Laboratorio di Reti

Ahmad Shatti

2021 - 2022

Indice

Chapter 1

Java multithreading

1.1 Introduzione

Il concetto alla base della **programmazione concorrente** è il fatto di gestire più task alla volta, ovvero, il programma principale interrompe quello che stava facendo, gestisce un altro problema e poi riprende l'esecuzione precedente. Quindi permette di partizionare il programma in sequenze di operazioni separate, eseguite in modo indipendente (non sequenziale). La programmazione concorrente possiede diversi vantaggi come miglior utilizzazione delle risorse e miglior performance ma ha anche alcuni problemi:

- più difficile il debugging e la manutenzione del software
- *race conditions* e sincronizzazioni
- deadlock, starvation

1.2 Thread

Con la programmazione concorrente si può suddividere un programma in attività separate che funzionano in modo indipendente. Ciascuno di questi task autonomi è pilotato da un **thread**: flusso di esecuzione all'interno di un processo. Quando si manda in esecuzione un programma Java, la JVM crea un thread che invoca il metodo main del programma. Quindi esiste sempre almeno un thread per ogni programma: il main. In seguito altri thread sono attivati automaticamente da Java e ogni thread durante la sua esecuzione può creare ed attivare altri thread. Per creare ed attivare un thread vi sono due modi:

- **Metodo 1 : `Runnable Interface`**

  1. si definisce un task (codice che il thread deve eseguire) e per farlo si crea una classe che implementa l'interfaccia `Runnable` che contiene solo la segnatura del metodo `run()`
  2. si crea un oggetto thread e gli si passa il task definito
  3. si attiva il thread con `start()`

  La creazione del task non implica la creazione di un thread che lo esegua; lo stesso task può essere eseguito da più threads

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

  Il metodo `start()` segnala allo schedulatore (tramite la JVM) che il thread può essere attivato. L'ambiente del thread viene inizializzato e restituisce immediatamente il controllo al chiamante senza attendere che il thread

attivato inizi la sua esecuzione. Cosa accade se si sostituisce l'invocazione del metodo `start()` con `run()`? Non viene attivato nessun thread: flusso di esecuzione sequenziale

- **Metodo 2: `Thread class`**

  1. si crea una classe che estenda `Thread` e si effettua l'overriding del metodo `run()`
  2. si istanzia un oggetto di quella classe che sarà un thread il cui comportamento è quello definito nel metodo `run`
  3. si invoca il metodo `start()` sull'oggetto istanziato

```
public class Task extends Thread {
    public void run(){
        System.out.println("it's running");
    }
    public static void main(String [] args){
        Task task = new Task();
        task.start();
    }
}
```

lo svantaggio di questa soluzione è che in Java l'ereditarietà è singola: se si estende la classe `Thread`, la classe i cui oggetti devono essere eseguiti come thread non può estendere altre classi

1.2.1 Attivazione di un insieme di threads

Scrivere un programma che stampa le tabelline dall'1 al 10. Si attivino 10 threads; ogni numero n ($1 \leq n \leq 10$) viene passato ad un thread diverso. Il task assegnato consiste nello stampare la tabellina corrispondente al numero che gli è stato passato come parametro

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

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

La stampa del messaggio *"Avviato calcolo tabelline"* precede quelle effettuate dai threads. Questo significa che il controllo è stato restituito al thread chiamante (quello associato al main) prima che sia iniziata l'esecuzione dei threads attivati. L'istruzione `currentThread()` restituisce un riferimento al thread che sta eseguendo il frammento di codice

1.3 Classe Thread

La classe contiene metodi per:

- *costruire* un thread interagendo con il sistema operativo ospite. Costruttori:

  `Thread()`, crea un thread con nome e priorità di default

  `Thread(Runnable target)`, crea un thread da un oggetto Runnable

  `Thread(Runnable target, String name)`, crea un thread da un oggetto Runnable e gli assegna un nome

  `Thread(String name)`, crea un thread assegnandogli un nome

- *attivare, sospendere, interrompere* i thread

  `void start()`, fa partire l'esecuzione di un thread. La macchina virtuale Java invoca il metodo `run()` del thread appena creato

  `void sleep(long M)`, sospende l'esecuzione del thread per M millisecondi

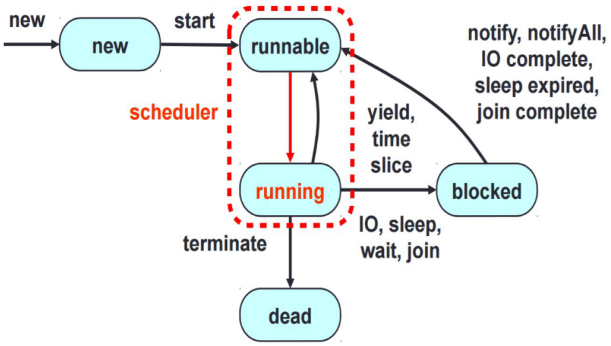
  `void yield()`, può mettere in pausa l'esecuzione del thread invocante per lasciare l'uso della CPU agli altri thread, con la stessa priorità in coda d'attesa

  `void join()`, se viene invocato sull'istanza t di un thread, il thread che lo esegue si sospende in attesa della terminazione di t

1.4 Stati di un thread

Il ciclo di vita di un thread è caratterizzato dai seguenti stati:

- **created/new**, subito dopo l'istruzione `new` le variabili sono state allocate e inizializzate; il thread è in attesa di passare allo stato di eseguibile
- **runnable/running**, il thread è in esecuzione o in coda d'attesa per ottenere l'utilizzo della CPU (le specifiche Java non separano i due stati in realtà, un thread running è nello stato runnable)
- **not runnable/blocked/waiting**, il thread non può essere messo in esecuzione dallo scheduler. Entra in questo stato quando è in attesa di un'operazione di I/O, o dopo l'invocazione di metodi come `sleep()`
- **dead**, termine naturale della sua esecuzione



1.5 Proprietà di un thread

La classe `Thread` salva alcune informazioni che aiutano ad identificare un thread:

- **ID - `getId()`**: identificatore del thread
- **nome - `getName()`**: nome del thread
- **priorità - `getPriority()`**: valore da 1 a 10 (1 priorità più bassa)
- **nome gruppo - `getThreadGroup()`**: gruppo a cui appartiene il thread
- **stato - `getState()`**: uno dei possibili stati: *new, runnable, blocked, waiting, time waiting, terminated*

1.6 Interrompere un thread

Il metodo `interrupt()` imposta a true una variabile booleana flag nel descrittore del thread; il flag vale true se esistono interruzioni pendenti. È possibile testare il valore del flag mediante i metodi `interrupted()` e `isInterrupted()`. Entrambi i metodi restituiscono un valore booleano che segnala se il thread ha ricevuto un'interruzione. La differenza tra i due è che il primo rimette il flag a false, il secondo invece non cambia il valore del flag

```
public class SleepInterrupt implements Runnable{
    public void run(){
        try{
            System.out.println("dormo per 20 secondi");
            Thread.sleep(20000);
            System.out.println("mi sono svegliato");
        }
        catch (InterruptedException e){
            System.out.println("interrotto");
        }
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Thread thread = new Thread(new SleepInterrupt());
        thread.start();
        try{
            Thread.sleep(2000);
        }
        catch (InterruptedException e){}
        System.out.println("interrompo l'altro thread");
        thread.interrupt();
        System.out.println("sto terminando...");
    }
}
```

stampa *"dormo per 20 secondi", "interrompo l'altro thread", "sto terminando...", "interrotto"*

1.7 Thread demoni

I thread demoni hanno il compito di fornire un servizio in background fino a che il programma è in esecuzione; sono thread a bassa priorità. Quando tutti i thread non-demoni sono completati, la JVM termina il programma (anche se ci sono thread demoni in esecuzione) e forza la terminazione dei thread demoni.

```
public class simpleDemon implements Runnable{
    public void run(){
        try{
            Thread.sleep(100);
            System.out.println("termino");
        }
        catch (InterruptedException e){}
    }
    public static void main(String[] args) throws InterruptedException {
        for(int i = 0; i<10; i++){
            Thread thread = new Thread(new simpleDemon());
            thread.setDaemon(true);
            thread.start();
        }
    }
}
```

Non viene stampato alcun messaggio "*termino*" perché quando il thread non demone `main` termina, la JVM forza la terminazione dei thread demoni. Per fare in modo che un thread demone non venga ucciso immediatamente si può

utilizzare il metodo `join()`: il programma principale si sospende in attesa che il thread attivato termini (anche se è demone).

1.8 Terminazione di programmi concorrenti

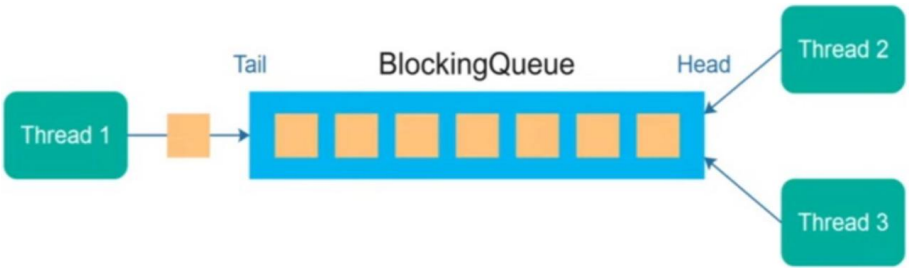
Un programma Java termina quando tutti i thread non-demoni terminano. Se il thread iniziale, cioè quello che esegue il metodo `main()` termina, i restanti thread ancora attivi e non demoni continuano la loro esecuzione. Se uno dei thread usa l'istruzione `System.exit()` per terminare l'esecuzione, allora tutti i threads terminano la loro esecuzione

Chapter 2

BlockingQueue, Thread Pool e Callable

2.1 BlockingQueue

L'interfaccia `BlockingQueue` estende `Queue` (inserimento in coda ed estrazione in testa) e permette ai thread che inseriscono ed eliminano elementi dalla coda di bloccarsi: in figura Thread-1 si blocca se la coda è piena, Thread-2 e Thread-3 se è vuota. Implementa una corretta sincronizzazione tra i thread



`BlockingQueue` offre 4 categorie di metodi differenti, rispettivamente per inserire, rimuovere, esaminare un elemento dalla coda; ogni metodo ha un comportamento diverso relativamente al caso in cui l'operazione non possa essere svolta

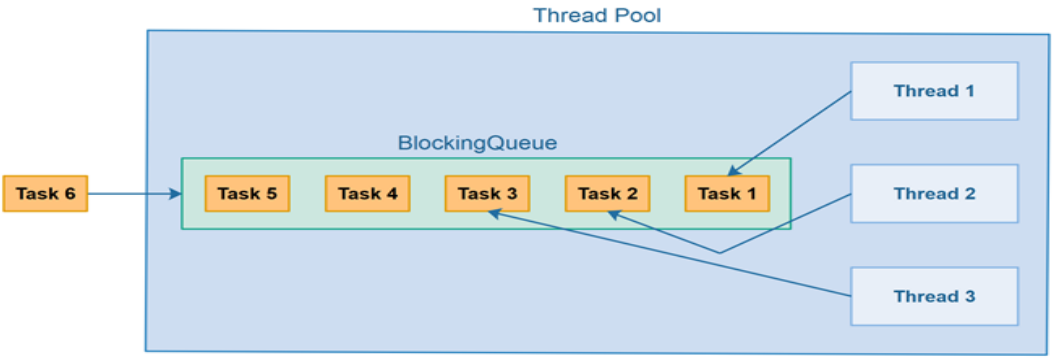
	Throw Exception	Special Value	Blocks	Times Out
insert	`add(o)`	`offer(o)`	`put(o)`	`offer(o, timeout, timeunit)`
remove	`remove(o)`	`poll()`	`take()`	`poll(timeout, timeunit)`
examine	`element()`	`peek()`		

Le operazioni `add, remove, element` lanciano una eccezione se si tenta di aggiungere un elemento ad una coda piena o rimuovere un elemento da una coda vuota. Le operazioni `offer, pool, peek` ritornano rispettivamente false, null, null se non possono portare a termine l'operazione. Infine `put, take` hanno un comportamento bloccante se non si può portare a termina l'operazione. Le implementazioni di `BlockingQueue` sono *thread-safe* e quelle maggiormente usate sono:

- **`ArrayBlockingQueue`**, è una coda di dimensione limitata che memorizza gli elementi all'interno di un array. Non sono possibili inserzioni/rimozioni in parallelo (una sola *lock* per tutta la struttura)
- **`LinkedBlockingQueue`**, può essere limitata o illimitata; mantiene gli elementi in una `LinkedList` quindi vi è maggiore occupazione di memoria ma è possibile avere inserzioni ed estrazioni concorrenti (*lock* separate per lettura e scrittura)
- **`SynchronousQueue`** è una `BlockingQueue` di dimensione 1. L'operazione di inserzione deve attendere per una corrispondente rimozione e viceversa (è un po' fuorviante chiamarla coda)

2.2 Thread Pool

La creazione di un thread è costosa: richiede tempo e attività di elaborazione da parte della JVM e del sistema operativo. Creare un nuovo thread per ogni task risulta una soluzione improponibile, specialmente nel caso di task "leggeri" molto frequenti. L'alternativa è creare un *pool di thread* dove ogni thread può essere usato per più task. L'obiettivo è diminuire il costo per l'attivazione/terminazione dei threads; riusare lo stesso thread per l'esecuzione di più tasks e eventualmente controllare il numero massimo di thread che possono essere eseguiti concorrentemente



Thread Pool è una struttura dati la cui dimensione massima può essere prefissata e contiene riferimenti ad un insieme di threads. Permette di gestire l'esecuzione di task senza dover gestire esplicitamente il ciclo di vita dei thread. I thread del pool possono essere riutilizzati per l'esecuzione di più tasks. La sottomissione di un task al pool viene disaccoppiata dall'esecuzione da parte del thread. L'esecuzione del task può essere ritardata se non vi sono risorse disponibili. Il progettista crea il pool e stabilisce una politica per la gestione dei thread del pool: quando i thread vengono attivati (al momento della creazione del pool, on demand, all'arrivo di un nuovo task, ...) e quando è opportuno terminare l'esecuzione di un thread. Il supporto al momento della sottomissione del task può:

- utilizzare un thread attivato in precedenza, inattivo in quel momento
- creare un nuovo thread
- memorizzare il task in una struttura dati (coda) in attesa dell'esecuzione
- respingere la richiesta di esecuzione del task

Il numero di thread attivi nel pool può variare dinamicamente. Interfacce che definiscono servizi generici di esecuzione sono `Executor` e `ExecutorService`. Il primo esegue un task `Runnable`; mentre il secondo estende il primo con metodi che permettono di gestire il ciclo di vita del pool, come ad esempio la terminazione. I task devono essere incapsulati in oggetti di tipo `Runnable` e passati a questi esecutori, mediante l'invocazione del metodo `execute()`. La classe `Executors` opera cone una *Factory* in grado di generare oggetti di tipo `ExecutorService` con comportamenti predefiniti

- **FixedThreadPool**, crea un pool con il seguente comportamento:
  vengono creati n thread, con n fissato al momento della inizializzazione del pool. Quando viene sottomesso un task t, se tutti i threads sono occupati nell'esecuzione di altri tasks, t viene inserito in una coda gestita automaticamente dall'`ExecutorService`. Se almeno un thread è inattivo, viene utilizzato quel thread. Utilizza una coda illimitata: `LinkedBlockingQueue`

```
public class Task implements Runnable {
    public int id;
    public Task(int id){this. id = id;}
    public void run(){
        try{
            long time = (long)(Math.random()*10);
            System.out.printf("%s: starting task-%d during %d seconds\n",
                              Thread.currentThread().getName(), id, time);
            Thread.sleep(time);
        }
        catch(InterruptedException e){}
        System.out.printf("%s: task-%d finished\n",
                          Thread.currentThread().getName(), id);
    }
}
```

```
public class Test {
    public static void main(String [] args){
        ExecutorService pool = Executors.newFixedThreadPool(10);
        for(int i = 0; i<100; i++){
            pool.execute(new Task(i));
        }
        pool.shutdown();
    }
}
```

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

- **`CachedThreadPool`**, crea un pool con il seguente comportamento:
se tutti i thread del pool sono occupati nell'esecuzione di altri task e c'è un nuovo task da eseguire, viene creato un nuovo thread. Quindi non vi è nessun limite sulla dimensione del pool. Se disponibile viene riutilizzato un thread che ha terminato l'esecuzione di un task precedente. Se un thread rimane inutilizzato per 60 secondi, la sua esecuzione termina. Non utilizza nessuna coda: `SynchronousQueue`

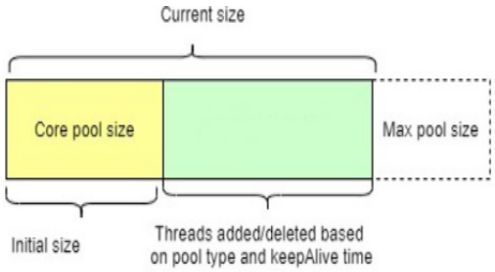
```
public class Test {
    public static void main(String [] args){
        ExecutorService pool = Executors.newCachedThreadPool();
        for(int i = 0; i<100; i++){
            pool.execute(new Task(i));
            // attivato un nuovo thread per ogni task
        }
        pool.shutdown();
    }
}
```

cosa accade se si distanzia la sottomissione dei task ai thread, inserendo una sleep nel for dopo la `execute()`?
Viene utilizzato sempre il Thread-1 per tutti i task

2.2.1 Thread Pool Executor

Il costruttore più generale della classe `ThreadPoolExecutor` permette di personalizzare la gestione del pool

- `CorePoolSize`, dimensione minima del pool, definisce il *core* del pool. I thread del core possono essere attivati al momento della creazione del pool o al momento della sottomissione di un nuovo task. Quando tutti i thread sono stati creati, la politica cambia
- `MaxPoolSize`, dimensione massima del pool; non vi possono essere più di `MaxPoolSize` threads nel pool
- `keepAliveTime`, per i thread non appartenenti al core; se nessun task viene sottomesso entro un tempo T, il thread termina la sua esecuzione
- `workqueue` è una struttura dati necessaria per memorizzare gli eventuali tasks in attesa di esecuzione



