

Reti laboratorio

Lorenzo Gazzella

June 2019

Contents

1	Introduzione	3
2	Prima lezione	3
2.1	Come creare ed attivare un thread	3
2.2	L'interfaccia runnable	4
2.3	Il metodo <i>start()</i>	4
2.4	I thread demoni	4
2.5	Terminazione programmi concorrenti	5
2.6	La classe Thread	5
2.7	Thread pool	6
3	Seconda lezione e terza lezione	9
3.1	La classe Thread	9
3.2	Meccanismi di sincronizzazione	10
3.2.1	Lock	10
3.2.2	Monitor	11
3.3	Collezioni e sincronizzazione	13
3.3.1	Concurrent Hash Map	13
4	Quarta lezione	14
4.1	Input e output in Java	14
4.2	Java stream	14
4.3	Java serialization	15
5	Quinta lezione	17
5.1	Reader e Writer // todo aggiungere collegamento	17
5.2	JSON	18
5.3	Bufferizzazione	18
5.4	Java NIO	19
5.4.1	Obiettivi	19
5.4.2	Buffers	19
5.4.3	Channel	21

6	Lezione 6	23
6.1	Comunicazione tramite socket	23
6.2	Connection oriented	23
6.3	Connectionless	23
6.4	Indirizzo IP	23
6.4.1	La classe InetAddress	24
6.5	Network interface	24
6.6	Indirizzo di loopback	24
6.7	Il protocollo TCP in Java	25
6.7.1	Lato client	25
6.7.2	Lato server	25
6.7.3	La comunicazione tra server e client	26
6.8	Shot down output	26
7	Lezione 7	26
7.1	Java NIO	26
7.1.1	Obiettivo	26
7.2	Non blocking mode	27
7.2.1	SocketChannel	27
7.2.2	Selector	27
7.2.3	ServerSocketChannel	27
7.2.4	Write nei SocketChannel	28
7.2.5	Creazione di un SocketChannel	28
7.2.6	Read su un SocketChannel	28
7.2.7	Non blocking connect	29
7.3	Server models	29
7.3.1	Single thread model	30
7.3.2	One thread for connection	30
7.3.3	Numero fisso di thread	30
7.3.4	I/O Multiplexing (Selector)	31
7.4	Selector	31
7.4.1	Multiplexing dei canali	32
8	Lezione 8	34
8.1	TCP vs UDP	34
8.2	UDP in Java	34
8.2.1	DatagramSocket	35
8.2.2	DatagramPacket	36
8.3	Comunicazione UDP	36
9	Lezione 9	37
9.1	Obiettivi	37
9.2	Remote Procedure Call (RPC)	37
9.2.1	Limiti	37
9.3	Remote Method Invocation (RMI)	37
9.3.1	Implementazione di un servizio tramite RMI	38

9.3.2	Connessione a un servizio RMI	40
9.3.3	Lo stub	40
9.3.4	Il registry	41
10	Lezione 10	42
10.1	Paradigmi di comunicazione	42
10.2	Gruppi multicast	42
10.3	Multicast API	42
10.3.1	Mutlicast addressing	43
10.3.2	Mutlicast scoping	43
10.4	Java API per Multicast	43
10.4.1	La classe MulticastSocket	44
10.5	Il meccanismo delle callback	44
10.5.1	Callback via RMI	45
10.6	RMI e concorrenza	48
11	Lezione 11	48
11.1	Dynamic Class Loading	48
11.2	RMIClassLoader	48
11.3	Alcuni scenari possibili	49
11.3.1	Thin client	49
11.3.2	Caricamento dinamico degli stub	50
11.3.3	Polimorfismo	50
11.4	Code mobility e sicurezza	51
11.5	Il security manager	52
11.5.1	Attivazione del security manager	52
11.5.2	Il file di policy	52
12	Lezione 12	52
12.1	Representational State Transfer (REST)	52
12.2	Operazioni	54
12.3	REST in Java	54

1 Introduzione

Il documento rappresenta un riassunto delle slide del corso di reti laboratorio. Tutte le nozioni dei thread già fatte nel corso di sistemi operativi non sono state riportate.

2 Prima lezione

2.1 Come creare ed attivare un thread

Esistono due modi per creare un thread:

1.
 - Definire un task, ovvero definire una classe C che implementi l'interfaccia **Runnable**.
 - Creare un'istanza di C
 - Creare un thread passandogli l'istanza del task appena creato
 - Chiamare il metodo **start()** del thread
2.
 - Estendere la classe *java.lang.Thread*
 - Istanziare un oggetto della classe appena creata
 - Chiamare il metodo **start()** del thread

2.2 L'interfaccia Runnable

- Appartiene al package *java.lang*
- Contiene solo la segnatura del metodo **void run()**
- Ogni classe C che implementa l'interfaccia Runnable deve fornire un'implementazione del metodo **run()**
- **N.B.** Un oggetto istanza di C è un task. La creazione di un task non implica la creazione di un thread. Lo stesso task può essere eseguito più volte anche da thread diversi
 - *CurrentThread()* restituisce il riferimento al thread che sta eseguendo il segmento di codice all'interno del quale si trova la sua invocazione
- **N.B. 2** Per avviare un thread bisogna invocare il metodo **start()** del Thread e non il metodo **run()** dell'oggetto che estende *Runnable*. Chiamare il metodo *run()* non comporta alcuna creazione di Thread, ma solo una chiamata di funzione.

2.3 Il metodo *start()*

- Tramite la JVM, segnala allo schedatore che il thread può essere attivato. Il sistema operativo inizierà l'ambiente del thread
- Restituisce immediatamente il controllo al chiamante, senza attendere che il thread attivato inizi la sua esecuzione

2.4 I thread demoni

- I thread demoni hanno il compito di fornire un servizio in background fino a che il programma è in esecuzione
- Quando tutti i thread non demoni sono completati, il programma termina, anche se ci sono thread demoni in esecuzione

- Se ci sono thread non demoni ancora in esecuzione, il programma non termina
- **Oss:** Il thread *main()* è un thread non demone
- **Oss:** Per dire che un thread è demone bisogna invocare il metodo **setDaemon(true)** prima di mandare in esecuzione il thread.

2.5 Terminazione programmi concorrenti

- Un programma Java termina quando terminano tutti i thread non demoni che lo compongono
- Se il thread iniziale (*main()*) termina, i restanti thread ancora attivi continuano la loro esecuzione, fino alla loro terminazione.
- Se uno dei thread usa l'istruzione **System.exit()** per terminare l'esecuzione, allora tutti i thread terminano la loro esecuzione

2.6 La classe Thread

Contiene tutti i metodo per:

1. Costruire un thread interagendo con il sistema operativo ospite
2. Attivare, sospendere e interrompere i thread
3. Non contiene i metodo per la sincronizzazione tra i thread. La sincronizzazione opera su oggetti.

Alcuni metodi utili:

- **public static native void sleep(long m)**
 - Sospende l'esecuzione del thread per M millisecondi
 - Mentre è in sleep, il thread può essere interrotto. Nel caso viene generata l'eccezione **InterruptedException**
 - * Per interrompere un thread si usa il metodo **interrupt()**. Questo imposta a true un valore booleano nel descrittore del thread.
 - * Il flag vale true se e solo se esistono interrupts pendenti
 - * E' possibile testare il valore del flag mediante:
 - **public static boolean Interrupted()**
 - **public boolean isInterrupted()**
 - E' un metodo statico

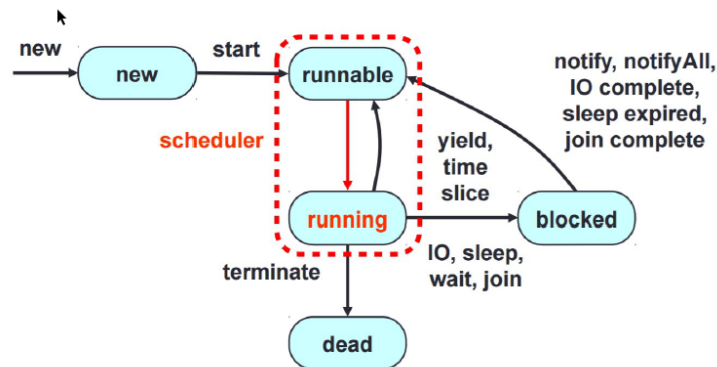


Figure 1: Ciclo di vita di un thread

2.7 Thread pool

Avere un task per ogni thread ha diversi svantaggi:

- Overhead per la creazione/distruzione dei thread
- Per richieste di servizio frequenti e "lightweight" può impattare negativamente sulle prestazioni dell'applicazione
- Molti thread in idel quando il loro numero supera il numero di processori disponibili
- Sia il garbage collector che lo schedulatore sono sotto stress
- C'è un limite al numero massimo di thread che si possono creare

Per ovviare a tutti questi svantaggi si introduce un thread pool:

- L'utente crea il pool e stabilisce una politica per la gestione dei thread del pool. Questa stabilisce quando i thread del pool vengono attivati (alla creazione, on demand, all'arrivo di un nuovo task) e come terminare l'esecuzione di un thread, se necessaria.
- Al momento della sottomissione di un task, il supporto può decidere se utilizzare un thread attivato in precedenza, se creare un nuovo thread, se memorizzare il task in una coda, se respingere la richiesta di esecuzione, etc.
- Il numero di thread nel pool può essere dinamico
- la libreria **java.util.concurrent** contiene i metodi per creare un pool ed il gestore associato. Permette di definire la struttura dati per memorizzare i task in attesa e di definire le politiche di gestione del pool

La classe **Executors** opera come una factory in grado di generare oggetti di tipo **ExecutorService**. Quando si vuole eseguire un task, basta passarlo ad un **Executor**, mediante l'invocazione del metodo **execute()**. Il task deve necessariamente estendere l'interfaccia *Runnable*. Esistono diversi tipi di **ExecutorService**:

- **NewCachedThreadPool**

- 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.
- Non esiste un limite al thread 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
- Oss: è una soluzione molto elastica poichè si espande all'infinito quando c'è parecchia richiesta e si ridimensiona se ce ne è poca

- **NewFixedThreadPool(n)**

- Vengono creati n thread al momento della inizializzazione del pool
- Quando viene sottomesso un task T:
 - * Se tutti i threads sono occupati nell'esecuzione di altri Task, T viene inserito in una coda
 - * La coda è illimitata
 - * Se almeno un thread è inattivo, viene utilizzato quel thread

- **ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, BlockingQueue<Runnable>workqueue)**

- Costruttore più generale, consente di personalizzare la politica di gestione del pool
- **CorePoolSize** è la dimensione minima del pool
- **MaxPoolSize** è la dimensione massima del pool
- Alla sottomissione di un nuovo task, funzionamento è il seguente:
 - * Se un thread del core è inattivo, il task viene assegnato ad esso
 - * Altrimenti se la coda passata come ultimo parametro del costruttore non è piena, il task viene inserito nella coda. I task vengono poi prelevati dalla coda ed inviati ai thread disponibili
 - * Altrimenti, se la coda è piena si crea un nuovo thread, attivando così k thread, con $corePoolSize < k < maxPoolSize$.
 - * Altrimenti il task viene respinto
- Eliminazione dei thread inutili: supponiamo che un thread termini l'esecuzione di un task e che il pool contenga k thread:

