in the socket send-buffer. It is best only to register for OP_WRITE once this buffer-full condition has been detected, i.e. when a channel write returns less than the requested write length, and to deregister for OP_WRITE once it has cleared, i.e. a channel write has fully succeeded. |