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

- se un thread del core è inattivo, il task viene assegnato ad esso
- se tutti i thread del core stanno eseguendo un task e la coda non è piena, il nuovo task viene inserito nella coda. I task verranno poi prelevati dalla coda ed inviati ai thread disponibili.
- se tutti i thread del core stanno eseguendo un task e la coda è piena si crea un nuovo thread attivando così k thread (`corePoolSize` $\leq k \leq$ `MaxPoolSize`). Se la coda è piena e sono attivi `MaxPoolSize` threads il task viene respinto

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

public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS)

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

2.2.2 Rifiutare un task

Per rifiutare un task è possibile scegliere esplicitamente una "*rejection policy*" al momento della creazione del task. `AbortPolicy` è la politica di default e consiste nel sollevare `RejectedExecutionException`. Altre politiche predefinite sono `DiscardPolicy, DiscardOldestPolicy, CallerRunsPolicy`... Per definire una custom *rejection handler* si implementa l'interfaccia `RejectExecutionHandler` ed il metodo `rejectedExecution`

```
public class Test {
    public static void main(String [] args){
        ExecutorService pool = new ThreadPoolExecutor(10, 12, 120,
                               timeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(3));
        for(int i = 0; i<100; i++){
            try{
                pool.execute(new Task(i));
            }
            catch(RejectedExecutionException e){
                System.out.println("task rejected " + e.getMessage());
            }
        }
        pool.shutdown();
    }
}
```

2.2.3 Terminazione

La JVM termina la sua esecuzione quando tutti i thread non-demoni terminano la loro esecuzione. È necessario analizzare il concetto di terminazione nel caso di `Executor Service` poiché i task vengono eseguiti in modo asincrono rispetto alla loro sottomissione. In un certo istante, alcuni task sottomessi precedentemente possono essere completati, alcuni in esecuzioni e alcuni in coda. Poiché alcuni thread possono essere sempre attivi, Java mette a disposizione dell'utente alcuni metodi che permettono di terminare l'esecuzione del pool. La terminazione può avvenire:

- **in modo graduale**: "finisci ciò che hai iniziato, ma non iniziare nuovi tasks". L'istruzione `shutdown()` inizia la terminazione e nessun task viene accettato dopo che è stata invocata. Tutti i task sottomessi in precedenza e non ancora terminati vengono eseguiti, compresi quelli accodati la cui esecuzione non è ancora iniziata
- **in modo istantaneo**: "stacca la spina immediatamente". L'istruzione `shutdownNow()` non accetta ulteriori tasks, elimina i tasks non ancora iniziati e restituisce una lista dei tasks che sono stati eliminati dalla coda. L'implementazione best effort tenta di terminare l'esecuzione del thread che stanno eseguendo i tasks, inviando una interruzione al thread in esecuzione nel pool. Tuttavia non garantisce la terminazione immediata dei threads del pool: se un thread non risponde all'interruzione, non termina

Per capire se l'esecuzione del pool è terminata:

- **attesa passiva**: si invoca la `awaitTermination(long timeout, TimeUnit unit)` si blocca finché tutti i task hanno completato l'esecuzione o scade il timeout
- **attesa attiva**: si invoca ripetutamente la `isTerminated()` che restituisce true se tutti i task sono completati, compresi quelli in coda

2.3 Thread che restituiscono risultati: Callable e Future

Un oggetto di tipo `Runnable` incapsula un'attività che viene eseguita in modo asincrono. Una `Runnable` si può considerare un metodo asincrono, senza parametri e che non restituisce un valore di ritorno. Per definire un task che restituisca un valore di ritorno occorre utilizzare le seguenti interfacce:

- `Callable`, per definire un task che può restituire un risultato e sollevare eccezioni. Contiene solo il metodo `call()` (analogo al metodo `run()` di `Runnable`). Il parametro di tipo `<V>` indica il tipo del valore restituito

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

- `Future` per rappresentare il risultato di una computazione asincrona. Definisce metodi per controllare se la computazione è terminata, per attendere la terminazione di una elaborazione e per cancellare una elaborazione. Sottomette direttamente l'oggetto di tipo `Callable` al pool mediante il metodo `submit()`; la sottomissione restituisce un oggetto di tipo `Future`. Ogni oggetto `Future` è associato ad uno dei task sottomessi al Thread Pool. Metodi:

  `V get()`, si blocca fino a che il thread non ha prodotto il valore richiesto e restituisce il valore calcolato

  `V get(long timeout, TimeUnit)`, definisce un tempo massimo di attesa della terminazione del task, dopo cui viene sollevata una `TimeoutException`

  `boolean cancel(boolean mayInterruptIfRunning)`, tenta di annullare la cancellazione del task

  `boolean isCancelled()`, restituisce true se il task è stato cancellato prima di essere completato

  `boolean isDone()`, restituisce true se il task è stato completato

Chapter 3

Sincronizzazione di threads

3.1 Condividere risorse tra threads

Lo scenario tipico di un programma concorrente è avere un insieme di threads che condividono una risorsa. Quindi più threads accedono concorrentemente allo stesso file, alla stessa parte di un database o di struttura di memoria. L'accesso non controllato a risorse condivise può provocare situazioni di errore ed inconsistenze: **race conditions**. I dati su cui a luogo la race condition diventano *inconsistenti*, cioè assumono uno stato non deterministico. Inoltre l'inconsistenza si verifica solo alcune volte, il comportamento è dipendente dal tempo, quindi non è facile verificare la correttezza di tali programmi. La **sezioni critica** è un blocco di codice in cui si effettua l'accesso ad una risorsa condivisa e che deve essere eseguito da un thread per volta. Java offre diversi meccanismi per la sincronizzazioni di thread per l'implementazione di sezioni critiche. Meccanismi a basso livello:

- interfaccia **`Lock`**
- **variabili di condizione** associate a `lock()`

Meccanismi ad alto livello:

- **`synchronized`** keyword
- `wait(), notify(), notifyAll()`
- concetto di **monitor**

Una classe si definisce **thread-safe** se l'esecuzione concorrente dei metodi definiti nella classe non provoca comportamenti scorrenti, ad esempio race conditions

3.2 Lock esplicite

In Java una **lock** è un oggetto che può trovarsi in due stati diversi: "*locked*" e "*unlocked*". Lo stato viene impostato con i metodi `lock()` ed `unlock()`. Un solo thread alla volta può impostare lo stato a *locked*, cioè ottenere la `lock()`. Gli altri thread che tentano di ottenere la lock si bloccano. La gestione dei thread bloccati dipende dalla politica di fairness:

- **fair lock**, thread bloccati serviti secondo una politica FIFO
- **unfair lock**, il thread bloccato aspetta il suo turno in un ciclo, controllando ripetutamente che gli venga dato il lock. No garanzia che ci sia un ordinamento FIFO

Quindi le lock vengono usate per definire le sezioni critiche, cioè assicurano che solo un thread per volta possa entrare in una sezione critica e permettono di evitare la race conditions.

```
Lock l = ...;
l.lock();
try{
    // accesso a questa risorsa e' protetto dalla lock
} finally {
    l.unlock();
}
```

Notare l'uso del blocco `finally()`: garantisce che la lock venga rilasciata anche nel caso venga sollevata una eccezione. Attenzione ai **deadlocks**: Thread-A acquisce la lock(X) e Thread-B acquisisce la lock(Y); se Thread-A tenta di acquisire la lock(Y) e simultaneamente Thread-B tenta di acquisire la lock(X) allora entrambi i thread sono bloccati all'infinito, in attesa della lock detenuta dall'altro thread. L'interfaccia `Lock` viene implementata da `ReentrantLock`, `ReentrantReadWriteLock.ReadLock` e `ReentrantReadWriteLock.WriteLock`

```
private ReentrantLock l = new ReentrantLock();
private void accessResource(){
    l.lock();
    try{
        // aggiorna la risorsa
        if(condition()) accessResource();
    }
    finally{
        l.unlock();
    }
}
```

In questo programma il thread potrebbe entrare in deadlock con se stesso. Per evitare queste situazioni `ReentrantLock` utilizza un contatore: un thread può acquisire più volte la lock su uno stesso oggetto senza bloccarsi. Il contatore viene incrementato ogni volta che un thread acquisisce la lock e decrementato ogni volta che un thread rilascia la lock. La lock viene definitivamente rilasciata quando il contatore diventa 0. Per questo motivo si chiama `ReentrantLock` perché il codice "rientra" nel blocco, cercando di riacquisire la lock. Metodi dell'interfaccia `Lock`:

`void lock()`, acquisisce la lock

`void unlock()`, rilascia la lock

`void lockInterruptibly()`, se un thread è bloccato in attesa di una lock, non è possibile interagirci in alcun modo, solo se acquisirà la lock sarà possibile inviargli una interruzione. Questo metodo consente di "rispondere" ad una interruzione, mentre si è in attesa di una lock. Solleva una `InterruptedException` quando un altro metodo invoca il metodo `interrupt`

`boolean tryLock()`, tenta di acquisire la lock; se è già posseduta da un altro thread, il metodo termina immediatamente e restituisce il controllo al chiamante. Restituisce vero se è riuscito ad acquisire la lock, falso altrimenti

`boolean tryLock(long time, TimeUnit unit)`

Scenario: esiste una applicazione che legge e scrive una risorsa. La scrittura è meno frequente delle operazioni di lettura. Due thread che leggono la stessa risorsa non causano problemi l'uno all'altro, invece se un singolo thread desidera scrivere sulla risorsa, non devono essere in corso altre operazioni di lettura o scrittura. La `ReentrantLock` garantisce la *mutua esclusione* ma non è una soluzione efficiente. L'interfaccia `ReadWriteLock` risolve il problema: mantiene una coppia di lock associate, una per le operazioni di lettura e una per le scritture

- *lock di lettura*: può essere acquisita da più thread lettori purché non vi sia uno scrittore che ha acquisito la lock
- *lock di scrittura*: è esclusiva

```
public class Teatro {
    private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    private Lock readLock = lock.readLock();
    private Lock writeLock = lock.writeLock();
    String [][] theatreMap = new String[4][6]; // posti disponibili teatro

    public Teatro(){ // inizialmente tutti posti vuoti
        for(int i = 0; i<4; i++) {
            for(int j = 0; j<6; j++) theatreMap[i][j] = "";
        }
    }
    public void readMap(){
        readLock.lock();
```

```
        // stampa la mappa
        readLock.unlock();
    }
    public void book(int x, int y){
        writeLock.lock();
        theatreMap[x][y] = "X"; // occopi il posto
        writeLock.unlock();
    }
}
```

Le lock introducono una performance penalty dovuta a più fattori: contention, bookkeeping, scheduling, blocking e unblocking. Inoltre introduce overhead, per cui vanno usate con oculatezza.

3.2.1 Variabili condizioni

Ad una lock possono essere associate un insieme di **variabili condizioni**. Lo scopo è quello di permettere ai thread di controllare se una condizione sullo stato della risorsa è verificata o meno

- se la condizione è falsa, si sospendono rilasciando la lock ed inseriscono il thread in una coda in attesa per quella condizione
- se la condizione è verificata risvegliano un thread in attesa per quella condizione

Quindi la JVM mantiene più code: una per i thread in attesa di acquisire la lock e una per ogni variabile di condizione. Solo dopo aver acquisito la lock su un oggetto è possibile sospendersi su una variabile di condizione, altrimenti viene generata una `IllegalMonitorException`. L'interfaccia **`Condition`** definisce i metodi per sospendere un thread e per risvegliarlo:

`void await()` fa in modo che il thread corrente attende fino a che non viene risvegliato o interrotto

`void await(long time, TimeUnit unit)`

`void awaitUninterruptibly()` fa in modo che il thread corrente attende fino a che non viene risvegliato

`void signal()`, risveglia un thread che era in attesa

`void signalAll()`, risveglia tutti i thread che erano in attesa

Tra il momento in cui ad un thread arriva una notifica ed il momento in cui riacquisisce la lock, la condizione può diventare di nuovo falsa. È meglio sempre ricontrollare sempre la condizione dopo aver acquisito la lock, inserendo la `await()` in un ciclo while

3.3 Lock implicite

Oltre alle lock esplicite (come le `ReentrantLock`) vi sono dei meccanismi a più alto livello. Java associa a ciascun oggetto una **lock implicita** ed una coda associata a tale lock. Per acquisire una lock implicita vi sono due metodi:

- **metodi synchronized**, quando il metodo synchronized viene invocato il metodo tenta di acquisire la lock implicita associata all'oggetto su cui esso è invocato. Se l'oggetto è bloccato (lock acquisita da un altro thread in un blocco sincronizzato o in un altro metododo synchronized) il thread viene sospeso nella coda associata all'oggetto fino a che il thread che detiene la lock la rilascia. La lock viene rilasciata al ritorno del metodo. La lock è associata ad una istanza dell'oggetto, non alla classe.

```
public synchronized void deposito(float soldi){
    saldo = saldo + soldi
}
```

- **blocchi di codice sincronizzato**, un thread acquisisce la lock sull'oggetto riferito da `mutex` quando entra nel blocco sincronizzato e la rilascia quando lascia il blocco.

```
Object mutex = new Object();
public void method(){
    nonCriticalSection();

    synchronized(mutex){
        criticalSecion()
    }

    nonCriticalSection();
}
```

Si possono sincronizzare anche metodi statici: acquisiscono la lock intrinseca associata alla classe, invece che all'oggetto. Regole:

- i costruttori non devono essere dichiarati synchronized: il compilatore solleva una eccezione.
- la keyword synchronized è riferito all'implementazione non alla signature del metodo, quindi non ha senso specificare synchronized delle interfacce
- se il metodo della superclasse non è synchronized, una sottoclasse che fa l'override di quel metodo può renderlo synchronized
- se il metodo della superclasse è synchronized e una sottoclasse vuole fare l'override di quel metodo allora lo deve specificare come synchronized. Nel caso non volesse fare l'override la sottoclasse lo eredita come synchronized

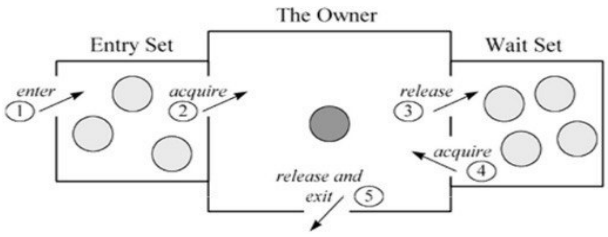
3.4 Monitor

Utilizzando il modificatori synchronized o i blocchi synchronized si può implementare un meccanismo linguistico ad alto livello per la sincronizzazione chiamato **monitor**. Serve per incapsulare un oggetto condiviso e le operazioni su di esso. Quindi le funzionalità offerte dal monitor sono:

- **mutua esclusione** sulla struttura: lock implicita gestite dalla JVM, un solo thread per volta accede all'oggetto condiviso
- **coordinazione tra i thread**: meccanismi per la sospensione/risveglio sullo stato dell'oggetto condiviso simili a variabili di condizione: `wait/notify`

In concreto significa che vi è un oggetto che offre un insieme di metodi synchronized che incapsula lo stato di una risorsa condivisa. Associato a questo meccanismo vi sono due code:

- **Entry set**, contiene i threads in attesa di acquisire la lock
- **Wait set**, contiene i threads che hanno eseguito una `wait` e sono in attesa di una `notify`: inserzione/estrazione in questa coda in seguito ad invocazione esplicita di `wait()` e `notify()`



Per invocare i seguenti metodi occorre aver acquisito precedentemente la lock sull'oggetto su cui sono invocati: devono essere quindi invocati all'interno di un metodo o di un blocco sincronizzato altrimenti viene sollevata l'eccezione `IllegalMonitorException()`:

`void wait()`, sospende il thread in attesa che sia verificata una condizione. Rilascia la lock prima di sospendere il thread

`void wait(long timeout)`, sospende per al massimo timeout millisecondi

`void notify()`, risveglia uno dei threads nella coda di attesa di quell'oggetto. Se viene invocato quando non vi è alcun thread sospeso la notifica viene "persa"

`notifyAll()`, risveglia tutti i threads in attesa e i thread risvegliati competono per l'acquisizione della lock

Come nelle variabili condizioni, anche qui conviene testare la condizione all'interno di un ciclo: la condizione su cui il thread t è in attesa si è verificata però un altro thread l'ha resa di nuovo non valida dopo che t è stato risvegliato

```
synchronized (obj) {              // acquisisce il monitor
    while(!cond) {
        obj.wait();               // rilascia il monitor
    }                             // acquisisce il monitor
                                  // modifica obj o altri dati
    obj.notifyAll();
}                                 // rilascia il monitor
```

Vantaggi lock implicite

- imposta una disciplina di programmazione per evitare errori dovuti alla complessità del programma concorrente: deadlocks, mancato rilascio di lock,...
- definisce costrutti strutturati per la gestione della concorrenza
- maggior robustezza
- svantaggio: minore flessibilità rispetto a lock esplicite

Vantaggi lock esplicite

- numero maggiore di funzione disponibili, maggiore flessibilità
- `tryLock()` consente di non sospendere il thread se un altro thread è in possesso della lock, restituendo un valore booleano
- presenza di variabili di condizioni
- miglior performance

3.5 Variabili atomiche e volatili

Le **variabile atomiche** incapsulano variabili di tipo primitivo e garantiscono l'atomicità delle operazioni senza usare sincronizzazioni esplicite o lock

```
AtomicInteger value = new AtomicInteger(1); // gli si assegna il valore 1
value.incrementAndGet(); // incrementa atomicamente di uno
value.decrementAndGet(); // decrementa atomicamente di uno
```

Ciascun thread può copiare le variabili dalla memoria principale in una cache della CPU per motivi di prestazioni. Quindi può capitare che gli aggiornamenti di un thread non sono visibili agli altri thread: vi è un *problema di visibilità*.

```
public class Test extends Threads {
    boolean keepRunning = true;
    public void run(){
        while(keepRunning){}
        System.out.println("Thread terminated");
    }
    public static void main(String [] args) throws InterruptedException {
        Test t = new Test();
        t.start();
        Thread.sleep(1000);
        t.keepRunning = false;
        System.out.println("keepRunning set to false");
    }
}
```

Questo codice non termina: quando il Thread-1 aggiorna il flag `keepRunning` la modifica può non essere letta dal Thread-2. Il problema può essere risolto modificando la dichiarazione della variabile

```
volatile boolean keepRunning = true;
```

Con la keyword **`volatile`** l'aggiornamento ad una variabile `volatile` è sempre effettuato nella main memory. Quindi assicura che i thread vedano sempre il valore più recente.