- * Se $k \leq \text{core}$: Il thread si mette in attesa di nuovi task da eseguire
- * Se $k > \text{core}$ ed il thread non appartiene al core, si considera il timeout T. Se nessun task viene sottomesso entro T, il thread termina la sua esecuzione

La scelta della coda da passare all'executor influisce sullo scheduling:

- **SynchronousQueue**: Dimensione uguale a 0. Ogni nuovo task o viene eseguito immediatamente oppure viene respinto. Viene eseguito immediatamente se esiste un thread inattivo, oppure se è possibile creare un nuovo thread
- **LinkedBlockingQueue**: Dimensione illimitata. E' sempre possibile accodare un nuovo task nel caso in cui tutti i thread siano attivi nell'esecuzione di altri task. La dimensione del pool non può superare core
- **ArrayBlockingQueue**: Dimensione limitata stabilita dal programmatore

Come si termina un ExecutorService visto che

- I task vengono eseguiti in modo asincrono rispetto alla loro sottomissione
- In un certo istante, alcuni task sottomessi precedentemente possono essere completati, alcuni in esecuzione e altri in coda.
- Un thread del pool può rimanere attivo anche quando ha terminato l'esecuzione di un task

Esistono due metodi per terminare un ExecutorService:

- In modo graduale (**shutdown()**): Finisci ciò che hai iniziato ma non iniziare nuovi task
 - * Nessun task viene accettato dopo che la **shutdown()** è stata invocata
 - * Tutti i task sottomessi in precedenza e non ancora terminati vengono eseguiti, compresi quelli la cui esecuzione non è ancora iniziata
 - * Successivamente tutti i thread del pool terminano la loro esecuzione
- In modo istantaneo (**shutdownNow()**): Stacca immediatamente la spina
 - * Non accetta ulteriori task ed elimina i task non ancora iniziati
 - * Restituisce una lista dei task che sono stati eliminati dalla coda
 - * Tenta di terminare l'esecuzione dei thread che stanno eseguendo i task. Questo viene fatto inviando una interruzione ai thread. Se un thread non risponde all'interruzione, il programma non termina

Per capire se l'esecuzione del pool è terminata:

- Attesa attiva: Invoco ripetutamente la **isTerminated()**
- Attesa passiva: Invoco la **awaitTermination()**

3 Seconda lezione e terza lezione

Nella prima lezione abbiamo visto come creare un thread. Nello specifico gli si passava un task (classe che implementa *Runnable*), si invocava il metodo **start()** del thread, che a sua volta chiama il metodo **run()** del thread, che a sua volta chiama il metodo **run()** del task.

Un altro metodo consiste nel creare una classe che estende **Thread** ed effettuare l'overriding del metodo **run()**. Quando si invocherà il metodo **start()** del thread, questa chiamerà a sua volta il metodo **run()** che abbiamo appena ridefinito. Data l'ereditarietà singola del Java, questo metodo a volte può essere limitante.

3.1 La classe Thread

La classe Thread contiene tutti i metodi per costruire un thread (interagendo con il sistema operativo ospite), impostare e reperire le caratteristiche di un thread, attivare, sospendere e interrompere un thread o attendere la terminazione. Non contiene invece tutti i metodi per la sincronizzazione tra thread. Questo perché la sincronizzazione viene fatta a livello di oggetti.

Un thread è definito da:

- ID: Identificatore univoco
- Nome: Nome del thread
- Priorità: Valore da 1 a 10, con 1 la priorità più bassa
- Nome gruppo: Gruppo a cui appartiene il thread
- Stato: new, runnable, blocked, waiting, time waiting, terminated
- Getter e setter

Per reperire il riferimento al thread corrente basta invocare il metodo **Thread.currentThread()**

Il metodo **join()** della classe Thread:

- Il thread che esegue la **join()** attende la terminazione dell'istanza del thread sul cui è stato chiamato
- Possibile specificare il timeout di attesa
- Il metodo **join()** può lanciare l'eccezione **InterruptedException** se il thread che ha chiamato la join riceve una interruzione

3.2 Meccanismi di sincronizzazione

3.2.1 Lock

Una lock in Java è:

- Un oggetto che può trovarsi in due stati diversi: *Locked* o *Unlocked*.
- Lo stato viene impostato con i metodi **lock()** e **unlock()**.
- Un solo thread può impostare lo stato a locked, gli altri che cercheranno di impostarlo si bloccheranno
- Quando il thread che detiene la lock, la rilascia, verrà risvegliato uno dei thread che si erano bloccati

Le lock forniscono mutua esclusione: Le sezioni critiche vanno eseguite con la lock

Per implementare una lock bisogna estendere l'interfaccia *Lock*. Un'implementazione già fornita è la classe **ReentrantLock**. L'interfaccia *Lock* mette a disposizione i seguenti metodi:

- *void lock()*
- *lockInterruptibly()*
- *boolean tryLock()*
- *tryLock(long time, TimeUnit unit)*
- *void unlock()*
- *Condition newCondition()*

Ovviamente, come visto nel corso di sistemi operativi, si può incorrere in deadlock.

Oss: Il tempo di esecuzione del programma senza uso di lock è circa la metà di quello con uso di lock. Usarle quindi con oculatezza.

Oss: La classe *ReentrantLock* permette di acquisire più volte la lock su uno stesso oggetto da parte dello stesso thread in modo da non andare in deadlock da solo.

Un'altra interfaccia utile è *ReadWriteLock*.

- Questa permette di acquisire più lock in lettura, purchè non vi siano scrittori
- La lock in scrittura è invece esclusiva

Una sua implementazione è la classe *ReentrantReadWriteLock()* Per evitare attesa attiva di una condizione, si introducono le *variabili di condizione*:

- Ad una lock possono essere associate un insieme di variabili di condizione
- L'interfaccia **Condition** fornisce i seguenti metodo:
 - **void await()**
 - **boolean await(long time, TimeUnit unit)**
 - **long awaitNanos(long nanoTimeout)**
 - **awaitUninterruptibly()**
 - **boolean awaitUntil(Date deadline)**
 - **void signal()**
 - **void signalAll()**
- Per ottenere una condizione su una lock si deve invocare il metodo **newCondition** su una lock

3.2.2 Monitor

- Java associa a ciascun oggetto una lock implicita ed una coda associata a tale lock
- Per acquisire tale lock si utilizzano i metodi o i blocchi di codice **synchronized**
- Quando un metodo con **synchronized** viene invocato, se nessun metodo **synchronized** della classe è in esecuzione, viene acquisita la lock sull'oggetto ed il metodo viene eseguito. Altrimenti se la lock su quell'oggetto è già stata presa, il thread viene sospeso nella coda associata all'oggetto fino a che il thread che detiene la lock non la rilascia
- La lock è associata ad una istanza dell'oggetto, non alla classe. Metodi su istanze diverse della stessa classe possono essere eseguiti in modo concorrente
- Esempio:

```
synchronized(obj){
    // Java code
}
```

Un thread acquisisce la lock sull'oggetto **obj** quando entra nel blocco sincronizzato e la rilascia quando lascia il blocco sincronizzato. Un thread alla volta esegue il blocco di codice

- Se **synchronized** lo si mette nella firma di un metodo equivale a prendere la lock sull'intero oggetto (**this**) per tutta la durata del metodo
- Ci sono due code gestire in modo implicito:

- **Entry set:** Contiene i thread in attesa di acquisire la lock
- **Wait set:** Contiene i thread che hanno eseguito una *wait* e sono in attesa di una *notify*
- Ogni Object ha i metodi:
 - **void wait():** Sospende il thread in attesa che sia verificata una condizione. Inoltre rilascia la lock sull'oggetto. In seguito ad una notifica, può riacquisire la lock
 - **void wait(long timeout):** Sospende per al massimo timeout millisecondi
 - **void notify():** Notifica ad un thread in attesa il verificarsi di una certa condizione
 - **void notifyAll():** Notifica a tutti i thread in attesa il verificarsi di una condizione

Per invocare questi metodi occorre aver acquisito la lock sull'oggetto su cui sono invocati. Devono quindi essere invocati all'interno di un blocco/metodo sincronizzato

- Poiché esiste una unica coda implicita in cui vengono accodati i thread in attesa, non è possibile distinguere thread in attesa di condizioni diverse. Quindi ogni thread deve ricontrollare se la condizione è verificata
- Testare sempre le condizioni all'interno di un ciclo
- Lock implicite - vantaggi:
 - Imposta una disciplina di programmazione
 - Evita errori dovuti a complessità del programma concorrente
 - Maggior robustezza
- Lock implicite - svantaggi:
 - Minore flessibilità
- Lock esplicite - vantaggi:
 - Ci sono un numero maggiore di funzioni disponibili, quindi maggiore flessibilità
 - Shared lock: ReadWriteLock
 - Migliori performance

3.3 Collezioni e sincronizzazione

Si possono distinguere tre tipi di collezioni:

- Collezioni che non offrono alcun supporto per il multithreading (Es: **ArrayList**)
- Synchronized collections (Es: **Vector**): Le operazioni individuali sulle collezioni sono safe, ma funzioni composte da più di una operazione singola possono non essere safe. La soluzione è sincronizzare l'intera collezione.
- Concurrent collections (*java.util.concurrent*): Garantiscono un supporto più sofisticato per la sincronizzazione.
 - Permettono l'overlapping di operazioni sugli elementi della struttura
 - Miglioramento di performance
 - Semantica leggermente modificata
 - La struttura non può essere modificata mentre viene scorsa mediante un iteratore

La classe **Collections** con la *s* contiene metodi statici per l'elaborazione delle collezioni. Ad esempio il metodo **synchronizedList()** produce un nuovo oggetto **List** che memorizza l'argomento in un campo privato. La lista ora sarà thread-safe ma la lock verrà presa su tutta la collezioni, quindi ci sarà un degrado di prestazione rispetto a sceglierne una thread-safe fin dall'inizio.

3.3.1 Concurrent Hash Map

- Invece di una sola lock per tutta la struttura, la tabella viene divisa in sezioni.
- In questo modo sono possibili più write simultanee se modificano diverse sezioni della tabella
- Aumento del parallelismo e della scalabilità
- Impossibile effettuare la lock su tutta la struttura. Non si può utilizzare quindi **synchronized**
- Approssimazione della semantica di alcune operazioni: **size()**, **isEmpty()**
- Sono definite delle nuove operazioni atomiche come: **removeIfEqual**, **replaceIfEqual**, **putIfAbsent**

4 Quarta lezione

4.1 Input e output in Java

La sorgente esterna può essere il file system, la connessione di rete, buffer in memoria o tastiera. Java I/O definisce un insieme di astrazioni per la gestione dell'IO.

