

# Multithreading

Come lavorare alla mutua esclusione

→ mutex Reentrantlock

## • lock:

- Si definiscono il lock e le condition variables. → Attenzione all'istanziazione del lock
- Si assegnano le CV al lock (lock.mutexCondition());
- Come si lavora alla Sezione Critica:
  - + Si acquisisce il lock (lock.lock());
  - + Si usa il while (condizione) { condizione.await(); } (non dimenticare il blocco try)
  - + Fuori dal while lavoriamo alle operazioni
  - + Nel finally si fa l'unlock

## • Semaphore:

- Si definiscono ed inizializziamo i Semaphori (uno per ogni condizione)
- Come si lavora alla Sezione Critica:
  - + Nel try-catch si fa l'acquisto del Semaphoro
  - + Si apre un try finally che conterrà un blocco synchronized (risorsa critica)
  - + In tal blocco si compiono le operazioni
  - + Nel finally si farà la release del semaphoro

## • Synchronized:

- Si lavora nel blocco synchronized applicando wait() e notify() direttamente alla risorsa critica

## Focus: Stampa log in un File:

### 1. Creare il File:

```
FileOutputStream fileOut = new FileOutputStream("./mommaFile.txt");
```

### 2. Crea oggetto "Stampante":

```
PrintStream printStream = new PrintStream(fileOut);
```

### 3. Metodo di utilizzo oggetto stampante:

```
printStream.println("... " + var + "...");
```

## Con Writers e Buffer:

### 1. Istanzia un oggetto FileWriter a cui passare il nome del file:

```
FileWriter fw = new FileWriter("momma.txt", true);
```

### 2. Istanzia un oggetto BufferedWriter a cui passare l'istanza fw.

### 3. Istanzia un oggetto PrintWriter a cui passare l'istanza bw.

3.1 Usa i metodi println e flush del pw per scrivere sul file.

### 4. Chiudi i file nell'ordine inverso di apertura.

# Proxy - Skeleton

## • Interfaccia Servizio:

- + Contiene unicamente il file IServizio.java ossia l'interfaccia dove troveremo le definizioni dei metodi da implementare.

## • Lato Server: (per delega)

### 1. ServerThread:

- Bisognerà creare la classe che "estende" thread e che eseguirà le operazioni lato server.
- Contiene:
  - + Costruttore: a cui passeremo gli oggetti Socket ed IServizio
  - + Metodi Run:
    - o Contiene i Data Stream input ed output (che prendono lo stream della socket)
    - o Optional: Una stringa contenente dei parametri utili
    - o Ciclo di if: un caso per ogni metodo dell'interfaccia.

### 2. Skeleton:

- Implementa l'interfaccia del servizio; in particolare la comunicazione lato server
- Contiene:
  - + Il costruttore: a cui si affidano gli oggetti interfaccia e port
  - + Un semplice override dei metodi dell'interfaccia
  - + Metodo runSkeleton:
    - o Si inizializza una serverSocket a cui si affida la porta del servizio
    - o All'interno di un while(true) si inizializza un oggetto socket a serverSocket.accept() e si istanzia un thread server a cui passare la socket e l'interfaccia del servizio

### 3. Service Impl:

- Si occupa dell'effettiva implementazione dei metodi delle interface e della gestione in Mutua Esclusione delle risorse.

### 4. Service Server:

- Qui è dove va eseguito lo runSkeleton, a cui passeremo 2 istanze:
  - o Un oggetto serviceImplementation
  - o Uno skeleton (a cui si affida il s.impl e la porta scelta per il servizio)

## • Lato Client:

### 1. Service Proxy:

- Si occupa dell'implementazione del passaggio dei pacchetti contenenti parametri e risultato delle procedure chiamate.
- Contiene:
  - o Costruttore (a cui passare indirizzo e porta del servizio su cui mettensi in ascolto.)
  - o Override dei metodi forniti dall'interfaccia:
    - o Si definisce una socket per l'ascolto.
    - o Si usano come canali di comunicazione i Data (Input/Output) Stream

## 2 Clientthread:

- Definisce i thread lato client ed il loro modo di operare.

- Contiene:

- o Costruttore (a cui passare l'interfaccia ed una stringa per la scelta dei metodi)
- o Override di run(): Cosa fa il thread per ogni metodo

(Per ereditarietà)

Cosa Combina:

### 1. Skeleton:

- Diventa una classe astratta

- Implementa solo runSkeleton (e all'oggetto Serverthread si passano (this, socket))

- Al costruttore si passa solo il port number.

### 2. ServiceImplementation: (extends SkeletonClass)

- Il costruttore inizializza anche il numero di porto con "super(port)"

### 3. Service Server:

- Si dichiara solo un oggetto ServiceImpl a cui si passa il numero di porto

- Si richiama con tale oggetto il metodo runSkeleton().