3.6 Synchronized collections

Si possono distinguere diversi tipi di collezioni riguardo alla thread safeness:

- **collezioni thread safe, sincronizzate automaticamente da Java** (`Vector, Hashtable`)
- **collezioni che non offrono alcun supporto per il multithreading** (`Map, LinkedList, ArrayList`)
- **synchronized collections**
- **concurrent collections**

Il metodo `add` non è atomica: determina quanti elementi ci sono nella lista; determina il punto esatto del nuovo elemento e incrementa il numero di elementi della lista. Analogamente per la `remove`. I metodi definiti nella interfaccia `Collections` trasformano una `Collection` non thread-safe in una thread-safe

```
List <String> synchList = Collections.synchronizedList(new ArrayList<String>());
synchList.add("hello world");
```

La thread safety garantisce che le invocazioni delle singole operazioni della collezione siano thread-safe. Se si vogliono definire funzioni che coinvolgono più di una operazione base?

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

I metodi `isEmpty()` e `remove()` sono entrambe operazioni atomiche, ma la loro combinazione non lo è. Per esempio vi è una lista con un solo elemento; il primo thread verifica che la lista non è vuota e viene deschedulato prima di rimuovere l'elemento; un secondo thread rimuove l'elemento. Il primo thread torna in esecuzione e prova a rimuovere un elemento non esistente. Java `Synchronized Collections` si dicono **conditionally thread-safe**: le operazioni individuali sulla collezione sono safe, ma funzioni composte da più di una operazione singola possono richiedere meccanismi esterni di sincronizzazione. Per rendere atomica una operazione composta da più di una operazione individuale si possono utilizzare i blocchi sincronizzati

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

3.7 Concurrent collections

Le collezioni sincronizzate garantiscono la thread-safety a discapito della scalabilità del programma. Il livello di sincronizzazione di una collezione sincronizzata può essere troppo stringente. Se una HashTable ha diversi buckets, perché sincronizzare l'intera struttura se si vuole accedere ad un solo bucket? L'idea è accettare un compromesso sulla semantica delle operazioni per mantenere un buon livello di concorrenza ed una buona scalabilità. Quindi le `Concurrent Collections` superano l'approccio di sincronizzare l'intera struttura dati tipico delle collezioni sincronizzate garantendo quindi un supporto più sofistico per la sincronizzazione.

```
ConcurrentHashMap <String, String> concurrentMap
                                = new ConcurrentHashMap <String, String>();
```

Vantaggi

- ottimizzazione degli accessi, non una singola lock
- overlapping di operazioni di scrittura su parti diverse della struttura
- maggior livello di concorrenza, quindi miglioramento di performance e scalabilità

Svantaggi

- approssimazione della semantica di alcune operazioni come `size(), isEmpty` perché restituiscono un valore approssimato
- non ha senso utilizzare synchronized per bloccare l'accesso all'intera struttura

Dato che non si può utilizzare synchronized, come si possono rendere atomiche operazioni composte? `ConcurrentHashMap` offre alcune nuove operazioni eseguite atomicamente come:

`boolean remove(K key, V value)`, rimuove la entry per la chiave key ma solo se è mappata con value

`boolean replace(K key, V oldValue, V newValue)`, sostituisci la entry per la chiave key solo se è mappata con value

Con la `ConcurrentHashMap` la struttura può essere modificata mentre viene scorsa mediante un iteratore (non è necessario effettuare la lock sull'intera struttura) e non viene sollevata una `ConcurrentModificationException`. Tuttavia le modifiche concorrenti possono non risultare visibili all'iteratore che scorre la struttura cioè un iteratore può o non può catturare le modifiche effettuate sulla collezione dopo la creazione di tale iteratore: *weakly consistent iterators*. Un iteratore con questo comportamento si chiama **fail-safe iterator**. Altre collezioni offrono iteratori **fail-fast** che lanciano una `ConcurrentModificationException` se un thread modifica una Collection

Chapter 4

Java.IO

4.1 Introduzione

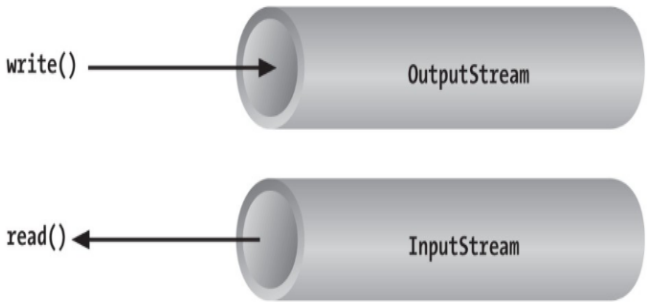
Quando si parla di input/output si parla di programmi che recuperano informazioni da una sorgente esterna o la inviano ad una sorgente esterna (*keyboard*, *files*, *directories*, ...). Vi sono diversi tipi di device di input/output: se il linguaggio dovesse gestire ogni tipo di device come caso speciale, la complessità sarebbe enorme. Quindi vi è la necessità di definire un insieme di astrazioni per la gestione dell'I/O. In Java la principale astrazione è basata sul concetto di **stream**. Classificazione I/O packages:

- **Java IO API**, package standard I/O basato sugli stream e, lavora su bytes e caratteri
- **Java NIO API**, funzionalità simili a Java IO, ma con comportamento non bloccante: migliori performance in alcuni scenari
- **Java NIO.2**, alcuni miglioramenti rispetto a Java NIO

4.2 Stream-based IO

Uno stream è un flusso di informazioni di lunghezza indefinita e rappresenta una connessione tra un programma Java ed un dispositivo esterno (*file*, *connessione di rete*, ...). Può essere descritta come un "tubo" tra una sorgente ed una destinazione: da un estremo entrano i dati, dall'altro escono. L'applicazione inserisce i dati o li legge ad/da un capo dello stream e i dati fluiscono da/verso la destinazione. Caratteristiche generali:

- accesso sequenziale
- mantengono l'ordinamento FIFO
- *one way*: read only oppure write only. Se un programma ha bisogno di dati input ed output, è necessario aprire due stream in input ed in output
- quando un applicazione legge un dato dallo stream (o lo scrive) si blocca finché l'operazione non è completata
- non è richiesta una corrispondenza stretta tra letture/scritture. Ad esempio una unica scrittura inietta 100 bytes sullo stream, che vengono letti con due write successive: la prima legge 20 bytes, la seconda 80 bytes



Le operazioni sugli stream I/O si eseguono in tre passi:

1. si apre l'input/output associato con la sorgente/destinazione costruendo l'istanza appropriata di I/O stream
2. si legge l'input stream già aperto fino a incontrare la fine dello stream; o si scrive sull'output stream già aperto e eventualmente svuotare (*flush*) l'output del buffer
3. si chiude l'input/output stream con il metodo `close()`

Vi sono due tipi principali di stream:

- **stream di bytes** servono a leggere/scrivere sequenze di byte: ideale per leggere/scrivere file binari come un'immagine o la codifica di un video
- **stream di caratteri**, servono a leggere/scrivere sequenze di caratteri UNICODE a 16 bit. L'I/O basato su questo stream traduce automaticamente questo formato interno da e verso il set di caratteri locali

4.3 File class

La classe `File` è una rappresentazione astratta e incapsula delle informazioni come il percorso, il nome e altre metainformazioni dei file. Metodi utili:

`boolean exist()`, verifica se esiste il file

`boolean isDirectory()`, verifica se il file è una directory

`String [] list()`, restituisce i nomi di tutti i file nella directory specificata sotto forma di un array di stringhe

`File [] listFiles()`, restituisce gli oggetti dei file nella directory specificata, sotto forma di un array di tipo `File`.

`long length()` restituisce la lunghezza del file in byte

`String getName()` e `String getPath()` restituiscono rispettivamente il nome del file e il percorso

```
public class ListFiles {
    public static void main(String [] args){
        File dir = new File("."); // directory corrente
        if(dir.isDirectory(){
            String [] files = dir.list();
            for(String file : files){
                if(file.endWith(".java")
                    System.out.println(file);
            }
        }
    }
}
```

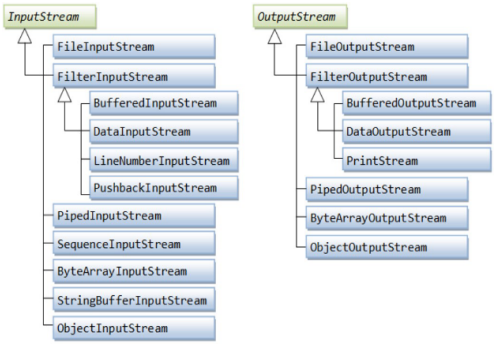
4.4 Stream di bytes

`InputStream` e **`OutputStream`** sono le due principali classi astratte per lo stream di bytes per file binari. Alcuni metodi `InputStream`:

`int read()`, legge un byte dallo stream come un intero unsigned tra 0 e 255; restituisce -1 se viene individuata la fine dello stream. Solleva una `IOException` se c'è un errore di I/O. È bloccante

`int read(byte [] bytes, int offset, int lenght)`, legge `lenght` bytes e li memorizza nell'array `bytes`, iniziando da `offset`. Restituisce il numero di byte letti

`int read(byte [] bytes)`, riempe 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. È equivalente a `read(bytes, 0, bytes.lenght)`



Alcuni metodi `OutputStream` :

`void write(int b)`, scrive un byte specifico nell'output stream

`void write(byte [] b, int off, int len)`, scrive `len` bytes dall'array di byte `b` partendo da offset `off` nell'output stream

`void write(byte [] b)`, scrive `b.lenght` byte nell'array di byte. È equivalente a `write(b, 0, b.lenght)`

`void flush()`, quando si scrivono dei dati, essi non vengono scritti immediatamente ma vengono bufferizzati. L'istruzione `flush()` svuota il buffer e forza la scrittura. Viene utilizzato per controllare che tutte le scritture siano completate prima di chiudere lo stream

Con il metodo `close()` si chiude l'input o output stream e si rilasciano tutte le risorse di sistema associate al flusso. Alcune implementazioni delle classi astratte:

- `FileInputStream` per leggere da un file
- `FileOutputStream` per scrivere in un file. Non utilizza nessun buffer quindi `flush()` non serve

```
InputStream in = null;
OutputStream out = null;
String nomeFile = "foto.jpg";
File f = new File(nomeFile);
try{
    in = new FileInputStream(nomeFile);
    out = new FileOutputStream("nuova-foto.jpg");
    byte [] b = new byte[(int) f.length()];
    in.read(b);
    out.write(b, 0, b.length);
} catch (FileNotFoundException e){
    System.err.println("Non ho trovato il file");
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if(in!=null)
        try {
            in.close();
        } catch (Exception e){
            e.printStackTrace();
        }
    if(out!=null)
        try{
            out.close();
        } catch (Exception e){
            e.printStackTrace();
        }
}
```

I flussi di input e output forniscono metodi di lettura/scrittura che possono essere utilizzati solo per leggere/scrivere bytes. Se si vogliono leggere/scrivere interi, float o stringhe si devono utilizzare le classi filtro: `FilterInputStream` e `FilterOutputStream`. Essi permettono di aggiungere funzionalità agli stream di input e output. Alcune sottoclassi:

- `BufferedInputStream` e `BufferedOutputStream`, implementano una bufferizzazione per stream di input e output. I dati vengono scritti e letti in blocchi di bytes, invece che un solo blocco per volta. Quando si crea un `BufferedInputStream` viene creato un buffer interno e via via che i byte sono letti, il buffer interno viene riempito. Una `read()` su un `BufferedInputStream` è quindi una lettura del buffer interno. Vi è un miglioramento significativo delle performance

```
BufferedInputStream in = null;
BufferedOutputStream out = null;
String nomeFile = "foto.jpg";
File file = new File(nomeFile);
try {
    in = new BufferedInputStream(new FileInputStream(nomeFile));
```

```
    out = new BufferedOutputStream(new FileOutputStream("nuova-foto.jpg"));
    byte [] b = new byte[(int) file.length()];
    in.read(b);
    out.write(b, 0, b.length);
    out.flush();
} catch (FileNotFoundException e){
    System.err.println("non ho trovato il file");
} catch (IOException e){
    e.printStackTrace();
} finally {
    if(in!=null)
        try {
            in.close();
        } catch (Exception e){
            e.printStackTrace();
        }
    if(out!=null)
        try {
            out.close();
        } catch (Exception e){
            e.printStackTrace();
        }
}
```

- `DataInputStream` e `DataOutputStream`, permettono di leggere/scrivere dati primitivi da/su un file. Alcuni metodi rispettivamente per `DataInputStream` e `DataOutputStream`:

  `int readInt()`, legge 4 byte e restituisce un int

  `double readDouble()`, legge 8 byte e restituisce un double

  `char readChar()`, legge 2 byte e restituisce un char

  `boolean readBoolean()`, legge 1 byte e restituisce false se il byte è zero, true altrimenti

  `int available()`, restituisce una stima dei byte che possono essere letti senza bloccarsi

  `void writeInt(int v)`, scrive un int nell'output stream come 4 byte

  `void writeDouble(double v)`, converte l'argomento in long e poi lo scrive nell'output stream come 8 byte

  `void readChar(int v)`, scrive un char nell'output stream come 2 byte

  `void readBoolean()`, scrive un boolean nell'output stream come 1 byte

```
Double [] doubles = {1.10, 9.99, 4.23};
String nomeFile = "prezzi.txt";
FileInputStream fileIn = null;
DataOutputStream out = null;
DataInputStream in = null;
try{
    out = new DataOutputStream(new FileOutputStream(nomeFile));
    for(Double el : doubles){       // scrivo in bytes in un file.txt
        out.writeDouble(el);
    }
    out.flush();
    in = new DataInputStream(new FileInputStream(nomeFile));
    while (in.available() > 0){     // leggo in double da un file.txt
        System.out.println("prezzo " + in.readDouble());
    }
} catch (FileNotFoundException e) {
    System.err.println("non ho trovato il file");
} catch (IOException e) {
    e.printStackTrace();
} finally {
```

```
    if(in!=null)
        try{
            in.close();
        } catch (Exception e){
            e.printStackTrace();
        }
    if(out!=null)
        try{
            out.close();
        } catch (Exception e){
            e.printStackTrace();
        }
}
```

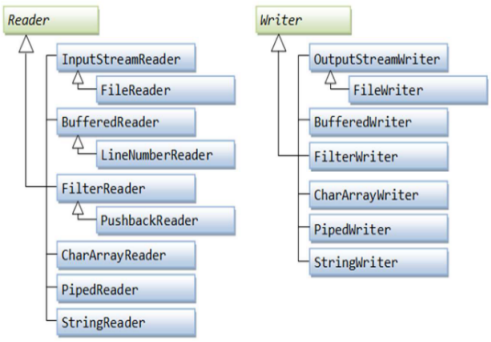
4.5 Stream di caratteri

`Reader` e **`Writer`** sono le due principali classi astratte per lo *stream di caratteri*. Alcuni metodi `Reader`:

`int read()`, legge un singolo carattere

`int read(char [] cbuf)`, legge i caratteri dallo stream e li carica nell'array `cbuf`

`int read(char[] cbuf, int off, int len)`, legge un numero di caratteri uguali a `len` dallo stream e li carica nell'array `cbuf` a partire dalla posizione `off`



Alcuni metodi `Writer`:

`void write(int c)` e `void write(String str)`, scrivi un singolo carattere e una stringa

`void write(char[] cbuf)`, scrive i caratteri dall'array `cbuf` allo stream di output

`void write(char[] cbuf, int off, int len)` e `void write(String str, int off, int len)` per scrivere una porzione di caratteri/stringa pari al numero di `len` e a partire da `off`

`void flush()`, per forzare la scrittura di tutti i dati a destinazione

Alcune implementazioni:

- `FileReader` e `FileWriter` per leggere/scrivere caratteri da o su un file
- `InputStreamReader` e `OutputStreamWriter` per convertire dati da bytes in caratteri
- `BufferedReader` e `BufferedWriter`, per leggere/scrivere testo da uno stream di caratteri utilizzando un buffer di dimensioni predefinite

4.6 Try with resource

Nel seguente codice:

```
public void printFile() throws IOException {
    InputStream input = null;
    try{
        input = new FileInputStream("file.txt");     // puo' sollevare eccezioni
        int data = input.read();                     // puo' sollevare eccezioni
        while(data != -1)
            data.read();                             // puo' sollevare eccezioni
        finally{
            if(input != null)
```

```
                input.close                                    // puo' sollevare eccezioni
            }
        }
}
```

il blocco `finally` viene eseguito anche se una eccezione viene rilevata nel blocco `try`. Se un'eccezione viene sollevata nel blocco `finally` solo questa viene propagata, e l'eccezione del blocco `try` ignorata anche se è più significativa. Il costrutto **`try-with-resource`** si occupa di chiudere automaticamente risorse aperte. Si utilizza un blocco `try` con uno o più argomenti tra parentesi, dove gli argomenti sono le risorse che Java garantisce di chiudere al termine del blocco

```
public void printFile() throws IOException {
    try(InputStream input = new FileInputStream("file.txt")){
        int data = input.read();
        while(data != -1)
            data = input.read();
    }
}
```

4.7 Java serialization

Gli oggetti esistono in memoria fino a che la JVM è in esecuzione: per la loro persistenza al di fuori della JVM occorre creare una rappresentazione dell'oggetto indipendente dalla JVM. La **serializzazione** consente di mantenere la persistenza degli oggetti al di fuori del ciclo di vita della JVM dove l'oggetto serializzato può essere scritto su un qualsiasi stream di output. La serializzazione effettua il salvataggio dello stato dell'oggetto, mentre, la deserializzazione ricostruisce lo stato dell'oggetto. Utilizzata in diversi contesti come:

- inviare oggetti su uno stream che rappresenta una connessione TCP
- generare pacchetti UDP: si scrive l'oggetto serializzato su uno stream di byte e poi si genera un pacchetto UDP

Per rendere un oggetto persistente, l'oggetto deve implementare l'interfaccia `Serializable` che non possiede nessun metodo ma attiva il meccanismo di serializzazione che verifica se la classe ha le caratteristiche per essere serializzata. La serializzazione vera e propria è gestita dalla classe `ObjectOutputStream` tale stream deve essere concatenato con uno stream di bytes come `FileOutputStream`.