- **java.io.file** Un'istanza della classe File describe:
 - Path per l'individuazione del file o della directory
 - Mette a disposizione metodi per verificare l'esistenza del path, restituire meta-informazioni sul file

4.2 Java stream

- Uno stream è un'astrazione che rappresenta una connessione tra un programma Java ed un dispositivo esterno (file, buffer di memoria, ...)
- Accesso sequenziale
- Mantengono l'ordinamento FIFO
- One way: read only oppure write only
- Sono bloccanti: Quando un'applicazione legge un dato dallo stream (o lo scrive) si blocca finché l'operazione non è completata
- Ci sono più di 60 tipi di stream diversi, tutti basati su 4 classi astratte fondamentali:
 - **Input/OutputStream:**
 - * Leggono e scrivono valori di 8 bits
 - * Utili per leggere/scrivere row data in binario
 - * Dati copiati byte a byte senza effettuare alcuna traduzione
 - * Ideale per leggere/scrivere file binari
 - * Alcune implementazioni sono System.out e System.in, FileInputStream/FileOutputStream, ByteArrayOutputStream/ByteArrayInputStream
 - * Metodi a disposizione: **int read(), int read(byte[] bytes, int offset, int length), public int read(byte[] bytes), close()**
 - **Readers/Writers:** Leggono e scrivono caratteri Unicode di 16 bit
- Try with resources:

```
try(FileInputStream in = new FileInputStream(new File("a.jpg"))){  
    // Java code  
}
```

Si occupa di chiudere automaticamente le risorse aperte. Possono comunque essere inseriti blocchi `catch` e `finally` che vengono comunque eseguiti dopo la chiusura della risorsa.

- Filter stream:
 - Concatenare gli stream di base condefli stream filtro
 - Classi **FilterInputStream** e **FilterOutputStream**
 - **BufferedInputStream** e **BufferedOutputStream** implementano una bufferizzazione per stream di input e di output. I dati vengono scritti e letti a blocchi di bytes invece che un solo blocco alla volta

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

- Formmatted data stream:

```
try (DataInputStream in = new DataInputStream(new
    BufferedInputStream(new FileInputStream(filename))))
```

Ora su `in` ci si possono chiamate tutte le funzioni come `in.readByte()`, `in.readInt()`, `in.readLong()`, `in.readFloat()`, etc

4.3 Java serialization

- Ogni oggetto è caratterizzato da uno stato (i campi) e da un comportamento (i metodi)
- Il salvataggio dell'oggetto riguarda il suo stato e può avvenire in diversi modi:
 - Accedendo ai singoli campi dell'oggetto, salvando i valori su un text file
 - Trasformando automaticamente l'oggetto o un grafo di oggetti in uno stream di byte, mediante il meccanismo della serializzazione
- La serializzazione è utilizzata per fornire un meccanismo di persistenza ai programmi, fornire un meccanismo di interoperabilità mediante oggetti condivisi tra diverse applicazioni, fornire un meccanismo per inviare oggetti
- Per rendere un oggetto persistente, l'oggetto deve implementare l'interfaccia *Serializable*. Gli oggetti primitivi sono tutti serializzabili.
- L'interfaccia *Externizable* consente di creare un proprio protocollo di serializzazione

- Gli oggetti che contengono riferimenti specifici con la JVM o al SO non possono estendere *Serializable*
- Le variabili marcate come **transient** non verranno serializzate
- Le variabili statiche non sono serializzabili poiché sono di classe
- Per serializzare un oggetto:

```
PersistentTime time = new PersistentTime();
FileOutputStream fos = new FileOutputStream(filename);
ObjectOutputStream out = new ObjectOutputStream(fos);
out.writeObject(time);
```

- Per deserializzare un oggetto:

```
PersistentTime = null;
FileInputStream fis = new FileInputStream(filename);
ObjectInputStream in = new ObjectInputStream(fis);
time = (PersistentTime)in.readObject();
```

Può sollevare l'eccezione **ClassNotFoundException**

- La serializzazione è transitiva, ovvero verranno serializzati anche tutti gli oggetti contenuti dall'oggetto che sto serializzando
- Se uno degli oggetti riferiti non è serializzabile viene sollevata l'eccezione **NotSerializableException**. Nel caso bisogna marcare tale oggetto come **transient**.
- Caching
 - Ogn qual volta un oggetto viene serializzato e inviato ad un **ObjectOutputStream**, la sua identità viene memorizzata in una *identity hash table*
 - Se l'oggetto viene scritto nuovamente sull'**OutputStream**, non viene nuovamente serializzato ma viene memorizzato un puntatore all'oggetto precedente. In questo modo si minimizzano le scritture
 - Stessa cosa per la lettura
- Controllo delle versioni
 - Per deserializzare un oggetto occorre conoscere i byte che rappresentano l'oggetto e il codice della classe che descrive la specifica dell'oggetto
 - La deserializzazione può avvenire in un ambiente diverso (compilatore diverso o a distanza di tempo)

- Quando un oggetto viene serializzato, gli si associa un **serialVersionUID (SUID)**. Consiste in un hash 64-bit generato a partire dalla struttura della classe, dal nome della classe, dai nomi dell'interfaccia e dai metodi e campi
- Il (SUID) identifica univocamente la classe utilizzata per la serializzazione. Modificando il codice della classe si modifica il *SUID*
- In fase di deserializzazione si calcola il *SUID* della classe locale e si confronta con quello dell'oggetto memorizzato. Se i due valori corrispondono si inizia la deserializzazione, altrimenti viene lanciata l'eccezione **java.io.InvalidClassException**.
- Ci sono alcune modifiche alla classe compatibili, ovvero che rendono comunque possibile la deserializzazione. Queste modifiche consistono nell'aggiunta di campi.
- Altre modifiche sono incompatibili, come la rimozione di campi
- Cosa viene serializzato?
 - I magic data: **STREAM_MAGIC**, **STREAM_VERSION** (versione della JVM)
 - I metadati che descrivono la classe associata all'istanza dell'oggetto serializzato: *nome della classe, serialVersionUID della classe, numero dei campi, altri flag*
 - Valori associati all'oggetto, ovviamente anche quelli dei padri
 - Non si registrano i metodi della classe

5 Quinta lezione

5.1 Reader e Writer // todo aggiungere collegamento

- La classe **InputStreamReader** che estende **Reader** assume che i caratteri nel file siano codificati usando solo la codifica di default adottata dalla piattaforma locale.
- Converte i caratteri dalla codifica locale ad Unicode-UTF-16
- **System.in** e **System.out** sono istanze di **InputStream** e **OutputStream**, quindi leggono byte
- Per leggere caratteri:

```
BufferedReader myIn = new BufferedReader(new
    InputStreamReader(System.in));
```

5.2 JSON

- Codifica:

```
JSONObject obj = new JSONObject ();
obj.put("name", "foo");
obj.put("num", new Integer(100));
obj.put("balance", new Double(1000.21));
obj.put("is_vip", new Boolean(true));
```

- Decodifica:

```
String s=
"[0,
  {\"1\":
    {\"2\":
      {\"3\":
        {\"4\":
          [5, {\"6\":7}]
        }
      }
    }
  ]";
JSONParser parser = new JSONParser();
Object obj = parser.parse(s);
JSONArray array = (JSONArray) obj; // Rappresenta l'array pi
esterno
JSONObject obj2 = (JSONObject)array.get(1); // Rappresenta
l'oggetto con campo "1"
...
```

5.3 Bufferizzazione

- La JVM esegue una read() e provoca una system call (native code)
- Il kernel invia un comando al disk controller
- Il disk controller scrive direttamente un blocco di dati nella kernel memory (kernel buffer)
- I dati sono copiati dal kernel buffer nello user space all'interno della JVM
- Definendo dei buffer nello user's space si ottimizza notevolmente la performance dei programmi
- Senza bufferizzazione:
 - Un'interazione con il kernel per ogni read effettuata

- Un solo byte trasferito per volta dal kernel space allo user space
- Continui cambi di contesto
- Con bufferizzazione:
 - Si definisce un buffer nello user space
 - Il buffer è gestito dalla JVM in modo implicito
 - Dimensione di default a 8k
 - Una read legge un chunk di dati

5.4 Java NIO

5.4.1 Obiettivi

- Incrementare la performance dell'I/O
- Fornire un insieme eterogeneo di funzionalità per I/O
- Aumentare l'espressività delle applicazioni

Per raggiungere questi obiettivi Java NIO ha introdotto: **Buffers**, **Channels**, **Selector**

5.4.2 Buffers

Contenitore di dati di dimensione fissa. E' simile ad un `byte[]` ma è gestito in modo tale che la memoria interna possa essere un blocco della memoria di sistema. Leggere e scrivere in un direct buffer è quindi un modo diretto di trasferire informazioni tra il programma ed il sistema operativo. Realizza quindi un accesso diretto alla kernel memory da parte della JVM.



- Contengono dati appena letti o che devono essere scritti su un **Channel**.
- Oggetti della classe `java.nio.Buffer`, che fornisce un insieme di metodi che supportano la sua gestione
- Non è thread-safe

Quando il programma effettua una `read()` indica al canale di scrivere nel buffer che viene passato alla read. Successivamente il programma legge i dati dal buffer.

La stessa cosa succede alla scrittura. Il programma scrive dei dati nel buffer e poi, successivamente, quando si effettua una `write()`, si indica al canale di leggere i dati nel buffer.



5.4.2.1 Allocazione di un Buffer

Esistono diverse soluzioni per allocare un buffer:

- In un array privato all'interno dell'oggetto **Buffer**:

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

- In un array creato dal programmatore e wrappato:

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

In questo caso ogni modifica al buffer è visibile nell'array e viceversa.

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

```
ByteBuffer directBuf = ByteBuffer.allocateDirect(10);
```

5.4.2.2 Stato del Buffer

Ogni Buffer ha associati alcuni campi e metodi:

- **position**: Puntatore alla posizione corrente. Viene aggiornata implicitamente dalle operazioni di lettura e scrittura sul buffer.
 - In modalità scrittura rappresenta la posizione in cui si deve scrivere. Inizializzata a 0 e incrementata in seguito a delle scritture.
 - In modalità lettura rappresenta la posizione da cui si iniziare a leggere
- **limit**: Limite per leggere/scrivere. Anche questo viene aggiornato implicitamente dalle operazioni di lettura e scrittura sul buffer.
 - In modalità scrittura rappresenta quanto spazio rimane per scrivere
 - In modalità lettura rappresenta fino a dove posso leggere
- **capacity**: Dimensione massima del buffer.
- **flip()**: Utilizzata per passare da modalità in scrittura a lettura.
 - $limit = position$
 - $position = 0$

Intuitivamente, in modalità scrittura ho scritto, iniziando da 0 fino a position. Quando eseguo la flip e passo in modalità lettura, inizierò a leggere da 0 e potrò leggere fino a dove avevo scritto, ovvero position.