## Nota: Creare un File

1. Definisce la variabile per il nome: String docName

2. Definisce il percorso: String path = ". / " + docName

3. Istruisce un oggetto File: File file = new File(path)

3.1 Check nel ciclo: if (file.exists == false) { file.createNewFile () }

4 Poi per la stampa si sfrutta il FileWriter (a cui si passa l'oggetto file)

## Lavorare ad un File che dovrà contenere più messaggi:

1. Istruisce l'oggetto File e gli passiamo il path nome.

2. blocco try:

{ 2.1 Istruisce un FileOutputStream e gli passiamo il file.

2.2 Istruisce un BufferedWriter e gli passiamo un istanza OutputStreamWriter che contiene il fileout.

2.3 Utilizziamo bw.write() e bw.newLine per scrivere (all'interno di un ciclo for!)

2.4 bw.newLine finale + bw.close();

while (true)

# JAVA RMI

## 1. Interfaccia:

- Contengono solo le firme dei metodi dei servizi a cui fanno riferimento
- Nel caso RMI le classi contengono "extends Remote"
- I metodi richiedono l'osservazione delle eccezioni con "throws RemoteException"

## Dato server:

### 2. ServiceServer:

- Contiene il main del server
- Si occupa di:
  - o Creare l'oggetto Registry e farne la LocateRegistry.getRegistry()
  - o Istruire un oggetto ServiceImplementation e farne il Bind

### 3. ServiceImplementation:

- Contiene l'implementazione dei metodi definiti per la sua interfaccia
- Dichiarazione:

```
public Class ServerImpl extends UnicastRemoteObject implements IService
```
- Importante per il riconoscimento che contenga un parametro: serialVersionUID (VID) come ottenerlo:
  - o Raggiungere la cartella bin dal terminale
  - o Scrivere: serial momenpackage.momenfileImpl
  - o inserire il valore risultante come: private static final long serialVersionUID = ...

## Dato Client:

### 4. ClientThread:

- Si occupa della definizione del comportamento dei thread tramite metodo Void run()

### 5. Client:

- È il main client, esegue le fasi per l'invio dei thread e per le seguenti: join.

\* Tale processo va seguito ogni volta che in un file vogliamo usare dei metodi di un oggetto che implementa la comunicazione tramite RMI!

\* Attenzione a servizi già esistenti e che hanno già avuto il loro bind(); in tal caso (Spresso jMS-RMI): IService service = (IService) rmiregistry.lookup("service");

# JMS

\* Ricorda di attivare activeMQ da terminale:

## Avvio applicazioni JMS



1. Avvio ActiveMQ
    - cd /<ACTIVEMQ\_PATH>/bin
    - ./activemq start
  2. Avvio subscriber (receiver) e publisher (sender)
    - cd /<PROJECT\_PATH>/bin
    - java -cp "<ACTIVEMQ\_PATH>/activemq-all-X.Y.Z.jar;" package\_name.class\_name
- Note:
  - <ACTIVEMQ\_PATH> è il path dove è stato estratto il contenuto del file compresso con ActiveMQ
  - In activemq-all-X.Y.Z.jar le lettere X.Y.Z vanno sostituite con la versione utilizzata.
    - Per ActiveMQ 5.16.6 (utilizzato al corso) il nome del file è activemq-all-5.16.6.jar
  - Attenzione ad includere il separatore ":" per i sistemi UNIX-based e ";" a valle del path al file menzionato al punto precedente
  - Gli step al punto 2, vanno ripetuti in diversi terminali, uno per ogni entità da avviare (cioè un terminale per ogni subscriber/receiver ed un terminale per ogni publisher/sender)
  - Il file activemq-all-5.16.6.jar è da includere anche nel classpath del progetto in Visual Studio Code, come *referenced library*

49

Questo è fondamentale anche per il riconoscimento delle librerie javax.jms

## Lookup dei Servizi (sia Client che Server):

Entrambi i file .java dovranno presentare una configurazione iniziale che è sempre "identica":

```
import java.util.Hashtable;
import javax.jms.*;
import javax.naming.*;

Hashtable<String, String> prop = new Hashtable<String, String> ();

prop.put( "java.naming.factory.initial",
"org.apache.activemq.jndi.ActiveMQInitialContextFactory" );
prop.put( "java.naming.provider.url", "tcp://127.0.0.1:61616" );

// jndi-queue-name, physical-queue-name
prop.put( "queue.test", "mytestqueue" );

try{

    Context jndiContext = new InitialContext(prop);
```

← **Attenzione:**  
Va nel main del "Server" e del "client"

## Caso Point-to-Point:

+ Andiamo ad utilizzare le queue  
• Server (Receiver):