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

```
public class Serialization {
    public static void main(String[] args) {
        String filename = "time.txt";
        PersistentTime time = new PersistentTime();
        try(FileOutputStream fos = new FileOutputStream(filename);
            ObjectOutputStream out = new ObjectOutputStream(fos);){
            out.writeObject(time);
        }
        catch (IOException e){e.printStackTrace();}
    }
}
```

```
public class Deserialization {
    public static void main(String[] args) {
        String filename = "time.txt";
        PersistentTime time = null;
```

```
        try(FileInputStream fis = new FileInputStream(filename);
            ObjectInputStream in = new ObjectInputStream(fis);){
            time = (PersistentTime) in.readObject();
        }
        catch (IOException | ClassNotFoundException e){e.printStackTrace();}

        System.out.println("Serialization: " + time.getTime());
        System.out.println("Current time: " + Calendar.getInstance().getTime());
    }
}
```

Il metodo `readObject()` legge la sequenza di bytes memorizzati in precedenza e crea un oggetto che è l'esatta replica di quello originale. Può leggere qualsiasi tipo di oggetto ma 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. Se non la trova viene sollevata una `ClassNotFoundException` ed il processo di deserializzazione viene abortito. Non tutti gli oggetti sono serializzabili:

- gli oggetti contenenti riferimenti specifici alla JVM o al SO (`Thread, OutputStream, File, ...`)
- le variabili marcate `transient` che sono quelle variabili che non devono essere scritte per questione di privacy (esempio numero carta di credito). Per rendere un oggetto persistente occorre marcare tutti i campi non serializzabili con questo campo
- le variabile statiche perché sono associate alla classe e non alla specifica istanza dell'oggetto che si sta serializzando

Tutti i componenti di un oggetto devono essere serializzabili: se ne esiste uno che non è serializzabile si solleva una `notSerializableException`. La serializzazione ha una limitata interoperabilità perché è utilizzabile quando sia l'applicazione che serializza l'oggetto che quella che lo deserializza sono scritte in Java. Per aumentare l'interroperabilità occorre utilizzare altre soluzioni: JSON, XML, e altri formati...

Chapter 5

TCP Socket

5.1 Protocolli di trasporto

In una applicazione di rete vi sono due o più processi in esecuzione su hosts diversi, distribuiti geograficamente sulla rete che *comunicano* e *cooperano* per realizzare una funzionalità globale. Per comunicare utilizzano dei protocolli, ovvero, insieme di regole che i partners devono seguire per comunicare correttamente. I protocolli di livello trasporto sono:

- **connection-oriented - TCP**: connessione stabile tra mittente e destinatario; vi sono tre fasi: apertura della connesione, invio dei dati sulla connessione, chiusura. Si utilizzano gli **stream socket**. Esempio: Web
- **connection-less - UDP**: non si stabilisce un canale di comunicazione dedicato; ogni messaggio viene instradato in modo indipendente dagli altri. Si utilizzano i **datagram socket**. Esempio: richieste DNS

Altre differenze:

TCP

- *Indirizzamento*: l'indirizzo del destinatario (indirizzo IP + porta) è specificato al momento dell'apertura della connessione
- *Ordinamento dei dati scambiati*: garantito ordinamento dei messaggi da parte del protocollo di trasporto
- *Utilizzo*: è richiesta la trasmissione affidabile dei dati

UDP

- *Indirizzamento*: l'indirizzo del destinatario (indirizzo IP + porta) viene specificato in ogni pacchetto
- *Ordinamento dei dati scambiati*: nessuna garanzia sull'ordinamento dei messaggi da parte del protocollo di trasporto
- *Utilizzo*: invio di un numero limitato di dati/non è importante che l'invio sia affidabile

5.2 Socket e indirizzi IP

La **socket** è uno standard di comunicazione utilizzato per connettere dispositivi distribuiti, diversi e eterogenei. È un end-point sull'host locale di un canale di comunicazione da/verso altri host, creato su richiesta dell'applicazione e controllato dal SO dell'host locale. Classi socket in Java:

- *connection-oriented* si sfruttano gli stream per modellare la connessione; lato client si utilizza la classe `Socket` mentre lato server `ServerSocket` e `Socket`.
- *connection-less* si utilizza la classe `DatagramSocket` sia per il client che per il server

Ogni comunicazione è individuata dalla seguente 5-upla:

- protocollo di trasporto (TCP o UDP)
- indirizzo IP del computer locale e porta locale
- indirizzo IP del computer remoto e la porta remota

Gli indirizzi IP semplificano l'elaborazione effettuata dai routers, ma sono poco leggibili per gli utenti della rete. Quindi a livello applicativo per ogni host della rete si utilizzano dei nomi simbolici. Il DNS tradurrà poi i nomi in indirizzi IP

5.3 InetAddress class

La classe `InetAddress` è una rappresentazione ad alto livello di un indirizzo IP (IPv4 e IPv6). Un oggetto di tipo `InetAddress` è formato dalla coppia `String hostName` che rappresenta il nome simbolico dell'host e `byte [] address` che rappresenta l'indirizzo IP. La classe non definisce costruttori, ma fornisce tre metodi statici:

`InetAddress.getByName (String hostname)`, cerca l'indirizzo IP corrispondente all'host di nome `hostname` e restituisce un oggetto di tipo `InetAddress`. Richiede una interrogazione del DNS per risolvere il nome dell'host.

```
InetAddress addr1 = InetAddress.getByName("www.codejava.net");
System.out.println(addr1.getHostAddress());
```

`InetAddress.getAllByName (String hostname)`, per hosts che possiedono più indirizzi come i web services

`InetAddress.getLocalHost ()`, per reperire nome simbolico ed indirizzo IP del computer locale

questi metodi possono sollevare l'eccezione `UnknownHostException` se non riescono a risolvere il nome dell'host. Metodi getter per reperire informazioni come: `String getHostName()`, `String getAddress()`, `String getHostAddress()`. Essi non effettuano collegamenti con il DNS, non sollevano eccezioni. `InetAddress` effettua l'overriding di tre metodi di `Object`:

`boolean equals()`, un oggetto o è uguale ad un oggetto `InetAddress` se e solo se o è un oggetto `InetAddress` e rappresenta lo stesso indirizzo IP (non è richiesto che abbiamo lo stesso hostname)

`int hashCode()`, calcola l'hash dei 4 bytes dell'indirizzo e restituisce un int

`String toString()`, restituisce ⟨*nome dell'host/indirizzo dotted quad*⟩; se non esiste il nome allora restituisce stringa vuota + *indirizzo dotted quad*

I metodi descritti effettuano caching dei nomi/indirizzi risolti, poiché l'accesso al DNS è una operazione costosa. Nella cache vengono memorizzati anche i tentativi di risoluzione non andati a buon fine che sono validi per un intervallo breve (tipicamente 10 secondi). Tramite `Security.setProperty` è possibile impostare il numero di secondi in cui una entrata nella cache rimane valida tramite le proprietà

- `networkaddress.cache.ttl`

```
final String CACHINGTIME = "1000";
Security.setProperty("networkaddress.cache.ttl", CACHINGTIME);
```

- `networkaddress.cache.negative.ttl`

5.4 Network interface

Ogni host di una rete IPv4 e IPv6 è connesso alla rete mediante una o più interfacce. Ogni interfaccia è caratterizzata da un indirizzo IP. Può essere una interfaccia virtuale non associata direttamente ad un dispositivi fisico. Se un host, ad esempio un router presente più di una interfaccia sulla rete, allora si hanno più indirizzi IP per lo stesso host, uno per ogni interfaccia. Per **multi-homed hosts** si intende un host che possiede un insieme di interfacce verso la rete e quindi un insieme di indirizzi IP, come un router. Il seguente programma restituisce un oggetto `NetworkInterface` che rappresenta la network interface collegata all'indirizzo "127.0.0.1".

```
InetAddress local = InetAddress.getByName("127.0.0.1");
NetworkInterface ni = NetworkInterface.getByInetAddress(local);
System.out.println(ni);
```

L'istruzione `NetworkInterface.getNetworkInterfaces()` restituisce una enumeration di tutte le interfacce della macchina

```
Enumeration <NetworkInterface> nets = NetworkInterface.getNetworkInterfaces();
for(NetworkInterface e : Collections.list(nets)){
    Enumeration <InetAddress> in = e.getInetAddresses();
        for(InetAddress el : Collections.list(in))
            System.out.println(el); // indirizzi IP associati all'interfaccia
}
```

L'indirizzo "127.0.0.1" è un indirizzo di **loopback**: può essere utilizzato da un nodo per inviare pacchetti a se stesso; utilizzabile per testare applicazioni. Quando si usa un indirizzo di loopback si possono eseguire client e server in locale sullo stesso host. Ogni dato spedito utilizzando questo indirizzo in realtà non lascia l'host locale

5.5 TCP socket programming

Paradigma client server:

- **client**, contatta il server che deve essere in esecuzione al momento della richiesta di contatto; crea un TCP socket specificando l'indirizzo IP e la porta associata al processo in esecuzione sul server; quando il client crea il socket viene inviata una richiesta di connessione con il server
- **server**, deve aver creato un *welcoming socket* che consente di ricevere richiesta di contatto dal client; quando accetta la richiesta di connessione crea un nuovo connection socket per comunicare con quel client;

La comunicazione vera e propria avviene mediante la coppia di active sockets presenti nel client e nel server. Quindi lato server esistono due tipi di socket TCP: **welcome sockets (passive)** utilizzati dal server per accettare le richieste di connessione e **connection sockets (active)** supportano lo streaming di byte tra client e server. Passi:

1. il *server* apre una `ServerSocket` e aspetta connessioni dai clienti. Quindi questa socket server solo per aspettare clienti e iniziare la connessione
2. il *client* apre una `Socket` e si connette al server
3. quando arriva una richiesta da un client, il *server* crea una `Socket` dedicato a questo client. Client e server comunicano utilizzando sempre `Socket`

5.5.1 Socket class

I costruttori della classe `Socket` sono:

`Socket(InetAddress host, int port)`, crea una active socket e tenta di stabilire tramite essa una connessione con l'host individuato da `InetAddress` sulla porta `port`.

`Socket(String host, int port)`, l'host può essere individuato dal suo nome simbolico `host`; viene interrogato automaticamente il DNS

`Socket(InetAddress host, int port, InetAddress localIA, int localPort)` insieme ad a `Socket(String host, int port, InetAddress localIA, int localPort)` tentano di creare una connessione verso `host`, sulla porta `port` dalla interfaccia locale `localIA` e dalla porta locale `localPort`

I costruttori lanciano `IOException` se la connessione viene rifiutata. Nel seguente programma il client richiede la connessione tentando di creare una socket su ognuno delle prime 1024 porte di un host

```
public class PortScanner {
    public static void main(String[] args) {
        String host = "localhost";
        for(int i = 1; i<=1024; i++){
            try{
                Socket s = new Socket(host, i);
                System.out.println("Esiste un servizio sulla porta " +i);
            }
            catch (UnknownHostException e){
                System.out.println("Host sconosciuto");
                break;
            }
            catch (IOException e){
                System.out.println("Non esiste un servizio sulla porta " +i);
            }
        }
    }
}
```

Per ottimizzare il comportamento del programma è meglio utilizzare `Socket(InetAddress host, int port)` per la presenza di cache locale dei nomi.

5.5.2 ServerSocket class

Socket speciale che aspetta delle connessioni: esiste ed è attivo anche senza nessun cliente. I `ServerSocket` non sono usati per trasmettere dati all'applicazione, servono solo ad aspettare, negoziare la connessione con i nuovi clienti e creare una socket normale da usare nell'applicazione. Costruttori:

> `ServerSocket(int port)` e `ServerSocket(int port, int length)`, costruiscono un listening socket, associandolo alla porta `port`. La variabile `lenght` si riferisce alla lunghezza della coda in cui vengono memorizzate le richieste di connessione. Se la coda è piena, ulteriori richieste di connessioni sono rifiutate
>
> `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 video, ad esempio un host con due indirizzi IP, uno visibile da Internet, l'altro visibile solo a livello di rete locale. Se si vuole servire solo le richieste in arrivo dalla rete locale, si associa il connection socket all'indirizzo IP locale

Con i costruttori si istanziano delle listening sockets specificando su quale porta si è in ascolto. Per accettare una nuova connessione dal connection socket si utilizza il metodo `accept()`. Comportamento:

- 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-out)
- all'arrivo di una richiesta, il processo si sblocca e costruisce un nuovo socket S tramite cui avviene la comunicazione effettiva tra client e server

5.6 Modellare la connessione mediante stream

Una volta stabilita la connessione tra client e server, inizia lo scambio dati. La connessione è modellata come uno stream: si associa uno stream di input o di output ad un socket. Ogni valore scritto sullo stream di output associato al socket viene copiato nel `Send Buffer` del TCP. Ogni valore letto dallo stream viene prelevato dal `Receive Buffer` del TCP. Ciclo di vita tipico di un server:

```
ServerSocket servSock = new ServerSocket (port);
while(!done){
    // si accetta la richiesta
    Socket sock = servSock.accept();
    // ServerSocket e' connesso
    InputStream in = sock.getInputStream();
    OutputStream out = sock.getOutputStream();

    sock.close();
}
servSock.close();
```

5.7 Half closure

L'istruzione `close()` chiude la socket in entrambe le direzioni. La **half closure** è una chiusura del socket in una sola direzione: `shutdownInput()` o `shutdownOutput()`. In molti protocolli, il client manda una richiesta al server e poi attende la risposta

```
try(Socket connection = new Socket("www.somesite.com", 80)){
    Writer out = new OutputStreamWriter(connection.getOutputStream(), "8859_1");
    out.write("GET / HTTP 1.0\r\n\r\n");
    out.flush();
    connection.shutdownOutput();
} catch(IOException e){
    e.printStackTrace();
}
```

scritture successive sollevano una `IOException`

Chapter 6

UDP socket

6.1 Quando usare UDP

In certi casi TCP offre più di quanto sia necessario, ad esempio se: non si è interessati a garantire che tutti i messaggi vengano recapitati; si vuole evitare l'overhead dovuto alla ritrasmissione dei messaggi e non è necessario leggere i dati nell'ordine con cui sono stati spediti. UDP supporta una comunicazione *connection-less* e fornisce un insieme molto limitato di servizi rispetto a TCP. Alcuni esempi di quando usare UDP:

- stream video/audio: meglio perdere un frame che introdurre overhead nella trasmissione di ogni frame
- se tutti gli host di un ufficio inviano, ad intervalli di tempo regolari un *keep-alive* ad un server centrale. La perdita di un *keep-alive* o se un messaggio spedito alle 10:05 arrivi prima di quello spedito alle 10:07 non sono importanti
- compravendita di azioni: la perdita di una variazioni di prezzo può essere tollerata per titoli azionari di minore importanza

Alcuni servizi su UDP: DNS, TFTP, alcuni protocolli peer-to-peer. Lo slogan per utilizzare UDP è "*Timely, rather than orderly and reliable delivery*"

6.2 UDP in Java

Ogni pacchetto è chiamato **datagram** ed è indipendente dagli altri. Non è richiesto un collegamento prima di inviare un pacchetto ma è necessario specificare l'indirizzo del destinatario per ogni pacchetto spedito. Classi UDP Java:

- `DatagramPacket` per costruire i datagram (sia per client che server); usato per riempire e leggere pacchetti; contiene indirizzo IP e porta destinatario
- `DatagramSocket`, per creare i sockets (sia per client che server); invia e riceve `DatagramPacket`; conosce solo la porta locale dalla quale inviare/ricevere pacchetti; una `DatagramSocket` può inviare pacchetti a più destinazioni (non c'è una connessione tra due applicazioni); non esiste uno stream tra applicazioni

UDP in breve:

Inviare un datagramma

- creare un `DatagramSocket` e collegarlo ad una porta locale
- 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 `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`

6.2.1 DatagramPacket

Un oggetto `DatagramPacket` contiene: un riferimento ad un array di byte che contiene i dati da spedire (o da ricevere); un insieme di informazioni per individuare la posizione dei dati da estrarre (o inserire) dall'array di byte: `lenght, offset`; eventuali informazioni di addressing come indirizzo IP e porta destinatario se il datagram deve essere spedito. Vi sono 2 costruttori per **ricevere i dati**:

`DatagramPacket(byte [] buffer, int lenght)`

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

costruiscono un `DatagramPacket` per ricevere un messaggio di lunghezza `length`. Esso indica il numero di byte nel buffer che saranno usati per ricevere i dati. Deve essere minore o uguale a `buffer.length`. Il buffer viene passato vuoto e poi verrà riempito al momento della ricezione di un pacchetto con il suo payload. Se l'offset è settato, la copia avviene dalla posizione individuata da esso. La copia del payload termina quando l'intero pacchetto è stato copiato, oppure, quando `length bytes` sono stati copiati se il payload è più grande (`getLength()` indica il numero di bytes effettivamente ricevuti). Vi sono 4 costruttori **per inviare i dati**:

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

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

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

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

La variabile `length` indica il numero di bytes che devono essere copiati dal buffer nel datagramma da inviare, a partire dal byte indicato da `offset` (se indicato). Invece `destination` + `port` individuano il destinatario. I **metodi set** invocati su un oggetto `DatagramPacket` inseriscono esplicitamente un riferimento al byte array in esso:

`void setData(byte[] buffer)`, setta il payload di "this" packet: `offset` uguale a 0 e `lenght` uguale a `buffer.lenght`

`void setData(byte[] buffer, int offset, int lenght)`, aggiorna sia `offset` che `lenght`

`void setPort(int iport)`, setta la porta nel datagram

`void setLength(int length)`, setta la lunghezza del payload del datagram

`void setAddress(InetAddress iaddr)`, setta l'`InetAddress` della macchina a cui il payload è diretto. Utile quando si deve mandare lo stesso datagram a più destinatari