- **clear()**: Ritorna in modalità scrittura:

- *limit* = *capacity*
- *position* = 0

5.4.2.3 Viste del Buffer

- Solo buffer di byte possono essere utilizzato per operazioni di I/O
- Si possono però utilizzare delle viste in modo da interpretare i byte letti come un altro tipo di dato

```
CharBuffer chBuf = ByteBuffer.allocateDirect(10).asCharBuffer();// 5 char
IntBuffer intBuf = ByteBuffer.allocateDirect(10).asIntBuffer(); // 2 int
```

5.4.3 Channel

Sono simili agli stream ma offrono diverse funzionalità. Tra queste è possibile realizzare:

- *two way communcation*
- *scatter read*: Distribuisce i dati letti da un canale in uno o più buffers
- *gathering write*: Scrive su un canale dati recuperati da più buffer
- Trasferimenti diretti tra canali

I Channel sono connessi a descrittori di file/socket gestiti dal sistema operativo. Le implementazioni dell'interfaccia *Channel* elencate di seguito utilizzano principalmente codice nativo.

- **FileChannel**: Legge e scrive dati su un File
- **DatagramChannel**: Legge e scrive dati sulla rete via UDP
- **SocketChannel**: Legge e scrive dati sulla rete via TCP
- **ServerSocketChannel**: Attende richieste di connessioni TCP e crea un SocketChannel per ogni connessione creata.

5.4.3.1 Channel vs Stream

- Lo stesso Channel può leggere dal dispositivo e scrivere sul dispositivo.
- Tutti i dati sono gestiti tramite oggetti di tipo Buffer.
- Possono essere bloccanti o meno. I channel non bloccanti sono utili per comunicazione in cui i dati arrivano in modo incrementali (tipiche dei collegamenti di rete).
- Possibile il traferimento da Channel a Channel, se almeno uno dei due è FileChannel.

5.4.3.2 File channels

- Possono essere creati direttamente utilizzando *FileChannel.open*.
- Bisogna dichiarare il tipo di accesso al channel (READ/WRITE)
- Lettura e scrittura richiedono come parametro un ByteBuffer
- Sono bloccanti e thread safe.
- Alcune operazioni (read) possono essere eseguite in parallelo

5.4.3.3 Stream vs buffer Supponiamo di voler leggere un po' di righe da un file.

- Con gli stream una soluzione potrebbe essere la seguente:

```
InputStream input = ... ; // get the InputStream from the client
    socket
BufferedReader reader = new BufferedReader(new
    InputStreamReader(input));
String nameLine = reader.readLine();
String ageLine = reader.readLine();
String emailLine = reader.readLine();
String phoneLine = reader.readLine();
```

In questo caso quando l'invocazione della `readLine` restituisce il controllo al chiamante, siamo sicuri che è stata letta una linea di testo. Quindi ad ogni passo il programma sa quali dati sono stati letti. D'altra parte, una volta letto dei dati, non si può tornare indietro sullo stream.

- Con i buffer:

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
```

In questo caso, quando la `read` restituisce il controllo non è detto che siano stati letti tutti i byte necessari per comporre una linea di testo. Bisogna quindi verificarlo a mano. Allo stesso tempo però ha notevoli vantaggi (quelli elencati all'inizio)

5.4.3.4 Direct channel E' possibile connettere direttamente due canali e di trasferire direttamente dati dall'uno all'altro. Il trasferimento è implementato direttamente nel kernel (quando esiste questa funzionalità a livello di sistema operativo)

6 Lezione 6

6.1 Comunicazione tramite socket

Una socket è un modo di standardizzare la comunicazione. Due tipi di socket:

6.2 Connection oriented

- Creazione di una connessione stabile tra mittente e destinatario.
- Canale di comunicazione dedicato.
- Invio dei dati sulla connessione.
- Chiusura della connessione.
- L'indirizzo del destinatario (IP + porta) è specificato al momento dell'apertura della connessione
- Ordinamento garantito
- Utilizzato per trasmettere grosse moli di dati
- Lato client utilizza la classe **Socket**
- Lato server utilizza la classe **Socket** e **ServerSocket**

6.3 Connectionless

- Non stabilisce un canale di comunicazione dedicato.
- Ogni messaggio viene instradato in modo indipendente dagli altri.
- L'indirizzo del destinatario viene specificato in ogni pacchetto.
- L'ordinamento dei dati scambiati non è garantito.
- Utilizzato per inviare un numero limitato di dati.
- Utilizza la classe **DatagramSocket** sia lato server che lato client

6.4 Indirizzo IP

- 4 byte, ognuno interpretato come un numero decimale senza segno.
- Ad ogni indirizzo si associa una porta in modo da identificare il servizio.

6.4.1 La classe `InetAddress`

Rappresenta gli indirizzi internet tramite:

- Una stringa che rappresenta il nome simbolico di un host.
- Un intero che rappresenta l'indirizzo IP dell'host.
- Non definisce costruttori ma solamente metodi statici per costruire oggetti di tipo **`InetAddress`**.

Alcuni metodi:

- **`public static InetAddress getByName(String hostName)`**: Cerca l'indirizzo IO corrispondente all'host di nome `hostName` e restituisce un oggetto di tipo `InetAddress`. Può essere utilizzato anche per tradurre un indirizzo IP nel nome simbolico corrispondente.
- **`public static InetAddress[] getAllByName(String hostName)`**: Comportamento analogo al metodo `getByName` ma restituisce tutti gli indirizzi associato a quel host
- **`public static InetAddress getLocalHost()`**: Utilizzato per reperire nome simbolico e indirizzo IP del computer locale

Un oggetto `O` è uguale ad un oggetto `InetAddress` se e solo se `O` è un oggetto `InetAddress` e i byte array restituiti dalla `getAddress` invocata sui due oggetti hanno la stessa lunghezza e le componenti uguali byte a byte. Ovvero hanno lo stesso indirizzo IP.

Oss: I metodi descritti sopra effettuano caching.

6.5 Network interface

Se un host ha più di una interfaccia sulla rete vuol dire che ha associati più indirizzi ip, uno per ogni interfaccia. La funzione **`public static NetworkInterface getByAddress(InetAddress address)`** restituisce un oggetto `NetworkInterface` che rappresenta la network interface collegata ad un indirizzo IP.

6.6 Indirizzo di loopback

L'indirizzo 127.0.0.1 è utile per:

- Testare applicazioni
- Eseguire client e server in locale sullo stesso host

Ogni dato spedito utilizzando l'indirizzo di loopback in realtà non lascia l'host locale.

6.7 Il protocollo TCP in Java

- Il server pubblica un proprio servizio associandolo al *listening socket*.
- Il client C che intende usufruire del servizio deve conoscere l'indirizzo IP del server ed il riferimento della porta remota a cui è associato il servizio.
- Quando il client si connette alla *listening socket*, il server crea una *active socket* che verrà utilizzata per la comunicazione tra i due.

6.7.1 Lato client

Per connettersi al server si possono utilizzare due metodi:

- ---

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

Questo metodo crea una active socket e tenta di stabilire, tramite esso, una connessione con l'host individuato da InetAddress e sulla porta port. Se la connessione viene rifiutata viene lanciata l'eccezione IOException.

- ---

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

Il comportamento di questo metodo è analogo al precedente tranne che l'host viene individuato dal suo nome simbolico, quindi effettua una interrogazione al DNS

6.7.2 Lato server

Per creare la *listening socket* ci sono tre metodi possibili:

- ---

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

Costruisce un *listening socket* associandolo alla porta port

- ---

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

Comportamento analogo al precedente. Inoltre si specifica la lunghezza della coda in cui vengono memorizzate le richieste di connessione. Se la coda è piena, ulteriori richieste di connessione verranno rifiutate.

- ---

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

Permette di collegare il socket ad uno specifico indirizzo IP locale. Utile per macchine dotate di più schede di rete

Per accettare una nuova connessione dal *listening socket* bisogna chiamare il seguente metodo della classe **ServerSocket**:

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

Quando il server invoca questo metodo, si mette in attesa di connessioni. Quando arriva una richiesta il server si blocca e costruisce una nuova socket S tramite la quale avverrà la comunicazione effettiva tra client e server.

6.7.3 La comunicazione tra server e client

Una volta che il client si è connesso, sia server che client hanno una **Socket** con cui comunicare. La classe **Socket** mette a disposizione i metodi **getInputStream()** e **getOutputStream**. Questi ritornano rispettivamente lo stream per leggere e per scrivere al client/server.

Oss: Lo stream di input di uno è quello di output dell'altro e viceversa.

6.8 Shot down output

- Quando si esegue una **shotDownOutput()** su una socket, questa rimarrà aperta solamente in input.
- Una eventuale successiva wrtie sullo stream di output da parte dell'applicazione locale genera una **IOException()**
- Il metodo può essere utilizzato per inviare un EOF sulla connessione in modo da segnalare la terminazione dello stream.
- Se l'host remoto effettua una **read()** su uno stream che è stato chiuso mediante una **shotDownOutput()** all'altro capo della comunicazione possono accadere due risultati:
 - Restituire un numero di caratteri letti pari a -1
 - Sollevare l'eccezione **EOFException**

7 Lezione 7

7.1 Java NIO

7.1.1 Obiettivo

- *Fast buffered binary e character I/O* come visto qualche lezione fa
- *Non blocking mode e multiplexing*

7.2 Non blocking mode

Come abbiamo visto nella lezione precedente sugli stream, il thread rimane bloccato in varie situazioni. Nella `accept()` finché non viene stabilita una nuova connessione. Nella `write(byte[] buffer)` finché tutto il contenuto del buffer non è stato spedito sulla rete. Infine nella `read()` finché non è stato letto un byte.

7.2.1 SocketChannel

- Operano in modalità non blocking
- Possono essere selezionati tramite un Selector

Le applicazioni sviluppate con i SocketChannel offrono maggior flessibilità e scalabilità

7.2.2 Selector

- Pochi thread possono gestire centinaia o migliaia di connessioni con client diversi

7.2.3 ServerSocketChannel

Analogo a ServerSocket ma con un interfaccia channel-based. Le differenze sono:


- Nella modalità blocking: Comportamento analogo a ServerSocket ma con un interfaccia buffer-based
- Nella modalità non blocking: L'`accept()` ritorna immediatamente il controllo al chiamante e può restituire null se non sono presenti richieste di connessione. Altrimenti restituisce un oggetto di tipo SocketChannel

La modalità non blocking è utile per gestire più socket contemporaneamente. E' l'alternativa all'attivare un thread per ogni connessione.

Esempio di ServerSocketChannel:

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

7.2.4 Write nei SocketChannel