- + File main con configurazione (e set del listener se presente)
- + File thread:
  - Metodo run()
  - Costruttore → a cui si possono passare messaggio, coda, queueConnection

```
QueueConnection queueConn = queueConnFactory.createQueueConnection();
queueConn.start(); //abilita il delivery dei messaggi!

QueueSession queueSession = queueConn.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
QueueReceiver receiver = queueSession.createReceiver(queue);

TextMessage message;

do{
    System.out.println ("In attesa di messaggi!");
    message = (TextMessage)receiver.receive();
    System.out.println (" + messaggio ricevuto: " +
    message.getText());
}while (message.getText().compareTo("fine") != 0);

receiver.close();
queueSession.close();
queueConn.close();
```

+ Listener (Vedere implementazione generale)

## • Client (Sender):

- + Listener

- + Main Client:

- Configurazione

- Creazione Struttura messaggi ed invio.

```
QueueSender sender = queueSession.createSender(queue);
TextMessage message = queueSession.createTextMessage();

for ( int i =0; i<5; i++){
    message.setText("hello_" + i);
    sender.send( message );
}

message.setText("fine");
sender.send( message );

System.out.println ("I messaggi sono stati inviati!");

// clean up risorse
sender.close();
queueSession.close();
queueConn.close();
```

## • Coda:

- + Interfaccia (ICoda e Wrapper)

- + Implementazioni: (fissa la classe CodaCircolare)

- CodaWrapperLock

- CodaWrapperSem

- CodaWrapperSynch

## Caso Publisher - Subscriber:

- + Utilizzeremo i Topic

- Publisher:

- Main con:

- Configurazione

- Creazione del publisher a cui passare il topic

- Creazione e "pubblicazione" dei messaggi

```
TopicConnectionFactory connFactory = (TopicConnectionFactory)
jndiContext.lookup("TopicConnectionFactory");

Topic topic = (Topic)jndiContext.lookup("test");

TopicConnection topicConn = connFactory.createTopicConnection();
TopicSession topicSession = topicConn.createTopicSession(false,
Session.AUTO_ACKNOWLEDGE);

TopicPublisher pub = topicSession.createPublisher(topic);
TextMessage text = topicSession.createTextMessage();

for ( int i =0; i<5; i++){
    text.setText("hello_" + i);
    pub.publish( text );
}

text.setText("fine");
pub.publish( text );

System.out.println ("I messaggi sono stati inviati!");

// clean up risorse
pub.close();
topicSession.close();
```

## • Subcriber:

- + Spetieci figli thread (pensiamo a TExecutor e TManager)
- + Listener
- + Main contenente:
  - o Configurazione
  - o Utilizzo Topic e TopicConnection
  - o Start threads

```
TopicConnectionFactory connFactory = (TopicConnectionFactory) jndiContext.lookup("TopicConnectionFactory");
Topic topic = (Topic) jndiContext.lookup("test");
TopicConnection topicConn = connFactory.createTopicConnection();
topicConn.start();
TopicSession topicSession = topicConn.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
TopicSubscriber sub = topicSession.createSubscriber(topic);
TextMessage msg;
do{
    System.out.println ("In attesa di messaggi!");
    msg = (TextMessage)sub.receive();
    System.out.println (" + messaggio ricevuto: " + msg.getText());
} while (msg.getText().compareTo("fine") != 0);
sub.close();
topicSession.close();
topicConn.close();
```

## • Coda:

- + Interfaccia
- + Coda Circolare

## LISTENER:

Da utilizzare solo in caso di comunicazione asincrona ed ha un costruttore (caso PS in cui gli si passa l'interfaccia Coda) e solo il metodo onMessage (Message m):

```
public class TextMsgListener implements MessageListener {
    public void onMessage ( Message m ){
        try{
            System.out.println (" + messaggio ricevuto: " +
                ((TextMessage)m).getText());
        }catch ( JMSException e ){
            e.printStackTrace();
        }
    }
}
```

## Esempio di Receiver con Listener:

```
QueueReceiver receiver = queueSession.createReceiver(queue);
TextMsgListener msglistener = new TextMsgListener ();
receiver.setMessageListener( msglistener );
```

Ricorda sempre questo ordine:

Dopo la configurazione:

Point - To - Point: Aritoponre Queue

1. Context
2. CommFactory
3. Queue (risposta o richiesta)
4. QueueConnection = commFactory.create...
- 4.1 (Receiver) QueueConnection.start()

5. QueueSession

6. QueueSender o QueueReceiver

Sender:

7. TextMessage

8. Send

Receiver:

7. MyListener

8. Receiver.setMessageListener

Publisher - Subsriber: Aritoponre Topic

1. Context

2. CommFactory

3. Topic

4. Connection

4.1 per il Subsriber: comm.setClientID("DurableService");

5 Session

6 Publisher o Subsriber:

Publisher:

7. Publisher

8. MapMessage

9. Close() varie.

Subsriber

7. DurableSubsriber

8. Listener

9. Start

\* Si usa SetString per  
assegnare una chiave al  
valore da pubblicare mentre  
SetStringProperties quando  
al posto della chiave  
abbiamo una "proprietà" con  
il nome!