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

`void setSocketAddress(SocketAddress addr)` utile per inviare risposte

```
DatagramPacket input = new DatagramPacket(new byte[8192], 8192);
socket.receive(input);
DatagramPacket output = new DatagramPacket("Hello here"
                                           .getBytes("UTF-8"), 11);
Socket address = input.getSocketAddress();
output.setSocketAddress(address);
socket.send(output);
```

Metodi get:

`InetAddress getAddress()`, restituisce l'indirizzo IP della macchina a cui il datagram è stato inviato, oppure, della macchina a cui è stato spedito

`int getPort()`, restituisce il numero di porta sull'host remoto cui il datagram è stato inviato, o dell'host a cui è stato spedito

`byte [] getData()` restituisce un byte array contenente i dati del buffer associato al datagram: ignora offset e lunghezza

`int getLength()` e `int getOffset()`, restituisce la lunghezza/offset del datagram da inviare o di quello ricevuto

`SocketAddress getSocketAddress()`, restituisce (IP + numero di porta) del datagram sull'host remoto cui il datagram è stato inviato, o dell'host a cui è stato spedito

6.2.2 DatagramSocket

Costruttori:

`DatagramSocket()`, crea un socket e lo collega ad una qualsiasi porta libera disponibile sull'host locale. Con l'istruzione `getLocalPort()` serve per reperire la porta allocata. Il server può inviare la risposta, prelevando l'indirizzo del mittente (IP + porta) dal pacchetto ricevuto

`DatagramSocket(int port)`, crea un socket e lo collega alla porta specificata sull'host locale. Il server crea un collegato ad una porta che rende nota ai client; la porta è allocata a quel servizio (porta non effimera); solleva una eccezione quando la porta è già utilizzata, oppure se non si hanno i diritti

Per inviare pacchetti si utilizza:

`void send(DatagramPacket dp)`, il processo che esegue la `send()` prosegue la sua esecuzione, senza attendere che il destinatario abbia ricevuto il pacchetto: non è bloccante

```
public class Sender {
    public static void main(String[] args) {
        try(DatagramSocket clientSocket = new DatagramSocket()){
            byte [] buffer = "1234567890abcdefghijklmnopqrstuvwxyz"
                                                    .getBytes("US-ASCII");
            InetAddress address = InetAddress.getByName("127.0.0.1");
            for(int i = 1; i < buffer.length; i++){
                DatagramPacket myPacket =
                                new DatagramPacket(buffer, i, address, 40000);
                clientSocket.send(myPacket);
                Thread.sleep(200);
            }
        }
        catch (Exception e){e.printStackTrace();}
    }
}
```

Per ricevere pacchetti:

`void receive(DatagramPacket dp)`, il processo che esegue la `receive()` si blocca fino al momento in cui viene ricevuto un pacchetto. Per evitare attese indefinite è possibile associare al socket un time-out:

`void setSoTimeout(int timeout)`, per evitare attese indefinite è possibile associare al socket un time-out

```
public class Receiver {
    public static void main(String[] args) {
        try(DatagramSocket serverSock = new DatagramSocket(40000)){
            serverSock.setSoTimeout(20000);
            byte [] buffer = new byte[100];
```

```
            DatagramPacket receivedPacket =
                                    new DatagramPacket(buffer, buffer.length);
            while (true){
                serverSock.receive(receivedPacket);
                String byteToString = new String(receivedPacket.getData(), 0,
                                        receivedPacket.getLength(), "US-ASCII");
                System.out.println("Lenght " + receivedPacket.getLength() +
                                    " data " + byteToString);
            }
        }
        catch (SocketTimeoutException e){
            System.out.println("bye bye");
        }
        catch (IOException e){e.printStackTrace();}
    }
}
```

6.3 Send/Receive buffers

Ad ogni socket sono associati due buffers: uno per la ricezione ed uno per la spedizione. Questi buffers sono gestiti dal sistema operativo, non dalla JVM. La loro dimensione dipende dalla piattaforma su cui il programma è in esecuzione

- **Receive buffers**, deve essere almeno uguale a quella del datagram più grande che può essere ricevuto tramite quel buffer. Può consentire di bufferizzare un insieme di datagram, nel caso in cui la frequenza con cui essi vengono ricevuti sia maggiore di quella con cui l'applicazione esegue la `receive()` e quindi preleva i dati dal buffer
- **Send buffer**, viene utilizzata per stabilire la massima dimensione del datagram. Consente di bufferizzare un insieme di datagram, nel caso in cui la frequenza con cui essi vengano generati sia molto alta (`send()`) rispetto alla frequenza con cui il supporto li preleva e spedisce sulla rete

6.4 Generare streams di byte

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

`Byte[] getBytes()` applicato ad un oggetto `String`, restituisce una sequenza di bytes che codificano i caratteri della stringa usando la codifica di default dell'host e li memorizza nel vettore

`String (byte[] bytes, int offset, int length)`, costruisce un nuovo oggetto di tipo `String` prelevando `length bytes` dal vettore `bytes`, a partire dalla posizione `offset`

Un altro meccanismo per generare pacchetti a partire da dati strutturati è utilizzare i filtri per generare streams di byte a partire da dati strutturati/ad alto livello. Gli oggetti della classe `ByteArrayOutputStream` rappresentano uno stream di bytes. Ogni dato scritto viene riportato in un buffer di memoria a dimensione variabile. Costruttori:

`ByteArrayOutputStream()`, crea un nuovo `ByteArrayOutputStream`

`ByteArrayOutputStream(int size)` crea un nuovo `ByteArrayOutputStream` con una dimensione del buffer specificata in bytes

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

`int size()`, restituisce il numero di bytes memorizzati nello stream

`void reset()`, svuota il buffer, assegnando 0 il numero di bytes memorizzati nello stream. Tutti i dati precedentemente scritti vengono eliminati

`byte toByteArray()`, restituisce un vettore in cui sono stati copiati tutti i bytes presenti nello stream. Non svuota il buffer

Ad un `ByteArrayOutputStream` può essere collegato un altro filtro: `DataOutputStream`. La classe `ByteArrayInputStream` crea uno stream di byte

`DataOutputStream (byte [] buf)`, crea un nuovo `ByteArrayInputStream` a partire dai dati contenuti nel vettore di byte `buf)`

`DataOutputStream (byte [] buf, int offset, int length)`, crea un nuovo `ByteArrayInputStream` copiando `length bytes` iniziando dalla posizione `offset`

È possibile concatenare un `ByteArrayInputStream` con `DataInputStream`

Chapter 7

JSON e Java.NIO

7.1 JSON

JSON è un formato lightweight per l'interscambio di dati, indipendente dalla piattaforma poiché è testo scritto in notazione JSON. È indipendente dal linguaggio di programmazione anche se la sintassi è basata su un sottoinsieme di JavaScipt. JSON è basato su due strutture: coppie chiave-valore (le chiavi sono sempre stringhe) e liste ordinate di valori. I tipi di dato ammissibili per i valori sono: stringhe, interi, booleani, object, ... JSON definisce due strutture:

- **JSON array**, una raccolta ordinata di valori. Esempio: `["Ford", "BMW", "Fiat"]`. Un array è delimitato da parentesi quadre e i valori sono separati da virgole. In JSON l'array è eterogeneo: ciascun elemento dell'array può essere di qualsiasi tipo e i valori possono essere annidati (array dentro altri array). Quando si traduce un array JSON in una struttura dati di un linguaggio di programmazione vi è un mapping diretto con array, list, vector, ...
- **JSON object**, una serie non ordinata di coppie (nome, valore). Esempio: `{"name":"John", "age":30, "car": ["Ford", "BMW", "Fiat"]}`. Delimitato da parentesi graffe, le coppie sono separate da virgole. Un JSON object può contenere un altro JSON object e un JSON array

Un altro linguaggio che è utilizzato come formato di interscambio è **XML**. Alcune librerie per serializzare/deserializzare oggetti in/da JSON sono: GSON, Jackson, JSON-Simple, FastJSON, ...

7.1.1 GSON

La libreria **GSON** presenta due semplici metodi:

- `String toJson(Object src)`, dato un oggetto Java restituisce la rappresentazione JSON dell'oggetto
- `<T> T fromJson(String json, Class <T> classOfT)`, data una stringa in formato JSON la deserializza ad oggetto Java

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

Per serializzare una classe `Person`:

```
Persona p = new Persona("Alice", 20);
Gson gson = new GsonBuilder().setPrettyPrinting().create(); // formattare output
String json = gson.toJson(p);
System.out.println(json);
```

Per deserializzare:

```
Gson gson = new Gson();
String json = "{ name: \"Alice\", age: 20 }";
Person p = gson.fromJson(json, Person.class);
System.out.println(p);
```

GSON streaming fornisce un supporto per leggere/scrivere file JSON di grosse dimensioni con le seguenti classi:

- `JsonReader`, legge un file JSON come un flusso di token che include sia i valori che i delimitatori di inizio e fine degli oggetti e array. Alcuni metodi:

`void beginArray()` e `void beginObject()`, consuma il token successivo dal flusso JSON che afferma che è l'inizio di un nuovo array/oggetto

`void endArray()` e `void endObject()`, consuma il token successivo dal flusso JSON che afferma che è la fine dell'array/oggetto corrente

`boolean hasNext()`, restituisce true se l'array/oggetto corrente ha un altro elemento

`boolean nextBoolean(), int nextInt(), double nextDouble(), String nextString()`... restituisce e consuma il prossimo token booleano, intero, double, stringa, ...

`void close()`, si chiude lo stream e si rilasciano le risorse

```
Gson gson = new GsonBuilder().setPrettyPrinting().create();
FileOutputStream fos = new FileOutputStream("output.json");
OutputStreamWriter ow = new OutputStreamWriter(fos);
Persona p = new Persona("Giovanni", 31);
String personaJson = gson.toJson(p);
ow.write('[');
for(int i = 0; i<10; i++){  // si serializzano 10 p
    if (i != 0) ow.write(",");
    ow.write(personaJson);
    System.out.println(personaJson);
}
ow.write("]");
ow.flush();

FileInputStream fin = new FileInputStream("output.json");
JsonReader reader = new JsonReader(new InputStreamReader(fin));
reader.beginArray();        // si legge il file JSON
while (reader.hasNext()) {
    Persona person = new Gson().fromJson(reader, Persona.class);
    System.out.println(person);
}
reader.endArray();
```

```
...
{
  "name": "Giovanni",
  "age": 31
}
{
  "name": "Giovanni",
  "age": 31
}
name: Giovanni age: 31
name: Giovanni age: 31
...
```

- `JsonWriter`, scrive un valore JSON un token alla volta su uno stream che include sia i valori che i delimitatori di inizio e fine degli oggetti e array. Alcuni metodi:

  `JsonWriter beginArray()` e `JsonWriter beginObject()`, inizia la codifica di un nuovo array/oggetto

  `JsonWriter endArray()` e `JsonWriter endObject()`, termina la codifica dell'array/oggetto

  `JsonWriter name(String name)`, codifica il nome (la chiave)

  `JsonWriter value(boolean value), value(double value), value(String value)`..., codifica un valore

  `void flush()`, per forzare la scrittura di tutti i dati a destinazione

  `void close()`, si chiude lo stream e si rilasciano le risorse

```
FileOutputStream fout = new FileOutputStream("output.json");
JsonWriter writer = new JsonWriter(new OutputStreamWriter(fout, "UTF-8"));
String [] cars = {"Ford", "BMW", "Fiat"};
writer.setIndent("  "); // settare l'indentazione nel file
writer.beginObject();
writer.name("name").value("Giovanni");
writer.name("age").value("31");
writer.name("car");
writer.beginArray();
for(String el : cars){
    writer.value(el);
}
writer.endArray();
writer.endObject();
writer.close();
```

7.2 Java.NIO

Java.NIO è orientato al blocco, cioè è un I/O dove ogni operazione produce o consuma dei blocchi di dati. Gli obiettivi sono: incrementare la performance dell'I/O; fornire un insieme eterogeneo di funzionalità per l'I/O e aumentare l'espressività delle applicazioni. Gli svantaggi sono: risultati dipendenti dalla piattaforma su cui si eseguono le applicazioni e primitive a più basso livello di astrazione quindi perdita di semplicità ed eleganza rispetto allo stream-based I/O. Java IO è basato su stream di byte o di caratteri a cui si possono applicare filtri. NIO invece opera su:

- **buffer**, contengono dati appena letti o che devono essere scritti su un channel. Deve essere gestito in maniera esplicita dal programmatore
- **channel**, sono un po' l'equivalente degli stream: oggetti dai quali si può leggere/scrivere dati attraverso i buffer. Gli stream sono monodirezioanali, mentre i channel sono bidirezionali: lo stesso channel può leggere dal dispositivo e scrivere sul dispositivo. Inoltre a differenza degli stream non si scrive/legge mai direttamente da un canale, ma si deve allocare un buffer. Quindi trasferimento dati dal canale nel buffer e viceversa
- **selectors**, oggetto in grado di monitorare un insieme di canali con un unico thread (vedi prossimo capitolo)

7.3 Channel

`Channel` è un interfaccia che è radice di una gerarchia di interfacce. Alcune implementazioni:

- `FileChannel`: legge/scrive dati su un file (è bloccante)
- `DatagramChannel`: legge/scrive dati sulla rete via UDP
- `SocketChannel`: legge/scrive dati sulla rete via TCP
- `ServerSocketChannel`: attende richieste di connessioni TCP e crea un `SocketChannel` per ogni connessione creata

Gli ultimi tre possono essere non bloccanti. Per leggere dati da un file e copiarli in un altro file:

```
    // canale associato ad un FileInputStream
try(FileInputStream fin = new FileInputStream("testIn.txt");
    FileChannel in = fin.getChannel();
    // canale associato ad un FileOuputStream
    FileOutputStream fout = new FileOutputStream("testOut.txt");
    FileChannel out = fout.getChannel();){
        // creazione di un ByteBuffer
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        while (in.read(buffer) != -1){  // lettura dal canale
            buffer.flip();
            while (buffer.hasRemaining()){
                out.write(buffer);      // scrittura nel canale
            }
            buffer.clear();
        }
} catch (FileNotFoundException e) {
    System.err.println("file non trovato");
} catch (IOException e) {
    e.printStackTrace();
}
```

Per leggere dal canale non è necessario specificare quanti byte il sistema operativo deve leggere, ma è necessario avere delle variabili interne all'oggetto `Buffer` che mantengono lo stato del buffer. Anche per indicare quale porzione del `Buffer` è significativa occorre modificare le variabili interne di stato

7.4 Buffer

I buffer sono dei contenitori di dimensione finita. Lo stato interno di un `Buffer` è caratterizzato da alcune variabili:

- **`capacity`**, dimensione massima del buffer definita al momento della creazione del buffer: non può essere modificata. Solleva una eccezione se si tenta di leggere/scrivere in/da una posizione superiore a `capacity`. Metodo get: `int capacity()`
- **`limit`**, indica il limite della porzione del buffer che può essere letta/scritta. Per le scritture è uguale a `capacity`. Per le letture delimita la porzione di `Buffer` che contiene dati significativi. Metodi get e set: `int limit()`, `Buffer limit(int)`
- **`position`** indica la posizione attuale nel buffer che individua il prossimo elemento da leggere/scrivere.
- **`mark`**, memorizza una posizione particolare. Valgono sempre le seguenti relazioni:

$$0 \leq mark \leq position \leq limit \leq capacity$$

Alcuni metodi utili:

`clear()`, per passare da modalità lettura a modalità scrittura; fa in modo che `limit = capacity` e `position = 0`. I dati non sono cancellati però saranno sovrascritti

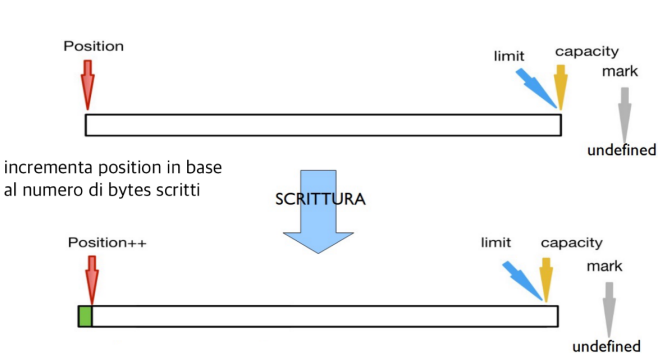
`flip()`, per passare da modalità scrittura a modalità lettura; fa in modo che `limit` diventa il puntatore all'ultimo elemento da leggere, assegna `position` a `limit` e setta `position` a 0

`compact()`, ritorna in modalità scrittura compattando il buffer. Utile se il contenuto del buffer non è stato completamente letto e si inizia una nuova scrittura. I bytes non ancora letti vengono copiati all'inizio del buffer

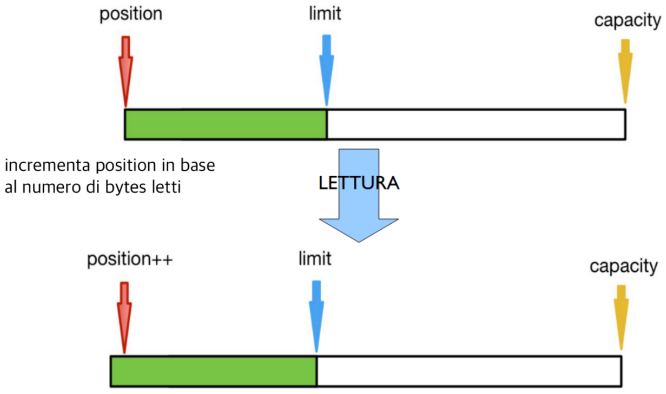
`rewind()`, prepara a rileggere i dati che sono nel buffer, ovvero, resetta position a 0 e non modifica `limit`

`remaining()`, restituisce il numero di elementi nel buffer compresi tra `position` e `limit`