```
SocketChannel socketChannel = serverSocketChannel.accept();
if (socketChannel != null) {
    String dateAndTimeMessage = "Date: " + new
        Date(System.currentTimeMillis());
    ByteBuffer buf = ByteBuffer.allocate(64);
    buf.put(dateAndTimeMessage.getBytes()); // Scrive una sequenza di
        byte nel buffer
    buf.flip(); // Gira il buffer in modo da prepararlo per la lettura
    while (buf.hasRemaining()) { // Finch c'è qualcosa nel buffer
        socketChannel.write(buf); // Lo scrivo nella socket channel
    }
    System.out.println("Sent: " + dateAndTimeMessage);
    Thread.sleep(10000);
} else {
    System.out.println("nessuna connessione rilevata");
    Thread.sleep(1000);
}
```

7.2.5 Creazione di un SocketChannel

- **Implicita:** Creato quando una connessione viene accettata su un ServerSocketChannel
- **Esplicita:** Quando dal client si apre una connessione verso il server:

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

Per settare la modalità blocking/non blocking basta chiamare il metodo *configureBlocking(true)* o *configureBlocking(false)* sulla socket channel.

Un socket channel non bloccante lato client è utile per quando si deve gestire l'interazione con l'utente

7.2.6 Read su un SocketChannel

- Se il canale è in modalità blocking e c'è almeno una posizione libera nel buffer, il metodo si blocca fino a che non è stata riempita quella posizione.
- Restituisce -1 quando il server chiude il socket. Viene utilizzato come segnalazione di fine della connessione.
- Il numero di byte letto può essere anche 0 se il canale è in modalità non blocking.

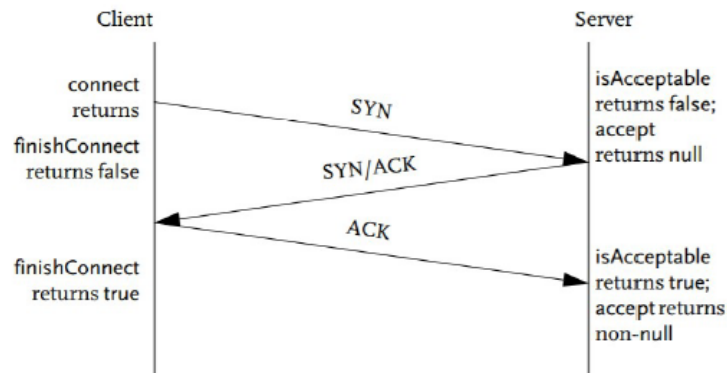


Figure 2: Non blocking connect

7.2.7 Non blocking connect

Se si effettua una `connect()` su un canale non bloccante:

- Il metodo può restituire il controllo al chiamante prima che venga stabilita la connessione
- Come si può vedere dalla figura 2, l'invocazione del metodo **finishConnect()** da parte del client restituisce `true` solo se il server ha accettato la connessione, ovvero solo se mi ha inviato `SYN/ACK`.
- Allo stesso modo, il metodo **isAcceptable()** invocato dal server torna `true` solo se il client ha inviato l'`ACK`.

7.3 Server models

Criteri per la valutazione delle prestazioni di un server:

- **Scalability:** Capacità di servire un alto numero di client che inviano richieste concorrentemente.
- **Acceptance latency:** Tempo tra l'accettazione di una richiesta da parte di un client e la successiva.
- **Reply latency:** Tempo richiesto per elaborare una richiesta ed inviare la relativa risposta.
- **Efficiency:** Utilizzo delle risorse utilizzate sul server (RAM, numero di threads, utilizzo della CPU)

7.3.1 Single thread model

Un solo thread per tutti i client:

- **Scalabilità:** Nulla. In ogni istante solo un client viene servito.
- **Accept latency:** Alta. Il prossimo client deve attendere fino a che il primo client termina la connessione.
- **Raply latency:** Bassa. Tutte le risorse a disposizione di un singolo client.
- **Efficiency:** Buona. Il server utilizza esattamente le risorse necessarie per il servizio dell'utente

In generale è adatto quando il tempo di servizio di un singolo utente è garantito che rimanga basso

7.3.2 One thread for connection

- **Scalabilità:** Possibilità di servire diversi client in maniera concorrente, fino al massimo di numero di thread previsto per ogni processore.
- **Accept latency:** In genera basso rispetto a quello di interarrivo delle richieste.
- **Raply latency:** Bassa. Le risorse del server vengono condivise tra connessioni diverse
- **Efficiency:** Bassa. Ogni thread può essere bloccato in attesa di IO, ma utilizza comunque risorse come la RAM.

Il grande problema di questo approccio è che la maggior parte del tempo si impiega nel context switching.

7.3.3 Numero fisso di thread

Viene utilizzato un thread pool.

- **Scalabilità:** Limitata al numero delle connessioni che possono essere supportate.
- **Raply latency:** Bassa fino al numero massimo di thread fissato. Degrada se il numero di connessioni è maggiori.
- **Efficiency:** Trade-off rispetto al modello precedente.

7.3.4 I/O Multiplexing (Selector)

- Il selettore è un componente che esamina uno o più NIO Channels e determina quali canali sono pronti per leggere/scrivere
- Più connessioni di rete gestite mediante un unico thread. Questo consente di ridurre l'overhead per il thread switching e l'uso di risorse per thread diversi.
- Miglioramento delle performance e della scalabilità
- Lo svantaggio è che l'architettura diventa più complessa da capire e da implementare.

7.4 Selector

Componente Java che può esaminare uno o più NIO Channels e che può determinare se sono pronti per una operazione. In Java è supportato da due classi principali:

- **Selector**: Fornisce le funzionalità di multiplexing.
- **SelectionKey**: Identifica i tipi di eventi pronti per essere elaborati.

Per utilizzare un selettore occorre:

- Creare il selettore

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

- Registrare i canali su quel selettore:

Applicata ad un oggetto Channel non bloccante

```
SelectionKey key =  
    channel.register(selector, [Operation1|Operation2|], [attachment]);
```

Restituisce una chiave **SelectionKey**. Lo stesso canale può essere registrato con più selettori. **La chiave identifica la particolare registrazione.**

Le operazioni possono essere del tipo:

- *OP_ACCEPT*
- *OP_CONNECT*
- *OP_READ*
- *OP_WRITE*

- Selezionare un canale quando è disponibile

La **SelectionKey** memorizza il Channel, l'operazione sul Channel, l'allegato e due bit set codificati come interi:

- **Interest set**: Bitmask che rappresenta le operazioni per cui è stato registrato un interesse. E' definita in fase di registrazione del canale passando alla `register(..)`, le operazioni. E' possibile modificarla successivamente invocando il metodo **interestOps(Operation1—Operation2—...)** della **SelectionKey**, con argomento una nuova maschera. Reperibile tramite il metodo **interestOps()**
- **Ready set**: Insieme delle operazioni sul canale associato alla chiave che sono pronte. E' un sottoinsieme dell'interest set. Viene inizializzato a ' quando la chiave viene creata e successivamente aggiornato quando si esegue il metodo **select()**. Per vedere il ready set basta invocare il metodo **readyOps()**. Per controllare ad esempio se il canale è pronto per la lettura posso:

- *if ((key.readyOps() & SelectionKey.OP_READ) != 0)*
- *key.isReadable()*

7.4.1 Multiplexing dei canali

Dopo aver visto alcuni metodi e classi utili passiamo a vedere come viene fatto il multiplexing vero e proprio:

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

- Seleziona i canali pronti per almeno una delle operazioni di I/O tra quelle registrate con quel selettore
- Restituisce il numero di canali pronti
- Si blocca finché o almeno un Channel non è pronto o il thread che esegue la selezione viene interrotto o il selettore viene sbloccato da una `wakeup()`

Una variante di questo metodo è quella che prende come parametro un *long timeout*. Questo determina il tempo massimo per cui il thread rimane in attesa sulla `select()`.

Ogni oggetto **Selector** mantiene i seguenti insiemi di chiavi:

- **Registered key**: **SelectionKey** dei canali registrati con quel selettore. Possibile vederle tutte tramite il metodo **keys()**
- **Selected key set**: **SelectionKey** dei canali identificati come pronti (nell'ultima operazione di selezione) ad eseguire almeno una operazione contenuta nell'interest set della chiave. Restituite dal metodo **selectedKeys()**

- **Cancelled key set:** Contiene le chiavi invalidate, ovvero quelle su cui è stato invocato il metodo *cancel()*, ma che non sono state ancora deregistrate.

Il processo di selezione (*select()*) prevede due fasi:

- Rimozione di ogni chiave appartenente al cancelled key set dagli altri due insiemi e rimozione della registrazione del canale associato alla chiave cancellata
- Interazione con il sistema operativo per verificare lo stato di "readiness" di ogni canale, per ogni operazione specificata nel suo interest set. Per ognuna di queste:
 - Se la chiave corrispondente non apparteneva già (dalla chiamata della *select()* precedente) al selected key set:
 - * La chiave viene inserita nel selected key set
 - * Il ready set 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.

Oss: Una chiave aggiunta al selected key set può essere rimossa solo con una operazione di rimozione esplicita. Questo è dato dal fatto che si esegue un or bit a bit.

Oss: Il ready set di una chiave inserita nel selected key set non viene mai resettato, ma viene incrementalmente aggiornato. Per resettare il ready set bisogna rimuovere la chiave dell'insieme delle chiavi selezionate.

Possiamo descrivere ora un pattern generale per implementare il multiplexing tramite *select*:

```
while(true) {
    try {
        selector.select();
    } catch (IOException ex) {
        ex.printStackTrace();
        break;
    }

    Set<SelectionKey> selectedKeys = selector.selectedKeys();
    Iterator <SelectionKey> keyIterator = selectedKeys.iterator();
    while(keyIterator.hasNext()) {
        SelectionKey key = (SelectionKey) keyIterator.next();
        keyIterator.remove();
        try{
            if(key.isAcceptable()) {
```

```

        // a connection was accepted by a ServerSocketChannel.
    } else if (key.isConnectable()) {
        // a connection was established with a remote server.
    } else if (key.isReadable()) {
        // a channel is ready for reading
    } else if (key.isWritable()) {
        // a channel is ready for writing
    }
    } catch (Exception e) {
        key.cancel();
        try {
            key.channel().close();
        } catch (IOException cex) {
            // Do something
        }
    }
}
}
}

```

8 Lezione 8

8.1 TCP vs UDP

- Con TCP occorre connettere il socket al server per aprire una connessione
- Con UDP un socket non deve essere connesso ad un altro socket prima di essere utilizzato.
- In UDP ogni messaggio (Datagram) è indipendente dagli altri e porta con sé l'informazione per il suo instradamento.
- Con TCP la trasmissione è vista come uno stream continuo di byte provenienti dallo stesso mittente.
- In UDP ogni ricezione si riferisce ad un singolo messaggio inviato mediante una unica send.

8.2 UDP in Java

Il datagram è un messaggio indipendente in cui l'arrivo ed il tempo di ricezione non sono garantiti. In Java viene modellato tramite la classe **DatagramPacket**.

Il mittente deve inizializzare il campo Data, l'indirizzo IP e la porta del destinatario. L'indirizzo IP e la porta della sorgente vengono inserite automaticamente.

Un'altra classe che ci servirà per implementare una trasmissione UDP è la classe **DatagramSocket**.

Per inviare un datagramma si deve:

- Istanziare un oggetto di tipo **DatagramSocket** collegato ad una porta locale (non è necessario pubblicarla).
- Creare un oggetto di tipo **DatagramPacket** in cui devo inserire:
 - Un riferimento ad un byte array contenente i dati da inviare nel payload del datagramma
 - Indirizzo IP e porta del destinatario dell'oggetto creato.
- Invia il **DatagramPacket** tramite una send invocata sull'oggetto **DatagramSocket**.

Per ricevere un datagramma si deve:

Creare una **DatagramSocket** e collegarla ad una porta che verrà pubblicata.

Creare un **DatagramPacket** per memorizzare il pacchetto che riceveremo. Il **DatagramPacket** contiene un riferimento ad un byte array che conterrà il messaggio ricevuto.

Invocare una receive sul **DatagramSocket** passando il **DatagramPacket**.

Oss: Un processo può utilizzare lo stesso socket per spedire pacchetti verso destinatari diversi.

8.2.1 DatagramSocket