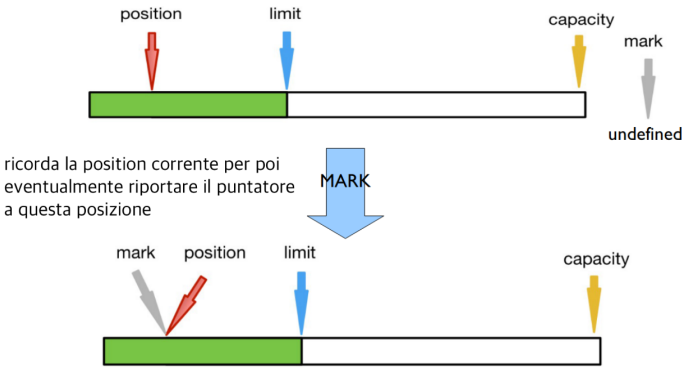
`hasRemaining()`, restituisce true se `remaining()` è maggiore di 0

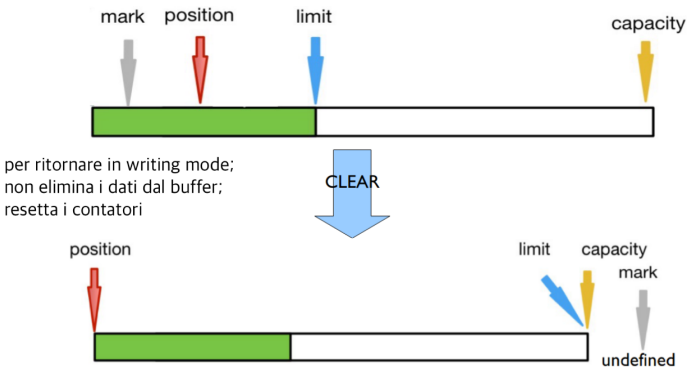


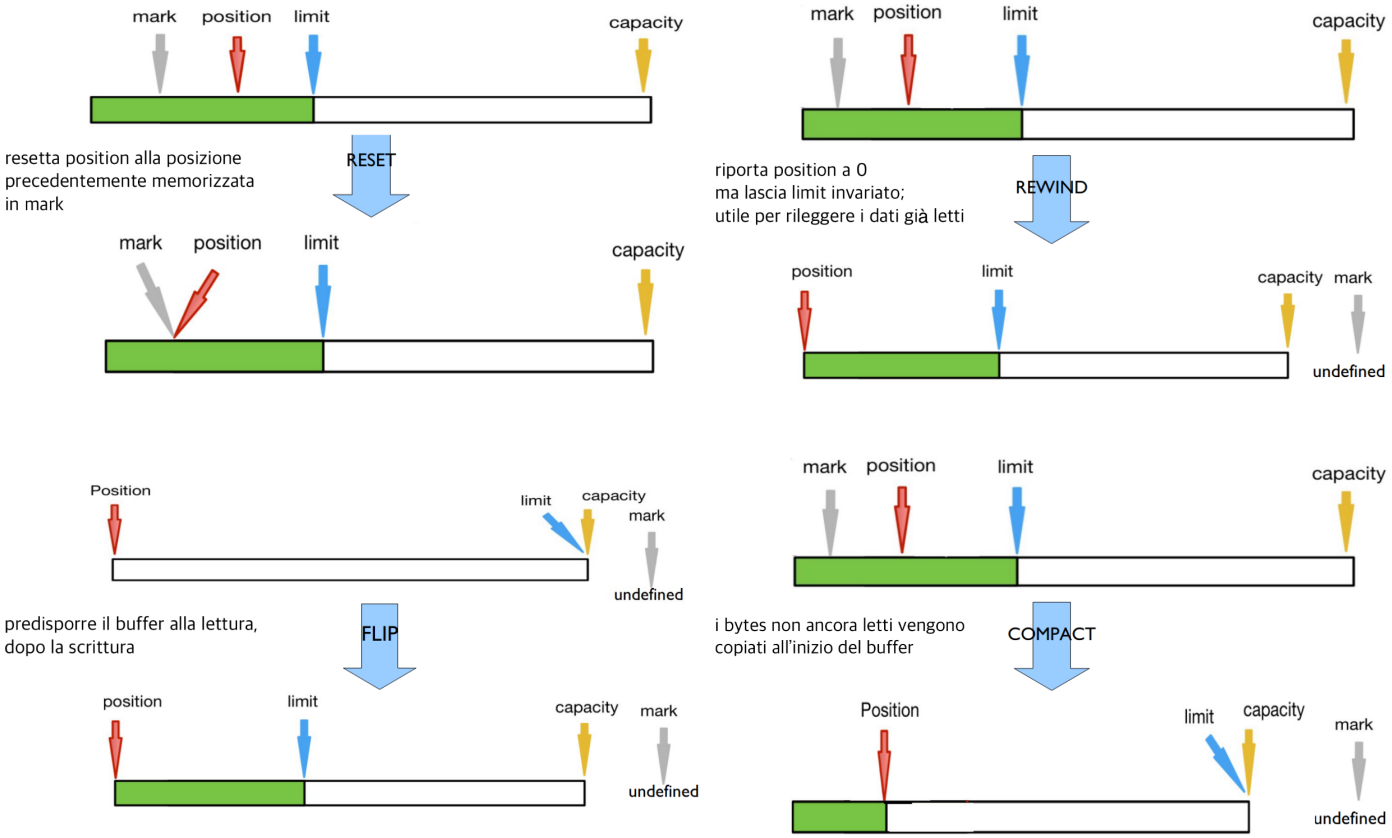
scrivere nel buffer



leggere dal buffer








```
ByteBuffer byteBuffer = ByteBuffer.allocate(10);
System.out.println(byteBuffer); // [pos=0 lim=10 cap=10]
byteBuffer.putChar('a');
System.out.println(byteBuffer); // [pos=2 lim=10 cap=10]
byteBuffer.putInt(1);
System.out.println(byteBuffer); // [pos=6 lim=10 cap=10]
// da scrittura a lettura
byteBuffer.flip();
System.out.println(byteBuffer); // [pos=0 lim=6  cap=10]
System.out.println(byteBuffer.getChar());   // a
System.out.println(byteBuffer); // [pos=2 lim=6  cap=10]
// da lettura a scrittura
byteBuffer.compact();
System.out.println(byteBuffer); // [pos=4 lim=10 cap=10]
byteBuffer.putInt(2);
System.out.println(byteBuffer); // [pos=8 lim=10 cap=10]
// da scrittura a lettura
byteBuffer.flip();
System.out.println(byteBuffer.getInt());    // 1
System.out.println(byteBuffer.getInt());    // 2
System.out.println(byteBuffer); // [pos=8 lim=8  cap=10]
byteBuffer.rewind();
System.out.println(byteBuffer); // [pos=0 lim=8  cap=10]
System.out.println(byteBuffer.getInt());    // 1
byteBuffer.mark();
System.out.println(byteBuffer); // [pos=4 lim=8  cap=10]
System.out.println(byteBuffer.getInt());    // 2
System.out.println(byteBuffer); // [pos=8 lim=8  cap=10]
byteBuffer.clear();
System.out.println(byteBuffer); // [pos=0 lim=10 cap=10]
```

Vi sono soluzioni diverse per memorizzare dati in un buffer:

- in un array privato all'interno dell'oggetto `Buffer`

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

  crea sullo heap un oggetto `Buffer` che incapsula una struttura per memorizzare gli elementi e le variabili di stato

- in un array creato dal programmatore (wrapping)

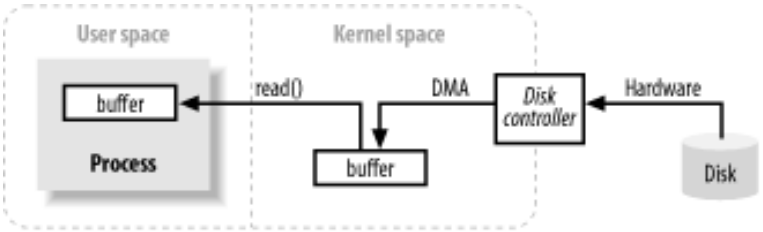
```
byte[] backingArray = new byte[100];
ByteBuffer byteBuffer = ByteBuffer.wrap(backingArray);
```

- con buffer diretti nello spazio di memoria nativa del kernel, esterno dell'heap della JVM

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

7.4.1 Gestione dei buffer

Il problema dell'I/O e la gestione dei buffer: vi è un buffer nell'*user space* (JVM) e un altro buffer nel *kernel space*. Con il meccanismo di DMA si possono leggere i dati dal dispositivo esterno nel buffer del kernel space. Poi i dati sono copiati dal kernel space nello user space. Si può ottimizzare questo processo? La gestione ottimizzata di questi buffer comporta un notevole miglioramento della performance dei programmi



NIO con i **direct buffer** consente di leggere i dati direttamente dal buffer messo a disposizione dal kernel, questo riduce il numero di copie del dato letto e quindi si ha una migliore performance. Lo svantaggio è il maggior costo di allocazione/deallocazione. Inoltre il buffer non è allocato sullo heap e di conseguenza il garbace collector non può recuperare memoria

Chapter 8

TCP e UDP con NIO

8.1 Introduzione

La rete è lenta rispetto alla CPU e alla memoria. Se si assegna un thread ad una socket, il thread occuperà la CPU con operazioni di calcolo, elaborazione, e preparazione di risposte e poi si bloccherà in attesa di eventi sul canale di comunicazione. Questo comporta che il thread continua a consumare memoria e si possono avere multipli switch di contesto. L'approccio di NIO, in modalità non bloccante e utilizzando *selector*, è di avere un solo thread che gestisce più connessioni (invece di assegnare un thread per connessione). Quindi se una connessione è pronta per inviare i dati, le passa i dati e poi si sposta alla connessione successiva, mentre altri thread possono essere dedicati per la preparazione ed elaborazione di task. NIO supporta sia la modalità bloccante che la modalità non bloccante

- **Blocking IO**, operazioni bloccanti su stream:

  `read()`: il thread si blocca fino a quando non è stato letto un byte

  `accept()`: il thread si blocca fino a che non viene stabilita una nuova connessione

  `write(byte [] buffer)`: il thread si blocca fino a che tutto il contenuto del buffer viene inviato sul canale

- **Non Blocking IO**, la chiamata di sistema restituisce il controllo alla applicazione, prima che l'operazione richiesta sia stata pienamente soddisfatta. Per completare l'operazione si effettuano system-call ripetute finché l'operazione non può essere completata. Questa modalità con NIO è possibile utilizzando `SocketChannel` e `ServerSocketChannels`

8.2 SocketChannel e ServerSocketChannels

- **`SocketChannel`**, combinazione di un socket TCP ed un channel NIO bidirezionale che permette di scrivere e leggere da una socket TCP. Ogni `SocketChannel` è associato ad un oggetto `Socket` reperibile mediante il metodo `socket()` di `SocketChannel`
- **`ServerSocketChannels`**, resta in ascolto di richieste di connessione su una porta e con una operazione di `accept()` si accettano le richieste (non trasferisce dati). Ad ogni `ServerSocketChannels` è associato un oggetto `ServerSocket`. In modalità bloccante vi è un comportamento analogo a `ServerSocket` ma con interfaccia basata su buffer. In modalità non bloccante, il metodo `accept()` ritorna immediatamente il controllo al chiamante e può restituire null se non sono presenti richieste di connessione, altrimenti un oggetto di tipo `SocketChannel`

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

Entrambe le classi estendono `AbstractSelectableChannel` e da questa mutuano la capacità di passare dalla modalità bloccante a quella non bloccante. Ciclo di vita di un `SocketChannel`:

1. **creazione**, lo si può fare implicitamente dopo che una connessione viene accettata su un `ServerSocketChannels` oppure esplicitamente quando si apre una connessione verso un server

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

2. **lettura da un `SocketChannel`**, in modalità non bloccante ritorna immediatamente e restituisce il numero di byte letti (anche 0)

```
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = socketChannel.read(buf);
```

   dove `bytesRead` dice quanti bit sono stati letti. Se viene restituito -1, è stata raggiunta la fine dello stream e la connessione viene chiusa. In modalità non bloccante può restituire 0 e può essere necessario ripetere la lettura

3. **scrittura in un `SocketChannel`**, in modalità non bloccante tenta di scrivere i dati nella socket, ritorna immediatamente, anche se i dati non sono stati completamente scritti

```
String str = "Hello";
ByteBuffer buf = ByteBuffer.allocate(64);
buf.put(str.getBytes());
buf.flip();
while(buf.hasRemaining())
    socketChannel.write(buf);
```

   il metodo `write()` è richiamato dentro un ciclo: in modalità non bloccante non ci sono garanzie di quanti byte sono scritti nel channel. Si ripete quindi `write()` finché nel buffer non ci sono più byte da scrivere

4. **chiusura**, tramite l'istruzione `socketChannel.close()`

Se il canale è in modalità bloccante, il metodo `connect()` si blocca fino a quando la connessione non viene completata o non può essere stabilità. In un canale non bloccante invece `connect()` può restituire il controllo al chiamante prima che venga stabilita la connessione. Il metodo `finishconnect()` server per controllare la terminazione delle operazioni: se la connessione non è ancora stabilita restituisce false, altrimenti true

```
SocketAddress address = new InetSocketAddress("127.0.0.1", 5000);
SocketChannel socketChannel = SocketChannel.open();
socketChannel.configureBlocking(false);
socketChannel.connect(address);
while(!socketChannel.finishConnect()){
    System.out.println("Connessione non terminata");
}
System.out.println("Terminata la fase di instaurazione della connessione");
```

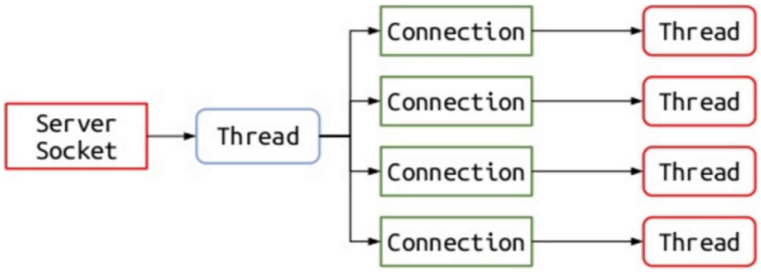
8.3 Server models

Criteri per la valutazione delle prestazioni di un server:

- **Scalability**, capacità di servire un alto numero di clienti 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 relatica risposta
- **Efficiency**, utilizzo delle risorse utilizzare sul server (RAM, numero di threads, utilizzo della CPU)

Models			
	un singolo thread	**un thread per ogni connessione**	**numero fisso di thread**
scalability	nulla: in ogni istante, solo un client viene servito	possibilità di servire diversi client in maniera concorrente fino al massimo numero di thread previsti per ogni processo. Ogni thread alloca il proprio stack: memory pressure	limitata al numero di connessioni che possono essere supportate
acceptance latency	alta: il prossimo cliente deve attendere fino a che il primo cliente termina la connessione	tempo tra l'accettazione di una connessione e la successiva è in genere basso rispetto a quello di interarrivo delle richieste	evita crash nel caso di alto numero di connessioni contemporanee
reply latency	tutte le risorse a disposizione di un singolo client	bassa: le risorse del server condivise tra connessioni diverse	bassa: fino al numero massimo di thread fissato, degrada se il numero di connessioni è maggiore
efficiency	buona: il server utilizza esattamente le risorse necessarie per il servizio dell'utente	bassa: ogni thread può essere bloccato in attesa di IO, ma utilizza risorse come la memoria	trade-off rispetto al modello precedente

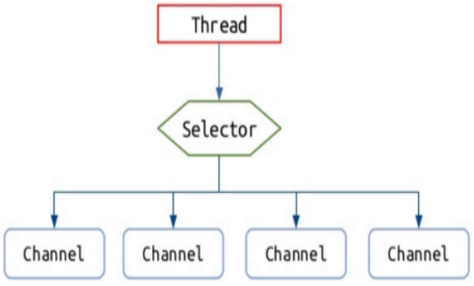
- *un singolo thread*, può essere adatto quando il tempo di servizio di un singolo utente è garantito rimanere basso
- *un thread per ogni connessione*, può avere problemi di scalabilità: il tempo per il cambio di contesto può aumentare notevolmente con il numero di thread attivi



- *numero fisso di thread: thread pool*, modo migliore per gestire le connessioni

8.4 Multiplexed I/O

Una delle funzionalità più importanti di NIO è il comportamento non-blocking per le varie operazioni IO. Usando le funzionalità non-blocking, il programma deve ripetere le operazioni finché queste vengono completate. Quindi se ci sono più operazioni da eseguire allora si deve iterare tra tutti i channels e questo non è una soluzione efficiente. Il modello introdotto da NIO con il **multiplexing** è quello di utilizzare un solo thread che usando un oggetto di tipo **selector** possa esaminare uno o più canali e determina quali di essi sono pronti per leggere/scrivere. A questo punto il programma può iterare tra i channel pronti ed eseguire le operazioni in modo non blocking.



I channel per essere usati con i selettori devono estendere la classe astratta `SelectableChannel`. Avere più connessioni di rete gestite mediante un unico thread, consente di ridurre l'uso di risorse per thread diversi e il tempo per il cambio di contesto. Lo svantaggio è avere una architettura più complessa da capire e da implementare. Il selector è supportato da due classi principali:

- `Selector`, fornisce la funzionalità di multiplexing
- `SelectionKey`, identifica i tipi di eventi pronti per essere elaborati

8.4.1 Selector: registrazione dei canali

L'istruzione `SelectableChannel configureBlocking(boolean block)` imposta il comportamento bloccante/non bloccante. Un channel deve diventare non bloccante prima di essere registrato con un selettore (`isBlocking()` verifica se un channel è bloccante). Quindi per utilizzare un selettore occorre:

- creare il selettore (`open()` crea un selettore mentre `isOpen()` verifica se il selettore è aperto)
- registrare i canali su quel selettore
- selezionare un canale quando è disponibile (si è verificato un evento su quel canale)

Il metodo **`register(Selector sel, int ops, Object att)`** applicata ad un canale non bloccante, aggiunge il canale alla lista di channel gestiti dal selettore `sel` e restituisce una `SelectionKey` che include un riferimento al channel. Se il canale era già registrato, la chiave iniziale dei selezione viene restituita dopo essere stata aggiornata con le nuove operazioni di interesse. Lo stesso canale può essere registrato con più selettori: la chiave identifica la particolare registrazione. L'operazione IO di interesse viene specificata dal parametro `ops`, vi sono 4 possibilità da combinare usando l'OR:

- `SelectionKey.OP_ACCEPT`, per server socket
- `SelectionKey.OP_CONNECT`, per client socket
- `SelectionKey.OP_READ`, per tutti i channel readable
- `SelectionKey.OP_WRITE`, per tutti i channel writeable

si può usare anche 0 se si vuole effettuare la registrazione senza operazioni di interesse che verranno aggiunte più tardi. Infine come ultimo elemento si può aggiungere un oggetto chiamato *attachment*. È uno spazio di memorizzazione, definito dal programmatore, in cui si possono memorizzare informazioni, una sorta di stato del channel. Ad esempio lo si può utilizzare per dare un custom ID al channel registrato o allegare il riferimento a un oggetto di cui vogliamo tenere traccia (esempio buffer)

8.4.2 SelectionKey

Una `SelectionKey` è un oggetto che memorizza il channel registrato ad un selettore e il suo stato. Sulla chiave di selezione si può verificare se il channel è pronto:

`boolean isAcceptable()`, verifica se il canale è pronto ad accettare una nuova connessione socket

`boolean isConnectable()`, verifica se il canale è pronto per una operazione di connect

`boolean isReadable()`, verifica se il canale è pronto per la lettura

`boolean isWritable()`, verifica se il canale è pronto per la scrittura

Altri metodi:

`SelectableChannel channel()`, restituisce il channel registrato. Il risultato può essere usato per eseguire l'operazione per cui è pronto

`Object attachment()`, restituisce l'attachment creato alla registrazione del channel

`void cancel()`, cancella la registrazione

Da una `SelectionKey` si possono recuperare due informazioni:

- **interest set**, indica per quali operazioni del canale si è registrato un interesse. Viene definito in fase di registrazione del canale con il selector ed è modificabile successivamente invocando il metodo `interestOps`. Il set può essere recuperato con l'istruzione `int interestSet = selectionKey.interestOps()`
- **ready set** indica quali operazioni sono pronte sul canale. Viene inizializzato a 0 quando la chiave viene creata e successivamente aggiornato quando si esegue una `select()`. Il set può essere recuperato dal metodo `readyOps()`

8.4.3 Multiplexing dei canali

Dopo aver creato il selettore e registrato i canali, il selector ha il compito di rimanere in ascolto sui canali per verificare se sono pronti. Per fare ciò si utilizza il metodo `select()`:

- è bloccante, si blocca finché almeno un channel è pronto
- seleziona i canali pronti per almeno una delle operazioni di I/O tra quelli registrati con quel selettore
- restituisce il numero di canali pronti (diverso da 0)
- costruisce un insieme contenente le chiavi dei canali pronti
- il thread che esegue la selezione può essere interrotto e il selettore viene sbloccato mediante il metodo `wakeup()` invocato da un altro thread

Altrimenti si può invocare `select(long timeout)` dove si blocca fino a che non è il trascorso il timeout o `selectNow()` non si blocca e nel caso nessun canale sia pronto restituisce il valore 0. Ogni oggetto selettore mantiene i seguenti insieme di chiavi:

- **key set**, `SelectionKeys` dei canali registrati con quel selettore. Restituito dal metodo `keys()`
- **selected key set**, `SelectionKeys` dei canali identificati come pronti. Restituito dal metodo `selectedKeys()`
- **cancelled key set**, contiene le chiavi invalidate, quelle su cui è stato invocato il metodo `cancel()`, ma non ancora deregistrate

Quando si invoca `select()`, il selector prende in carico il set di chiavi cancellate e rimuove effettivamente le chiavi, poi verifica quali canali sono pronti. Per ogni canale con almeno una operazione pronta nel suo interest set:

- se la chiave corrispondente non appartiene al selected key set allora la chiave viene inserita nel set. Il ready set di quella chiave viene resettato ed impostato con le chiavi corrispondenti alle operazioni pronte
- altrimenti il ready set viene aggiornato calcolando l'OR bit a bit con il valore precedente del ready set. Un bit settato non viene mai resettato, ma i bit ad 1 si "accumulano" man mano che le operazioni diventano pronte

A questo punto si itera sull'insieme di chiavi che individuano i canali pronti, per ogni chiave si individua il tipo di operazione per cui il canale è pronto e si effettua l'operazione (lettura, scrittura, ...). Quando il programma ha preso in carico gli eventi, li si cancella dal selected key set: si invoca `keyIterator.remove`. Il seguente programma apre una listening socket su una porta e quando arriva una connessione, il server l'accetta, trasferisce al client un messaggio e poi chiude la connessione

```
public class Server {
    static int port = 6789;
    public static void main(String[] args) {
        // inizializza il serverSocket e apri il selettore
        try(ServerSocketChannel serverSocket = ServerSocketChannel.open();
            Selector sel = Selector.open()){
            serverSocket.bind(new InetSocketAddress("localhost", port));
            serverSocket.configureBlocking(false);
            serverSocket.register(sel, SelectionKey.OP_ACCEPT);
            System.out.println("Server pronto alla porta: " +port);
```

```
            while (true){
                if(sel.select() == 0) continue;
                // insieme delle chiavi corrispondenti a canali pronti
                Set<SelectionKey> selectedKeys = sel.selectedKeys();
                // iteratore dell'insieme definito sopra
                Iterator <SelectionKey> iter = selectedKeys.iterator();

                while (iter.hasNext()){
                    SelectionKey key = iter.next();
                    if(key.isAcceptable()){
                        // accetta una connessione creando un SocketChannel
                        // per la comunicazione con il client che la richiede
                        SocketChannel client = serverSocket.accept();
                        client.configureBlocking(false);
                        ByteBuffer buffer
                                = ByteBuffer.wrap("Hello Client!\n".getBytes());
                        client.register(sel, SelectionKey.OP_WRITE, buffer);
                    }
                    if(key.isWritable()){
                        SocketChannel client = (SocketChannel) key.channel();
                        ByteBuffer buffer = (ByteBuffer) key.attachment();
                        client.write(buffer);
                        client.close();
                        key.cancel();
                    }

                    iter.remove();
                }
            }
        }
        catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

8.5 UDP Channels

La classe `DatagramChannel` supporta operazioni UDP non bloccanti: i metodi ritornano il controllo al chiamante se la rete non è immediatamente "pronta" a ricevere o a spedire dati (poiché UDP è più asincrono di TCP, l'effetto è minore). Anche i `DatagramChannel` possono essere registrati su un selector, chiaramente in UDP le uniche operazioni che hanno senso registrare sono `OP_READ` e `OP_WRITE`

```
ByteBuffer buffer = ByteBuffer.allocate(8192);
DatagramChannel channel = DatagramChannel.open();
channel.configureBlocking(false) // modalita' non bloccante
channel.socket().bind(new InetSocketAddress(9999));
SocketAddress address = new InetSocketAddress("localhost", 7); // destinatario
buffer.put(1); // dati da inviare
buffer.flip();
while(channel.send(buffer, address) == 0);
    // do something useful...
buffer.clear();
while(address = channel.receive(buffer)) == null);
    // do something useful...
```

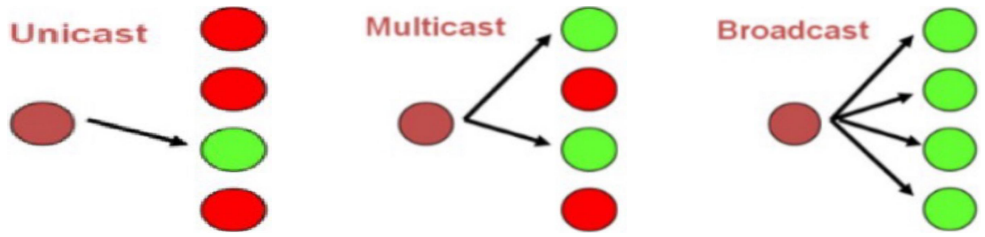
Se il metodo `send()` restituisce 0 significa che il buffer non ha sufficiente spazio. Se il metodo `receive()` restituisce null, allora non è stato ricevuto alcun datagram. Il programma può eseguire altre operazioni, nell'attesa che il canale sia disponibile per l'operazione

Chapter 9

Multicasting, URL e URLConnection

9.1 Multicasting

Per **multicasting** si intende inviare dati da un host ad un insieme di altri nodi. La maggior parte del lavoro viene svolto dai router ed è trasparente al programmatore, ovvero, i router assicurano che il pacchetto spedito dal mittente sia consegnato a tutti gli host interessati. Utilizza il *time-to-live* del livello IP. Il problema maggiore è che non tutti i router supportano il multicast. Inoltre il multicast utilizza solo UDP, non esiste il TCP multicast.



Il multicast è basato sul concetto di gruppo: insieme di processi in esecuzione su host diversi. Tutti i membri di un gruppo di multicast ricevono un messaggio spedito su quel gruppo, mentre, non occorre essere membri del gruppo per inviare i messaggi su di esso. Una API multicast deve contenere delle primitive per:

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

Invece il supporto IP deve fornire: uno schema di indirizzamento per identificare univocamente un gruppo; un meccanismo che registri la corrispondenza tra un gruppo ed i suoi partecipanti e un meccanismo che ottimizzi l'uso della rete nel caso di invio di pacchetti ad un gruppo di multicast

9.1.1 Schema di indirizzamento

Lo schema di indirizzamento multicast è basato sull'idea di riservare un insieme di indirizzi IP per il multicast. In IPv4 l'indirizzo di un gruppo è un indirizzo in classe D, mentre, in IPv6 tutti gli indirizzi di multicast iniziano con FF. Come scegliere un indirizzo di multicast? Un indirizzo multicast deve essere noto "collettivamente" a tutti i partecipanti del gruppo: è una procedura complessa. Gli indirizzi possono essere assegnati in modo:

- **statico**, assegnati da una autorità di controllo (IANA, amministratori di rete o di una certa organizzazione), utilizzati da particolari protocolli/applicazioni. L'indirizzo rimane assegnato a quel gruppo, anche se, in un certo istante non ci sono partecipanti
- **dinamico**, si utilizzano protocolli particolari che consentono di evitare che lo stesso indirizzo di multicast sia assegnato a due gruppi diversi. Esistono solo fino al momento in cui esiste almeno un partecipante

Per limitare la diffusione di un pacchetto si può utilizzare il time-to-live: indica il numero massimo di router attraversati dal pacchetto (valori nell'intervallo 0-255), dopo il pacchetto viene scartato. I valori 1, 16, 63 e 127 limitano il pacchetto, rispettivamente, alla sottorete locale, alla rete dell'organizzazione di cui fa parte l'host, la rete regionale e la rete globale

```
public class IndirizziIP {
    public static void main(String args[]) throws Exception {
        InetAddress address = InetAddress.getByName(args[0]);
        if(address.isMulticastAddress()){
            if(address.isMCGlobal())
                System.out.println("e' un indirizzo multicast globale");
            else if(address.isMCOrgLocal())
                System.out.println("e' un indirizzo organization local");
            else if(address.isMCSiteLocal())
                System.out.println("e' un indirizzo site local");
            else if(address.isMCLinkLocal())
                System.out.println("e' un indirizzo link local");
        }
        else {
            System.out.println("non e' un indirizzo di multicast");
        }
    }
}
```

9.1.2 Multicast: connection-less

Multicast utilizza il paradigma *connection-less* perché è adatta per il tipo di applicazioni verso cui è orientato il multicast (trasmissioni di dati video/audio). Una *connection-oriented* richiederebbe la gestione di un alto numero di connessioni ($n \cdot (n-1)$ connessioni per un gruppo di n applicazioni). Per esempio nell'invio dei frame di una animazione, è più accettabile la perdita occasionale di un frame piuttosto che un ritardo tra la spedizione di due frame successivi. Si perde l'affidabilità della trasmissione, ma esistono librerie Java non standard che forniscono multicast affidabile: garantiscono che il messaggio venga recapitato una sola volta a tutti i processi del gruppo e altre proprietà relative all'ordinamento dei messaggi

9.1.3 MulticastSocket

La classe `MulticastSocket` estende `DatagramSocket` e ne effettua l'overriding di alcuni metodi. Inoltre fornisce nuovi metodi per l'implementazione di funzionalità tipiche del multicast. Per unirsi ad un gruppo multicast si utilizza il metodo `joinGroup()` che lega il multicast socket ad un gruppo di multicast: tutti i messaggi ricevuti tramite quel socket provengono da quel gruppo. Viene sollevata una `IOException` se l'indirizzo di multicast è errato. Per ricevere pacchetti si utilizza `receive()`

```
public class ProvaMulticast {
    public static void main (String [] args) throws Exception {
        int port = 6789;
        byte [] buf = new byte[10];
        InetAddress ia = InetAddress.getByName("228.5.6.7");
        DatagramPacket dp = new DatagramPacket(buf, buf.lenght);
        MulticastSocket ms = new MulticatSocket(port);
        ms.joinGroup(ia);
        ms.receive(dp);
    }
}
```

Settando la proprietà `ms.setReuseAddress()` a true, è possibile associare più socket alla stessa porta, quindi è possibile attivare due istanze di `ProvaMulticast` senza sollevare una `BindException`. L'eccezione verrebbe sollevata se si utilizzasse un `DatagramSocket`. Per spedire messaggi ad un gruppo di multicast: si crea un `DatagramSocket` su una porta anonima (non è necessario collegare il socket ad un gruppo di multicast), si crea un pacchetto inserendo nell'intestazione l'indirizzo del gruppo di multicast a cui si vuole inviare il pacchetto e infine si utilizza il metodo `send()`

```
public class ProvaMulticast {
    public static void main (String [] args) throws Exception {
        int port = 6789;
        byte [] data = "hello".getBytes();
        InetAddress ia = InetAddress.getByName("228.5.6.7");
        DatagramPacket dp = new DatagramPacket(data, data.lenght, ia, port);
        DatagramSocket ms = new DatagramSocket(6800);
        ms.send(dp);
        Thread.sleep(8000);
    }
}
```

Si può impostare il TTL con l'istruzione `ms.setTimeToLive()`

9.2 Java URL

URL specifica la locazione di una risorsa su internet e come reperirla. La classe Java `URL` fornisce un'astrazione di una URL. Il seguente esempio mostra come creare una serie di URL:

```
public class JavaNetURLExample {
    publi static void main(String [] args){
        try{
            URL url1 = new URL("http://www.gnu.org");
            // stampa URL1: http://www.gnu.org
            System.out.println("URL1: " + url1.toString());

            URL url2 = new URL(url1, "licenses/gpl.txt");
            // stampa URL2: http://www.gnu.org/licenses/gpl.txt
            System.out.println("URL2: " + url2.toString());

            URL url3 = new URL("http", "www.gnu.org", "/licenses/gpl.txt");
            // stampa URL3: http://www.gnu.org/licenses/gpl.txt
            System.out.println("URL3: " + url3.toString());
        }
    }
}
```

Se si deve solamente scaricare il contenuto della risorsa identificata dalla URL, è possibile usare il metodo `openStream()` della classe `URL`: restituisce un `InputStream`. Il programma che segue invia una richiesta HTTP GET e poi la risposta è disponibile nello stream

```
InputStream uin = url.openStream();
BufferedReader in = new BufferedReader(new InputStreamReader(uin));
String line;
while((line = in.readLine()) != null){
    // process line
}
```

9.3 URLConnection

Se sono richieste ulteriori informazioni su una risorsa, oppure, si vuole modificare una risorsa e crearne una nuova occorre usare la classe `URLConnection`. Con il metodo `openConnection` si ottiene un oggetto `URLConnection`. Per connettersi alla risorsa remota invece si invoca `connect()`. È possibile impostare alla connessione delle proprietà come `setDoInput, setDoOutput, setIfModifiedSince`...