```
public DatagramSocket ( ) throws SocketException
```

- Crea una socket e la collega ad una porta anonima. Il sistema sceglie una porta non utilizzata e la assegna alla socket.
- Utilizzato generalmente lato client per spedire datagrammi.
- Per reperire la porta allocata bisogna utilizzare il metodo **getLocalPort()**

```
public DatagramSocket (int p) throws SocketException
```

- Utilizzato in genere lato server
- Crea una socket sulla porta specificata
- Solleva un'eccezione quando la porta è già utilizzata o quando si tenta di connettere il socket ad una porta su cui non si hanno diritti.

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

8.2.2 DatagramPacket

Ci sono 4 diversi tipi di costruttore:

```
DatagramPacket(byte[] buffer, int length)
DatagramPacket(byte[] buffer, int offset, int length) // Definisce la
    struttura utilizzata per memorizzare il pacchetto ricevuto. Il
    buffer viene passato vuoto alla receive che lo riempirà al momento
    della ricezione di un pacchetto. Se è stato settato un offset, la
    copia avviene da quella posizione in avanti. Length indica il
    numero massimo di byte che possono essere copiati nel buffer
DatagramPacket(byte[] buffer, int length, InetAddress remoteAddr, int
    remotePort)
DatagramPacket(byte[] buffer, int offset, int length, InetAddress
    remoteAddr, int remotePort) // Utilizzato per costruire in datagram
    da inviare. length indica il numero di byte che devono essere
    copiati dal buffer nei pacchetti IP a partire da offset (se
    indicato)
```

Un oggetto **DatagramPacket** contiene:

- Un riferimento ad un byte buffer che contiene i dati da spedire/ricevuti
- Un insieme di informazioni utilizzate per individuare la posizione dei dati da inserire/estrarre dal buffer (length, offset)
- Se il datagramma deve essere spedito conterrà anche informazioni di addressing (indirizzo IP e porta del destinatario)

Oss: La stessa cosa può essere fatta utilizzando i getter e setter.

8.3 Comunicazione UDP

- Il metodo send non è bloccante. Non bisogna attendere che il destinatario abbia ricevuto il pacchetto.
- Il metodo receive è bloccante. Il processo che esegue la receive si blocca fino al momento in cui viene ricevuto un pacchetto. Per evitare attese infinite è possibile associare alla socket un timeout. Quando il timeout scade, viene sollevata una **InterruptedException**

// TODO da pag 34

9 Lezione 9

Un'applicazione RMI (Remote Method Invocation) è in generale composta da due programmi separati: server e client. Il server crea un oggetto remoto e pubblica un suo riferimento e attende che i client invochino metodi sull'oggetto. Il client ottiene il riferimento all'oggetto remoto e invoca i suoi metodi.

9.1 Obiettivi

L'obiettivo principale di RMI è quello di permettere al programmatore di sviluppare applicazioni Java distribuite utilizzando la stessa sintassi e semantica utilizzata per i programmi non distribuiti.

Il meccanismo dei socket è flessibile e potente per la programmazione di applicazioni distribuite ma è di basso livello. Richiede la progettazione di veri e propri protocolli di comunicazione.

9.2 Remote Procedure Call (RPC)

Predecessore di RMI. E' un paradigma di interazione a domanda/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.
- La connessione remota è trasparente rispetto a client e server

In questo modo il programmatore non deve più preoccuparsi di sviluppare protocolli che si occupino del trasferimento di dati, della verifica e della codifica/decodifica.

9.2.1 Limiti

- Parametri e risultati devono avere tipi primitivi
- La programmazione è essenzialmente procedurale
- La localizzazione non è trasparente. Il client deve conoscere l'IP e la porta su cui il server è in esecuzione.
- Non basata sulla programmazione ad oggetti. Mancano quindi i concetti di ereditarietà, incapsulamento, polimorfismo, ...

9.3 Remote Method Invocation (RMI)

- Consente di avere oggetti remoti attivabili, ovvero server che si attivano "on demand", ovvero a seguito di una invocazione e che si disattivano quando non sono utilizzati.

- L'utilizzo di oggetti remoti risulta largamente trasparente. Una volta localizzato l'oggetto, il programmatore utilizza metodi dell'oggetto come se questi fossero locali. Inoltre codifica, decodifica, verifica e trasmissione dei dati sono effettuati dal supporto RMI in maniera completamente trasparente all'utente.
- Supporta un Java Security Manager per controllare che le applicazioni distribuite abbiano i diritti necessari per essere eseguite.
- Supporta un meccanismo di Distributed Garbage Collection (DGC) per deallocare gli oggetti per cui non esistono più riferimenti attivi.

L'architettura RMI prevede tre entità:

- **registry**: E' un servizio di naming che agisce da yellow pages. Il server registra gli oggetti remoti nel registry tramite la `bind` e i client cercano gli oggetti remoti nel registry tramite la `lookup`.
- **client**
- **server**

Quando si invoca un metodo di un oggetto remoto, il supporto provvede a trasformare i parametri della chiamata remota in dati da spedire sulla rete.

9.3.1 Implementazione di un servizio tramite RMI

Def: Un'interfaccia è remota se e solo se estende **java.rmi.Remote** o un'altra interfaccia che estende **java.rmi.Remote**.

L'interfaccia **java.rmi.Remote** non definisce alcun metodo. Il solo scopo è quello di identificare gli oggetti che possono essere utilizzati in remoto. I metodi definiti dalle interfacce remote devono sollevare l'eccezione **java.rmi.RemoteException**

9.3.1.1 Step 1

Definire l'interfaccia remota, ovvero un'interfaccia che estende **java.rmi.Remote** e definire i suoi metodi.

9.3.1.2 Step 2

Implementazione del servizio

- **Soluzione 1:** Definire una classe che implementi i metodi dell'interfaccia remota e che estenda la classe **RemoteObject** o una sua sottoclasse.

```
public class MyServerRemoto extends RemoteServer implements
    IntRemota {
    public MyServerRemoto() throws RemoteException { ..... }
}
```

Richiede esportazione esplicita dell'oggetto remoto. Ha il vantaggio di ereditare la semantica degli oggetti remoti definita da `RemoteServer`. Ha lo svantaggio di non poter estendere altre classi

- **Soluzione 2:** Definire una classe che implementi i metodi dell'interfaccia remota ed estenda la classe **UnicastRemoteObject**.

```
public class MyServerRemoto extends UnicastRemoteObject implements
    IntRemota {
    public MyServerRemoto() throws RemoteException {
        super(); // E' qui che il server viene esportato
    }
}
```

Simile alla soluzione precedente. Stessi vantaggi e svantaggi ma diversa esportazione dell'oggetto. Il costruttore di **UnicastRemoteObject** crea automaticamente un server socket per ricevere le invocazioni di metodi remoti.

- **Soluzione 3:** Definire una classe che implementi i metodi dell'interfaccia remota senza estendere alcuna delle classi viste.

```
public class MyServerRemoto extends MyClass implements IntRemota {
    public MyServerRemoto() throws RemoteException {...}
}
```

Richiede l'esportazione esplicita dell'oggetto remoto. Ha la possibilità di estendere un'altra classe utile per l'applicazione, ma d'altra parte, la semantica degli oggetti remoti viene demandata al programmatore (overriding dei metodi equals, hashCode, ...)

9.3.1.3 Step 3

Generazione dello stub.

Def: Lo stub è un oggetto che consente di interfacciarsi con un altro oggetto (il target) modo da sostituirsi ad esso. Nel nostro caso il target è l'oggetto remoto.

In breve lo stub si comporta da intermediario tra l'oggetto target e l'oggetto che invoca i metodi. Non fa altro che inoltrare le chiamate che riceve, al suo target.

Per attivare il servizio bisogna:

- **Crearlo:** Allocare un'istanza dell'oggetto remoto.

```
MyServerRemoto service = new MyServerRemoto();
```

- **Attivarlo:** Il servizio viene attivato mediante:

- Creazione dell'oggetto remoto

```
IntRemota stub =
    (IntRemota)UnicastRemoteObject.exportObject(service, 0);
```

- Registrazione dell'oggetto remoto in un registry

```
LocateRegistry.createRegistry(2048); // o su qualsiasi altra
    porta disponibile
Registry r = LocateRegistry.getRegistry(2048);
r.bind("NOME_SERVIZIO", stub);
```

N.B.: Da ora in avanti c'è un thread in attesa di invocazione di metodi remoti. Quindi anche se il main dovesse terminare, il server non terminerà.

9.3.2 Connessione a un servizio RMI

- Ricercare lo stub dell'oggetto remoto.

```
Registry r = LocateRegistry.getRegistry(2048);
IntRemota obj = (IntRemota) r.lookup("NOME_SERVIZIO");
```

- Accedere al Registry attivato sul server effettuando una ricerca con il nome simbolico dell'oggetto remoto.
- Il riferimento restituito è un riferimento allo stub dell'oggetto.
- Il riferimento restituito è di tipo Object, è necessario effettuare il casting al tipo definito nell'interfaccia remota.
- Invocare i metodi dell'oggetto remoto come se fossero locali. L'unica differenza è che occorre intercettare **RemoteException**

9.3.3 Lo stub

Uno stub è composto da due parti:

- **Stato:** Contiene essenzialmente tre informazioni:
 - L'IP dell'host su cui è in esecuzione il servizio (oggetto) remoto
 - La porta di ascolto
 - Un identificativo RMI associato all'oggetto
- **Metodi:** La classe stub implementa la medesima interfaccia remota del server remoto:
 - Il codice dei metodi dello stub non ha niente a che vedere con i metodi corrispondenti del server

- Ogni metodo contiene il codice per inviare argomenti, invocare l’oggetto remoto e ricevere risultati.
- Generato nella fase di esportazione del server e utilizzato dal client.

Alcune osservazione sugli stub:

- La generazione dello stub utilizza il meccanismo della **Reflection**. Questo consiste nell’ottenere informazioni sui metodi, campi e costruttori di una classe.
- La classe **Stub** riceve un parametro **ref** di tipo **RemoteRef**: la classe **Stub** è un wrapper per questo riferimento remoto.
- Lo scheduling dei metodi remoti avviene invocando il metodo **invoke()** su **ref**

9.3.4 Il registry

Java mette a disposizione del programmatore un semplice name server (registry) che consente la registrazione ed il reperimento dello Stub. E’ simile ad un DNS per oggetti remoti, contiene legami tra il nome simbolico dell’oggetto remoto ed il riferimento all’oggetto. Per creare e gestire oggetti di tipo Registry bisogna utilizzare la classe **LocateRegistry**. Prima abbiamo utilizzato due suoi metodi:

- *public static Registry createRegistry(int port)*: Lancia un servizio di registry RMI sull’host locale, su una porta specifica e restituisce un riferimento al registro.
- *public static Registry getRegistry(String host, int port)*: Restituisce un riferimento ad un oggetto RMI su un certo host ad una certa porta. Una volta ottenuto il riferimento si possono invocare i metodi definiti dall’interfaccia **Registry**, Solo allora viene creata una connessione con il registro. Se non esiste nessun registro RMI con quel nome e a quella porta, restituisce un’eccezione.
 - **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 del registro r, viene sollevata una eccezione.
 - **r.rebind()**: Crea un collegamento tra un nome simbolico ed un riferimento all’oggetto. Se esiste già un collegamento per lo stesso oggetto all’interno dello stesso registry, tale collegamento viene sovrascritto.

Oss: E’ possibile inserire più istanze dello stesso oggetto remoto nel registry, con nomi simbolici diversi.

10 Lezione 10

10.1 Paradigmi di comunicazione

- **Multicast:** Il sender invia solamente un messaggio. Questo sarà ricevuto da più utenti.
- **Unicast:** Il sender manda un messaggio per ogni utente che vuole raggiungere.
- **Broadcast:** Il sender invia solamente un messaggio. Questo sarà ricevuto da tutti gli altri utenti.

10.2 Gruppi multicast

- IP multicast è basato sul concetto di gruppo, ovvero un insieme di processi in esecuzione di host diversi.
- Tutti i membri di un gruppo di multicast ricevono un messaggio spedito su quel gruppo.
- Non occorre essere membri del gruppo per inviare i messaggi su di esso.
- Sono gestiti a livello IP dal protocollo IGMP.

10.3 Multicast API

Deve contenere almeno le primitive per:

- Unirsi ad un gruppo multicast.
- Lasciare un gruppo multicast.
- Spedire un messaggio ad un gruppo. Il messaggio viene recapitato a tutti i processi che fanno parte del gruppo in quel momento.
- Ricevere messaggi indirizzati ad un gruppo.

Il supporto deve fornire:

- Un meccanismo di indirizzamento per identificare univocamente un gruppo.
 - In IPv6 tutti gli indirizzi multicast iniziano con *FF*
 - In IPv4 gli indirizzi multicast sono tutti quelli di classe D:
 - * 224.0.0.0 - 239.255.255.255
 - * I primi 4 bit del primo ottetto sono *1110*.
 - * I restanti bit identificano il particolare gruppo.
- Un meccanismo che registri la corrispondenza tra un gruppo ed i suoi partecipanti (IGMP). Un meccanismo che ottimizzi l'uso della rete nel caso di invio pacchetti ad un gruppo di multicast.

10.3.1 Multicast addressing

L'indirizzo multicast deve essere noto collettivamente a tutti i partecipanti al gruppo. Esistono due tipi di indirizzi mutlticast:

- **Indirizzi statici:** Scelti da una autorità per un certo servizio. L'indirizzo rimane assegnato a quel gruppo anche se in un certo istante non ci sono partecipanti.
- **Indirizzi dinamici:** Si utilizzano protocolli particolari che consentono di evitare che lo stesso indirizzo multicast sia assegnato a due gruppi diversi. Questi indirizzi esistono solo fino al momento in cui esiste almeno un partecipante

10.3.2 Multicast scoping

Come limitiamo la diffusione di un pacchetto?

- **TTL scoping**
 - Il TTL limita la diffusione del pacchetto.
 - Ad ogni pacchetto IP viene associato un valore rappresentato su un byte.
 - Il valore TTL è nell'intervallo 0-255
 - Indica il massimo numero di router attraversati dal pacchetto
 - Il pacchetto viene scartato dopo aver attraversato TTL router
- **Administrative scoping:** A seconda dell'indirizzo in classe D scelto, la diffusione del pacchetto viene limitata ad una parte della rete i cui confini sono definiti da un amministratore di rete. Possono essere:
 - *Global scope*
 - *Organization-Local scope*
 - *Site-Local scope*
 - *Link-Local scope*
 - *Node-Local scope*

Oss: Multicast utilizza il paradigma connectionless (UDP)

10.4 Java API per Multicast

- Si utilizza la classe **MulticastSocket** che estende a sua volta la classe **DatagramSocket**.
- **MulticastSocket** rappresenta la socket su cui ricevere i messaggi da un gruppo di multicast.

10.4.1 La classe `MulticastSocket`

- Mette a disposizione il metodo `joinGroup`. Questa lega il **MulticastSocket** ad un gruppo multicast: tutti i messaggi ricevuti tramite quel socket provengono da quel gruppo.
- Se attivo due istanze di un programma che si connette su una MulticastSocket, non viene sollevata una **BindException** (cosa che avveniva con il **DatagramSocket**)
- Non esiste una corrispondenza biunivoca tra porta e servizio.

10.4.1.1 Ricezione di un pacchetto

```
InetAddress group = InetAddress.getByName("nome_server");
int port = 4000;

MulticastSocket ms = new MulticastSocket(port);
ms.joinGroup(group);

byte [] buffer = new byte[8192];
DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
ms.receive(dp);

ms.leaveGroup(group);
ms.close();
```

10.4.1.2 Spedire un pacchetto

```
InetAddress ia = InetAddress.getByName("228.5.6.7");
byte [] data;
data = "hello".getBytes();
int port = 6789;

DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);
DatagramSocket ms = new DatagramSocket(6789);
ms.send(dp);
```

Oss: Alla spedizione di un pacchetto, se si volesse specificare il TTL, basta invocare il metodo `setTimeToLive(n)` sull'oggetto `MulticastSocket`.

10.5 Il meccanismo delle callback

Il meccanismo delle callback consente di realizzare il pattern **Observer** in ambiente distribuite. Utile quando un client è interessato allo stato di un oggetto

e vuole ricevere una notifica asincrona quando tale stato viene modificato. Due possibili soluzioni per realizzare un servizio di notifica di eventi al client:

- **Polling:** Il client interroga ripetutamente il server per verificare l'occorrenza dell'evento atteso. L'invocazione può avvenire mediante l'invocazione di un metodo remoto (RMI). Soluzione molto costosa e non efficiente.
- Registrazione dei client interessati agli eventi e successiva notifica (asincrona) del verificarsi dell'evento ai client, da parte del server. In questo modo il client può proseguire la sua elaborazione senza bloccarsi ed essere avvertito solo quando l'evento si verifica.

10.5.1 Callback via RMI

RMI si può utilizzare sia per l'interazione client-server (registrazione dell'interesse per un evento) che server-client (notifica del verificarsi di un evento).

- Il server definisce un'interfaccia remota **ServerInterface** con un metodo remoto utilizzato dal client per registrare il suo interesse per un certo evento (**meccanismo di bootstrap**).
 - Quando riceve una invocazione del metodo remoto, memorizza il riferimento all'oggetto remoto del client (ROC) in una struttura dati.
 - Al momento della notifica, utilizza il ROC per invocare il metodo remoto sul client per notificarlo.
- Il client definisce una interfaccia remota **ClientInterface** con un metodo remoto utilizzato dal server per notificare un evento al client.
- Il client reperisce il riferimento all'oggetto remoto del server (ROS) tramite il registry.
- Il server riceve, al momento della registrazione del client, lo sub dell'oggetto remoto allocato sul client. Quindi il server non utilizza il registry per individuare l'oggetto remoto del client, ma si fa passare il riferimento dal client.

10.5.1.1 Esempio di implementazione

- Interfaccia client:

```
public interface NotifyEventInterface extends Remote {  
    public void notifyEvent(int value) throws RemoteException;  
}
```

Il metodo `notifyEvent` è il metodo esportato dal client e che verrà utilizzato dal server per la notifica di un evento.

- Implementazione dell'interfaccia del client:

```
public class NotifyEventImpl extends RemoteObject implements
    NotifyEventInterface {
    public NotifyEventImpl throws RemoteException {
        super( );
    }

    public void notifyEvent(int value) throws RemoteException {
        System.out.println("Notifica client: " + value);
    }
}
```

- Lancio del client:

```
Registry registry = LocateRegistry.getRegistry(5000);
ServerInterface server = (ServerInterface)
    registry.lookup("NOME_SERVER");

NotifyEventInterface callbackObj = new NotifyEventImpl();
NotifyEventInterface stub = (NotifyEventInterface)
    UnicastRemoteObject.exportObject(callbackObj, 0);
server.registerForCallback(stub);
// Do something
server.unregisterForCallback(stub);
```

- Interfaccia del server:

```
public interface ServerInterface extends Remote {
    public void registerForCallback (NotifyEventInterface
        ClientInterface) throws RemoteException;

    public void unregisterForCallback (NotifyEventInterface
        ClientInterface) throws RemoteException;
}
```

- Implementazione dell'interfaccia del server:

```
public class ServerImpl extends RemoteObject implements
    ServerInterface {
    private List <NotifyEventInterface> clients;
    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." );
    }
}

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.");
    }
}

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.");}
}

```

- Attivazione del server:

```

ServerImpl server = new ServerImpl( );
ServerInterface stub = (ServerInterface)
    UnicastRemoteObject.exportObject (server, 39000);
LocateRegistry.createRegistry(5000);
Registry registry = LocateRegistry.getRegistry(5000);
registry.bind ("NOME_SERVER", stub);

// Do something

server.update(10);

```

10.6 RMI e concorrenza

- Invocazioni di metodi remoti provenienti da client diversi (diverse JVM) sono 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.
- Invocazioni concorrenti provenienti dallo stesso client possono essere eseguite sia dallo stesso thread che da thread diversi.
- La politica di Java RMI di implementare automaticamente multithreading di chiamate diverse presenta il vantaggio di evitare all'utente di scrivere codice per i thread server side
- L'utente che sviluppa il server deve assicurare che l'accesso all'oggetto remoto sia correttamente sincronizzato (**synchronized**, **locks**, ...)

11 Lezione 11

11.1 Dynamic Class Loading

- RMI offre la capacità di reperire dinamicamente la definizione della classe di un oggetto, se non presente nel *local classpath*.
- Il meccanismo fornisce la possibilità di definire applicazione espandibili.
- Il **Dynamic Class Loading** è offerto da RMI e permette di scaricare dinamicamente un file *.class* ed eseguirlo.
- Meccanismi necessari per l'implementazione:
 - Individuazione del repository remoto delle classi.
 - Meccanismi di sicurezza: Quando si scarica un frammento di codice e poi lo si esegue, è possibile che quel codice possa recare danno, in modo intenzionale o meno, all'host che ha scaricato quel codice e che lo esegue.

11.2 RMIClassLoader

E' utilizzato per il caricamento remoto delle classi. Può essere invocato:

- Implicitamente con un comando Java o alla ricezione di oggetti remoti serializzati.
- Esplicitamente.

11.3 Alcuni scenari possibili

11.3.1 Thin client

Sviluppare un client minimale che carica dinamicamente il proprio codice da un'area condivisa dove lo ha memorizzato il server.

I vantaggi di questo approccio sono:

- Evitare la reinstallazione dei client quando vengono modificati
- Una unica copia del client presente in un'area condivisa
- Minor numero di inconsistenze. Se il client condividono la stessa classe nell'area pubblica, non si verificheranno inconsistenze (ad esempio con versioni diverse)

Il client scarica l'ultima versione del codice dal server e lo esegue:

- Metodo *RMIClassloader.loadClass*: carica dinamicamente ed in modo esplicito classi remote. Necessario un URL che riferisce la directory da cui scaricare il codice.

11.3.1.1 Esempio