```
try {
    URL u = new URL("http://www.bogus.com");
    URLConnection uc = uc.openConnection();
    // per ricevere informazioni che descrivono il contenuto
```

```
    InputStream raw = u.getInputStream();
    // read from URL...
}
catch(MalformedURLException ex){
    System.err.println(ex);
}
catch(IOException ex){
    System.err.println(ex);
}
```

Metodi per reperire il valore degli header: `getHeaderFieldKey`, `getHeaderField`, `getContentType`, `getContentLenght`, `getContentEncoding`, ...

Chapter 10

RMI: Multithreading

10.1 Remote Procedure Call

Un'applicazione Java distribuita è composta da computazioni eseguite su JVM differenti, magari in esecuzione su host differenti comunicanti tra loro. Questo lo si è visto, a basso livello, utilizzando le *socket* scegliendo il paradigma di comunicazione TCP o UDP. Un'alternativa è utilizzare una tecnologia di più alto livello, originariamente indicata come **Remote Procedure Call** (RPC): interfaccia di comunicazione rappresentata dall'invocazione di una procedura remota, invece che dall'utilizzo diretto di una socket. Il paradigma di interazione è a *domande/risposta*:

- il client invoca una procedura del server remoto
- il server esegue la procedura con i parametri passati dal client e restituisce a quest'ultimo il risultato dell'esecuzione

Per esempio un client richiede ad un server la stampa di un messaggio e attende l'esito dell'operazione. In seguito il server restituisce un codice che indica l'esito della operazione. La connessione remota è trasparente rispetto a client e server, e in genere prevede meccanismi di affidabilità a livello sottostante. I meccanismi utilizzati dal client sono gli stessi utilizzati per una invocazione di una procedura locale ma:

- l'invocazione di una procedura avviene sull'host su cui è in esecuzione il client
- la procedura viene eseguita sull'host su cui è in esecuzione il server
- i parametri della procedura vengono inviati automaticamente sulla rete dal supporto

Il programmatore non deve più preoccuparsi di sviluppare protocolli che si occupino del trasferimento dei dati, della verifica, e della codica/decodifica. RPC è stato implementato con vari meccanismi e uno di questi è **Java RMI**

10.2 Java RMI

Un'applicazione **Remote Method Invocation** (RMI) è, in generale, composta da due programmi separati:

- *Server*, crea un oggetto remoto, pubblica un riferimento all'oggetto e attende che i client invochino metodi sull'oggetto remoto
- *Client*, ottiene un riferimento all'oggeto remoto e invoca i suoi metodi

L'obiettivo è permettere al programmatore di sviluppare applicazioni Java distribuite utilizzando la stessa sintassi e semantica utilizzate per i programmi non-distribuiti. Quindi una volta localizzato l'oggetto, il programmatore utilizza i metodi dell'oggetto come se questo fosse locale. La codifica, decodifica, verifica e trasmissione dei dati sono effettuati dal supporto RMI in maniera completamente trasparente all'utente. I problemi da gestire sono:

1. il client deve in qualche modo trovare un riferimento all'oggetto remoto (è diverso dal creare l'oggetto o dal farsi dare un riferimento passato come parametro)
2. l'oggetto che vuole rendere i suoi servizi invocabili deve esplicitamente dire che vuole fare ciò e deve rendersi reperibile

Un **oggetto remoto** è un oggetto i cui metodi possono essere acceduti da un diverso spazio di indirizzamento e dove è possibile sfruttare le caratteristiche della programmazione ad oggetti e tutte le funzionalità standard di Java (meccanismi di sicurezza, serializzazione dei dati...). Si può avere un **Java Security Manager** per controllare che le applicazioni distribuite abbiano i diritti necessari per essere eseguite

10.3 RMI: schema architetturale

L'architettura RMI prevede tre componenti:

- *Server*, esporta gli oggetti remoti
- *Client*, richiede i metodi degli oggetti remoti
- *Registry*, è un servizio di naming che agisce da "pagine gialle": registra i nomi e i riferimenti degli oggetti i cui metodi possono essere invocati da remoto. Tali oggetti devono essere registrati (`bind`) con un nome pubblico. Gli oggetti del client possono richiedere (`lookup`) oggetti registrati chiedendo, a partire dal nome pubblico, un riferimento all'oggetto

Le principali operazioni da effettuare rispetto all'uso di oggetto locale:

1. il server deve esportare gli oggetti remoti, il client deve individuare un riferimento all'oggetto remoto
2. il server registra gli oggetti remoti nel registry, tramite `bind`
3. i client cercano gli oggetti remoti, tramite la `lookup`, chiedendo a partire dal nome pubblico dell'oggetto, un riferimento all'oggetto remoto al registry
4. il client invoca il servizio mediante chiamate di metodi che sono le stesse delle invocazioni di metodi locali

Quindi l'invocazione dei metodi di un oggetto remoto, a *livello logico*, è identica all'invocazione di un metodo locale; a *livello di supporto* è gestita dal supporto RMI che provvede a trasformare i parametri della chiamata remota in dati da spedire sulla rete. Il network support provvede quindi all'invio vero e proprio dei dati sulla rete. Quando, lato client, si fa richiesta ad un componente remoto, si ha bisogno di conoscere l'interfaccia, ovvero conoscere i metodi offerti e come invocarli. Alcuni vincoli sui componenti:

1. *definizione* del comportamento, con
    - interfaccia deve estendere **`Remote`**
    - ogni metodo deve propagare **`RemoteException`**
2. *implementazione* comportamento, una classe che deve implementare l'interfaccia definita deve estendere la classe **`UnicastRemoteObject`**

10.3.1 Interfacce remote

`Remote` è implementata da due classi:

- *sul server*, la classe che implementa il servizio
- *sul client*, la classe che implementa il proxy del servizio remoto

Un'interfaccia è remota se e solo se estende `Remote` o un'altra interfaccia che la estende. `Remote` non definisce alcun metodo, il solo scopo è quello di identificare gli oggetti che possono essere utilizzati in remoto. I metodi definiti devono dichiarare di sollevare eccezioni remote della classe `RemoteException` in aggiunta a eccezioni specifiche della applicazione. L'implementazione della interfaccia fa uso delle seguenti classi:

- **`RemoteObject`**, fornisce l'implementazione per i metodi `hashCode`, `equals` e `toString`, implementandone il comportamento per un oggetto remote
- **`RemoteServer`**, super classe astratta comune per tutte le implementazioni di oggetti remoti (`UnicastRemoteObject` è una implementazione)
- **`UnicastRemoteObject`**, definisce un oggetto remoto i cui riferimenti sono validi solo mentre il processo server è attivo. Fornisce supporto per riferimenti ad oggetti (chiamate, parametri e risultati) utilizzando flussi TCP

10.3.2 Registry

Il registry è un server remoto in ascolto sulla porta indicata; è simile ad un DNS per oggetti remoti che contiene legami tra il nome simbolico dell'oggetto remoto e il riferimento all'oggetto. Metodi:

`LocateRegistry.createRegistry(int port)`, crea ed esporta un'istanza di registry

`LocateRegistry.getRegistry()`, restituisce un riferimento al registry. Può avere come parametri `int port, String host` o entrambi. Nel caso di nessun parametro si utilizza la porta di default 1099

Supponiamo che `r` sia l'istanza di un registro individuato mediante `getRegistry()`

`r.bind()`, crea un collegamento tra un nome simbolico ed il riferimento ad un oggetto remoto. Se esiste già un collegamento per lo stesso oggetto all'interno di `r`, viene sollevata una eccezione

`r.rebind()`, crea un collegamento tra un nome simbolico (qualsiasi) ed un riferimento all'oggetto. Se esiste già un collegamento per lo stesso oggetto all'interno dello stesso registry, tale collegamento viene sovrascritto. È possibile inserire più istanze dello stesso oggetto remoto nel registry, con nomi simbolici diversi

`r.list()`, restituisce un array di stringhe dei nomi dei server remoti presenti nel registro RMI

`r.lookup(String name)`, passa al registro RMI un nome di servizio remoto; riceve una copia serializzata dello stub associato a quel server remoto. Può restituire un'eccezione `NotBoundException` se il server non è presente nel registro

Gli ultimi due metodi sono utilizzati dal client per interrogare il registro RMI. Entrambi i metodi, dopo aver fatto un parsing del parametro, aprono una connessione socket verso il registro RMI

10.3.3 Stub

Lo stub è un oggetto che consente di interfacciarci con un altro oggetto (il target) in modo da sostituirsi ad esso. Struttura di un oggetto stub:

- **stato**, contiene essenzialmente tre informazioni: l'IP dell'host su cui è in esecuzione il servizio, la porta di ascolto, un identificativo RMI associato all'oggetto
- **metodi**, la classe `Stub` implementa la medesima interfaccia remota del server remoto

10.4 RMI: step by step

Si definisce un server che implementi, mediante un oggetto remoto, il seguente servizio: su richiesta del client, il server restituisce le principali informazioni relative ad un paese dell'Unione Europea di cui il client ha specificato il nome, linguaggio ufficiale, popolazione, nome della capitale

1. **Definizione dell'interfaccia remota**

```
public interface EUStats extends Remote {
    String getLanguages(String CountryName) throws RemoveException;
    int getPopolation(String CountryName) throws RemoteException;
    String getCapitalName(String CountryName) throws RemoteException;
}
```

2. **Implementazione del servizio remoto**, si utilizza una classe di appoggio per definire record che descrivono una singola nazione

```
public class EUData {
    private String language;
    private int population;
    private String capital;
    public EUData (String lang, int pop, String cap){
        language = lang;
        popolation = pop;
```

```
        capital = cap;
    }
    String getLangs(){ return language;}
    int getPop(){ return population;}
    String getCapital(){ return capital;}
}
```

implementazione dell'oggetto remoto: si utilizza una hash table per memorizzare i dati

```
public class EUStatsServiceImpl implements EUStats {
    // memorizza i dati in una hashtable
    Hashtable <String, EUData> EUDbase = new Hashtable <String, EUData>();
    public EUStatsServiceImpl() throws RemoteException {
        EUDbase.put("France", new EUData("French",  57800000, "Paris"));
        EUDbase.put("Italy", new EUData("Italian", 57998000, "Rome"));
        EUDbase.put("Greece", new EUData("Greek", 10270000, "Athens"));
        ...
    }
    public String getLanguages(String CountryName) throws RemoteException {
        EUData data = (EUData) EUDbase.get(CountryName);
        return data.getLangs();
    }
    public int getPopolation(String CountryName) throws RemoteException {
        EUData data = (EUData) EUDbase.get(CountryName);
        return data.getPop();
    }
    public String getCapitalName(String CountryName) throws RemoteException {
        EUData data = (EUData) EUDbase.get(CountryName);
        return data.getCapital();
    }
}
```

3. **Attivazione servizio**, il servizio viene creato allocando una istanza dell'oggetto remoto. Il servizio viene attivato mediante: creazione dell'oggetto remoto e registrazione dell'oggetto remoto in un registry

```
public class EUStatsServer {
    public static void main (String args[]) {
        try {
            // creazione di un'istanza dell'oggetto EUStatsServiceImpl
            EUStatsServiceImpl statsService = new EUStatsServiceImpl();
            // esportazione dell'oggetto
            EuStats stub = (EuStats)
                           UnicastRemoteObject.exportObject(statsService, 0);
            // creazione di un registry sulla porta args[0]
            LocateRegistry.createRegistry(args[0]);
            Registry r = LocateRegistry.getRegistry(args[0]);
            // pubblicazione dello stub nel registry
            r.rebind("EUSTATS-SERVER", stub);
            System.out.println("Server ready");
        }
        catch (RemoteException e) {
            System.out.println("Communication error " + e.toString());
        }
    }
}
```

nel main si sono effettuate le seguenti operazioni:

- si è creato un'istanza del servizio (oggetto remoto)
- si è invocato il metodo statico `UnicastRemoteObject.exportObject(obj, 0)` che esporta dinamicamente l'oggetto. Se si indica la porta 0 viene scelta una porta anonima. Il metodo restituisce il riferimento allo stub dell'oggetto remoto, che "rappresenta" il riferimento all'oggetto remoto

- si è pubblicato il riferimento all'oggetto remoto nel registry

4. **Client RMI**, per accedere all'oggetto remoto il client deve ricercare lo stub dell'oggetto remoto e poi accede al registry attivato sul server mediante il nome simbolico dell'oggetto remoto

```
public class EUStatsClient {
    public static void main (String args[]) {
        EUStats serverObject;
        Remote remoteObject;

        if (args.length < 2) {
            System.out.println("usage: java EUStatsClient port countryname");
            return;
        }

        try {
            Registry r = LocateRegistry.getRegistry(args[0]);
            remoteObject = r.lookup("EUSTATS-SERVER");
            serverObject = (EUStats) remoteObject;
            System.out.println("Main language(s) of " + args[1] + "
                is/are " + serverObject.getMainLanguages(args[1]));
            System.out.println("Population of " + args[1] + " is "
                + serverObject.getPopulation(args[1]));
            System.out.println("Capital of " + args[1] + " is "
                + serverObject.getCapitalName(args[1]));
        }
        catch (Exception e) {
            System.out.println("Error in invoking object method " +
                e.toString() + getMessage());
            e.printStackTrace();
        }
    }
}
```

In Java standard i valori di tipo primitivo sono passati per valori, mentre, gli oggetti sono passati per riferimento. In Java RMI:

- i valori di tipo primitivo sono passati *per valore*;
- gli oggetti locali sono passati *per valore*, utilizzando la serializzazione;
- gli oggetti remoti sono passati *per riferimento*, viene passato lo stub.

10.5 Meccanismo delle Callbacks

Il **meccanismo delle callbacks** consente di realizzare il pattern *Observer* in ambiente distribuito utile quando un client è interessato allo stato di un oggetto e vuole ricevere una notifica asincrona quando tale stato viene modificato. Soluzioni possibili sono:

1. **Polling**, il client interroga ripetutamente il server, per verificare l'occorrenza dell'evento atteso. Svantaggio: alto costo per l'uso non efficiente delle risorse di sistema
2. **Callback**, il client registra sul server il suo interesse per un evento e il server notifica tale evento, quando questo occorre, ad ogni client registrato. Vantaggio: il client non si blocca perché la notifica è asincrona e uso efficiente delle risorse. Noto anche come paradigma **publish-subscribe** e realizza il pattern Observer in ambiente distribuito

Si può utilizzare RMI per implementare un meccanismo di registrazione e notifica composto da: *interazione client-server* (registrazione dell'interesse per un evento) e interazione *server-client* (notifica del verificarsi di un evento). La notifica asincrona è implementata mediante due invocazioni remote asincrone

- *Server*, definisce un oggetto remoto (ROS) che implementa l'interfaccia **`ServerInterface`** e un metodo di registrazione per permettere al client di registrare il suo interesse per un certo evento. Quando riceve una invocazione del metodo remoto, memorizza lo stub del client (ROC) in una struttura dati. Quando occorre l'evento di cui il client si è registrato, il server utilizza ROC per invocare il metodo remoto sul client per la notifica. Il server riceve lo stub dell'oggetto remoto del client al momento della registrazione del client: non utilizza un registry
- *Client*, definisce un oggetto remoto ROC che implementa **`ClientInterface`** che definisce il metodo per ricevere una notifica. In seguito il client ricerca l'oggetto remoto del server ROS che contiene il metodo per la registrazione mediante un servizio di registry. Al momento della registrazione sul server, gli passa lo stub di ROC. Non registra l'oggetto remoto ROC in un registry

10.5.1 Callback: step by step

Esempio: un server gestisce le quotazioni di borsa di un titolo azionario. Ogni volta che si verifica una variazione del valore del titolo, vengono avvertiti tutti i client che si sono registrati per quell'evento. Il server definisce un oggetto remoto che fornisce metodi per consentire al client di registrare/cancellare una callback. Il client vuole essere informato quando si verifica una variazione, espone un metodo per la notifica, registra una callback presso il server, aspetta per un certo intervallo di tempo durante cui riceve le variazioni di quotazione. Alla fine cancella la registrazione della propria callback presso il server

- **interfaccia del server e la sua implementazione**

```
public interface ServerInterface extends Remote {
    // registrazione per la callback
    public void registerForCallback(NotifyEventInterface ClientInterface)
        throws RemoteException;
    // cancella registrazione per la callback
    public void unregisterForCallback (NotifyEventInterface ClientInterface)
        throws RemoteException;

}
```

```
public class ServerImpl extends RemoteServer implements ServerInterface{
    // lista dei client registrati
    private List <NotifyEventInterface> clients;
    // crea un nuovo servente
    public ServerImpl() throws RemoteException{
        super();
        clients = new ArrayList<NotifyEventInterface>();
    }
    public synchronized void registerForCallback(NotifyEventInterface
                                    ClientInterface) throws RemoteException {
        if(!clients.contains(ClientInterface)){
            clients.add(ClientInterface);
        }
        System.out.println("New client registered");
    }
    // annulla registrazione per il callback
    public synchronized void unregisterForCallback(NotifyEventInterface
                                    Client) throws RemoteException {
        if(clients.remove(Client))
            System.out.println("Client unregistered");
        else
            System.out.println("Unable to unregister client");
    }
    // notifica di una variazione di valore dell'azione
    // quando viene richiamato, fa il callback a tutti i client registrati
    public void update(int value) throws RemoteException {
        doCallbacks(value);
```

```
    }
    private synchronized void doCallbacks(int value) throws RemoteException {
        System.out.println("Starting callbacks");
        Iterator i = clients.iterator();
        while(i.hasNext()){
            NotifyEventInterface client = (NotifyEventInterface) i.next();
            client.notifyEvent(value);
        }
        System.out.println("Callbacks complete");
    }
}
```

- **interfaccia del client e la sua implementazione**

```
public interface NotifyEventInterface extends Remote {
    // metodo invocato dal server per notificare un evento
    // ad un client remoto
    public void notifyEvent(int value) throws RemoteException;
}
```

```
public class NotifyEventImpl extends RemoteObject implements
                                                    NotifyEventInterface {
    // crea un nuovo callback client
    public NotifyEventImpl() throws RemoteException{
        super();
    }
    // metodo che puo' essere richiamato dal server per notificare
    // una nuova quotazione del titolo
    public void notifyEvent(int value) throws RemoteException {
        String returnMessage = "Update event received: " + value;
        System.out.println(returnMessage); }
    }
}
```

- **Attivazione del server**

```
public class Server {
    public static void main(String [] args){
        try{
            // registrazione presso il registry
            ServerImpl server = new ServerImpl();
            ServerInterface stub = (ServerInterface)
                                UnicastRemoteObject.exportObject(server, 39000);
            String name = "Server";
            LocateRegistry.createRegistry(5000);
            Registry registry = LocateRegistry.getRegistry(5000);
            registry.bind(name, stub);
            while(true){
                int val = (int) (Math.random()*1000);
                System.out.println("nuovo update: " + val);
                server.update(val);
                Thread.sleep(1500);
            }
        }
        catch(Exception e){
            System.out.println("Server exception: " +e.getMessage);
        }
    }
}
```

- **Attivazione del client**

```
public class Client {
    public static void main (String []args){
        try{
            System.out.println("Cerco il server");
            Registry registry = LocateRegistry.getRegistry(5000);
            String name = "Server";
            ServerInterface server = (ServerInterface) registry.lookup(name);
            // si registra per la callback
            System.out.println("Registering for callback");
            NotifyEventInterface callbackObj = new NotifyEventImpl();
            NotifyEventInterface stub = (NotifyEventInterface)
                                        UnicastRemoteObject(callbackObj, 0);
            server.registerForCallback(stub);
            // attende gli eventi generati dal server
            // per un certo intervallo di tempo
            Thread.sleep(20000);
            // cancella la registrazione per la callback
            System.out.println("Unregistered for callback");
            server.unregisterForCallback(stub);
        }
        catch(Exception e){
            System.err.println("Client exception: " + e.getMessage());
        }
    }
}
```

10.6 RMI: concorrenza

Si consideri un'istanza di un oggetto remoto, se diversi client invocano concorrentemente i metodi di quell'oggetto, come vengono gestiste le invocazioni concorrenti? Dipende dall'implementazione di RMI: è possibile attivare un thread per ogni richiesta o un thread per ogni client. Invocazioni di metodi remoti provenienti da client diversi sono tipicamente eseguite da thread diversi. Questo consente di non bloccare un client in attesa della terminazione dell'esecuzione di un metodo invocato da un altro client e ottimizza la performance del servizio remoto. Anche le invocazioni concorrenti provenienti dallo stesso client possono essere eseguite dallo stesso thread o da thread diversi. Il server che istanza l'oggetto remoto in generale non è thread safe, quindi richieste concorrenti di client diversi possono portare la risorsa ad uno stato inconsistente. L'utente che sviluppa il server deve assicurare che l'accesso all'oggetto remoto sia correttamente sincronizzato