```
public class LoadClient {
    public static void main(String[] args) {

        System.setSecurityManager(new mySecurityManager());
        Class cl = RMIClassLoader.loadClass(new URL("<url>"),
            "RunAway");
        Runnable client = (Runnable) cl.newInstance();
        client.run();
    }
}

public class mySecurityManager extends SecurityManager {
    public void checkConnect(String host, int port, Object context) { }
    public void checkPermission(Permission perm) { }
}

public class RunAway implements Runnable { // Questa sta solo nel server
    public void run() {
        System.out.println("I made it!");
    }
}
```

11.3.2 Caricamento dinamico degli stub

Come già visto nelle lezioni precedenti, il server genera uno stub. Questo però deve essere reso disponibile al client. Due soluzioni:

- Copiare off-line il file .class dello stub in una directory riferita dal **classpath** del client.
- Utilizzare il dynamic class loading

Oss: Nelle ultime versioni di Java questo non è più necessario in quanto lo stub viene generato dinamicamente con il meccanismo delle reflection.

11.3.3 Polimorfismo

Nei linguaggi ad oggetti denota la possibilità di usare un oggetto di una sotto-classe B ogni volta che viene indicato di usare un oggetto di classe A, con B *A*. Java RMI supporta la possibilità di caricare dinamicamente l'implementazione della sottoclasse. Questa soluzione può essere utilizzata sia lato client che lato server.

Il server conosce il tipo T del parametro, ma il client gli passa un oggetto sottotipo che estende T.

11.3.3.1 Individuare il class repository

codebase: Server su cui vengono caricati file .class che successivamente verranno scaricati dinamicamente. Viene annotata la rappresentazione serializzata dell'oggetto con un URL del codebase. Quando l'oggetto viene deserializzato, la JVM può scaricare il bytecode della classe dell'oggetto da deserializzare in modo trasparente, utilizzando l'annotazione.

Il meccanismo di individuazione del codebase consiste in:

- Viene settata la proprietà *java.rmi.server.codebase* della JVM. Il valore è un riferimento all'URL del codebase
- Le classi RMI che effettueranno la serializzazione di un oggetto:
 - Leggono il valore di questa proprietà
 - Incorporano l'URL nella rappresentazione serializzata dell'oggetto

Oss: Il *program argument* è un vettore di dati passati al programma main

Oss: I *VM arguments* sono passati alla macchina virtuale e vengono utilizzati dalla JVM. Ogni VM argument corrisponde ad una proprietà della VM. Questi possono essere impostati:

- Da linea di comando: *java -Dprop=val MyClass*

- Impostati attraverso dei metodi della classe System: *System.setProperty(prop, vla)*

java.rmi.server.codebase è una delle proprietà della JVM.

11.3.3.2 Esempio

- Considerando una classe C, possono eseguirla nel seguente modo:

```
java Djava .rmi.server.codebase=http://www.di.unipi.it/~ricci C
```

- Questo comando setta la proprietà **java.rmi.server.codebase** a `http://www.di.unipi.it/ricci`
- Questa URL verrà incorporata in ogni oggetto serializzato da C
- Se un oggetto viene successivamente ricreato mediante una deserializzazione effettuata da una JVM diversa e quella JVM non trova una copia locale del file .class associato a C (nel *classpath*), la JVM automaticamente genera una richiesta di download alla URL specificata nell'oggetto serializzato.

11.4 Code mobility e sicurezza

- La mobilità del codice pone ovvi problemi di sicurezza.
- Da Java 2 in poi la politica di sicurezza di Java impone all'utente di definire esplicitamente i permessi di cui deve disporre un'applicazione scaricata dalla rete.
- Tali permessi definiscono una sandbox, ovvero uno spazio virtuale in cui l'applicazione deve essere eseguita.
 - Una sandbox dovrebbe contenere il minimo numero di permessi che consentono l'esecuzione della applicazione
 - I permessi sono elencati in un *file di policy*.
 - Esistono due diversi tipi di *file di policy*
 - * *Amministratore di sistema*: Utilizza un file di policy per indicare cosa possono fare i programmi lanciati all'interno del sistema.
 - * *Sviluppatore dell'applicazione*: Distribuisce il file di policy che elenca i permessi di cui l'applicazione ha bisogno.

11.5 Il security manager

La garanzia che vengano rispettati i permessi, fissati nei file di policy, durante l'esecuzione del codice è fornita installando un'istanza della classe **SecurityManager**. Quando un programma tenta di eseguire un'operazione che richiede un permesso esplicito (come una lettura su un file o una connessione via socket), viene interrogato il **SecurityManager**.

Def: I programmi che scaricano dinamicamente del codice dalla rete devono impostare un **SecurityManager**.

Oss: RMI abilita il caricamento dinamico del codice solo in presenza del **SecurityManager**

Oss: In un'applicazione RMI, se non viene settato un security manager, gli stub e le classi possono essere caricate solamente dal *local classpath*.

11.5.1 Attivazione del security manager

Per attivare un **SecurityManager** ci sono due modi:

- All'inizio del programma chiamando il metodo **System.setSecurityManager(new SecurityManager())**
- Definendo la proprietà di sistema da linea di comando: **java -Djava.security.manager applicazione**

11.5.2 Il file di policy

- La politica di sicurezza deve essere specificata nel file di policy
- Un esempio di politica:

```
grant {// Allow everything
    permission java.security.AllPermission;
};
```

- La configurazione di default garantisce completa sicurezza. Ogni permesso richiesto dall'applicazione deve essere indicato esplicitamente. Ad esempio, le classi caricate dinamicamente non possono interagire col file system, la rete o l'interfaccia grafica.
- Si possono fornire permessi diversi a classi caricate da codebase differenti

12 Lezione 12

12.1 Representational State Transfer (REST)

- E' uno stile architetturali per sistemi software distribuiti

- Indica una serie di principi architetturale per la progettazione di Web Services
- Introdotto per diminuire la complessità dei framework heavyweight come SOAP
- Adottato da molti website che offrono una API per l'accesso ai loro servizi (Twitter, Paypal, ...)
- L'interazione molto spesso utilizza JSON, piuttosto che XML

Caratteristiche che deve avere un servizio web per essere conforme alle specifiche REST:

- Architettura client e server
- Stateless: Ogni ciclo di request-response deve rappresentare un'interazione completa del client con il server. In questo modo non è necessario mantenere informazioni sulla sessione di un utente, minimizzando l'uso di memoria del server e la sua complessità.
- Uniformemente accessibile: Ogni risorsa deve avere un'indirizzo univoco e ogni sistema presenta la stessa interfaccia per identificare le risorse, in genere individuata dal protocollo HTTP.
- Risorse:
 - Ogni risorsa è identificata attraverso un URL specifico.
- Rappresentazione delle risorse:
 - Le risorse sono entità astratte caratterizzate da uno stato che può cambiare nel tempo.
 - Lo stato può essere statico o dinamico.
 - Client e server comunicano scambiandosi una rappresentazione dello stato delle risorse. Il formato della rappresentazione deve essere concordato tra client e server. Ad esempio JSON, XML, PDF, ...
 - Nel protocollo HTTP, lo stato di una risorsa viene identificato attraverso la codifica definita dallo standard MIME (Multimedia Internet Mail Extension).
 - * MIME TYPE: specificato da una stringa che ha come formato
 - *Type/subtype*
 - *Type/subtype; option*
 - * Nomi dei tipi MIME assegnati da IANA:
 - *text/plain*: Formato di testo semplice
 - *text/plain; charset=UTF-8* testo semplice con codifica
 - *text/html*: Formato HTML

- *applicazione/xml*: Formato XML
- *image/jpeg*: Formato JPEG
- ...

- Operazioni sulle risorse

12.2 Operazioni

Nell'approccio REST si usano un numero limitato di operazioni per leggere o modificare lo stato di una risorsa. Questo sono:

- *GET articles/3*: Recupera una rappresentazione dell'articolo desiderato
- *POST articles/*: Crea un nuovo articolo, la cui rappresentazione è nel body della richiesta
- *PUT articles/3*: Modifica l'articolo esistente
- *DELETE articles/*: Rimuovo l'intera collezione di articoli

12.3 REST in Java

La definizione di un client che accede ad un server REST può essere effettuata in Java sfruttando le seguenti classi:

- *URL*

```
URL url = new
    URL("https://docs.oracle.com/javase/tutorial/networking/urls/creatingUrls.html");
InputStream stream = url.openStream();
```

- *URLConnection*

- Classe astratta implementata da classi specializzate nel controllo di una connessione ad un URL
- Si ottiene richiamando il metodo *openConnection()* della classe *URL*
- Permette di scegliere diversi parametri di connessione, di analizzare gli header inviati da un server, settare i campi header del client e di leggere/scrivere sulla connessione.
- Dopo aver scelto le opzioni, la connessione è avviata usando il metodo *connection()*
- L'header possiede diverse informazioni che possono essere reperite con metodi Java:
 - * content type: reperibile con il metodo *public String getContentType()*
 - * length

```
* charset
* expiration date
* modification date
* ...
```

```
try {
    URL u = new URL("http://www.bogus.com");
    URLConnection uc = u.openConnection();
    System.out.println(uc.getHeaderField(1) +
        uc.getHeaderFieldKey(1)); // Stampa il primo header
        e il valore associato
    InputStream raw = uc.getInputStream();
    // Read from URL
} catch (MalformedURLException ex) {
    System.err.println(ex);
} catch (IOException ex) {
    System.err.println(ex);
}
```

- *HTTPURLConnection*

- Sottoclasse astratta di *URLConnection*
- Fornisce i metodi aggiuntivi per lavorare con URL HTTP:
 - * Metodi get/set
 - * Decidere se seguire redirect
 - * Reperire i response code: *public int getResponseCode()* e *public String getResponseMessage()*
 - * Determinare se è utilizzato in proxy

```
URL u = new URL("http://www.di.unipi.it");
URLConnection uc = u.openConnection();
HttpURLConnection http = (HttpURLConnection) uc;
```
