Java Message Service (JMS)

Advanced Computer Programming

Prof. Luigi De Simone

Sommario

- JMS e Apache ActiveMQ
- Modelli PTP e PUB-SUB
- Scrittura di applicazioni JMS
  - Entità JMS e modello di programmazione
- Ricezione asincrona
- Messaggi e Sessioni
- Aspetti avanzati

Riferimenti:

Java Message Service API Tutorial – Kim Haase (Capp. 1-5) disponibile in versione on-line e PDF al link

http://docs.oracle.com/javaee/1.3/jms/tutorial/

JMS (Java Message Service)

- Definizione dalla specifica **Java Messagge Service (JMS)** (JSR 914):

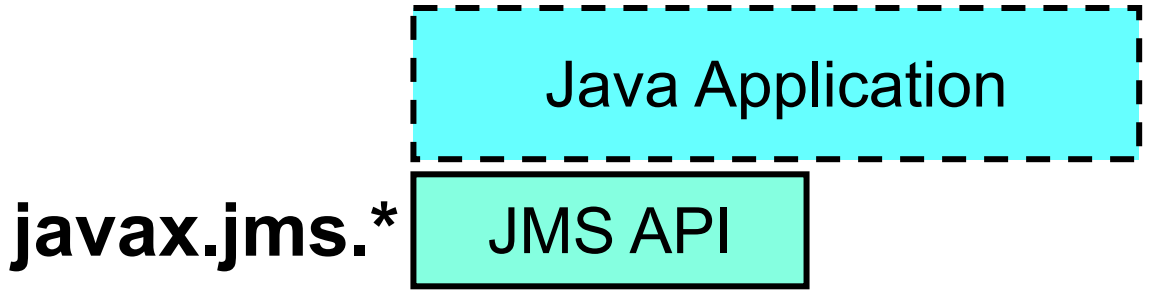
*"JMS is a set of **interfaces** and associated semantics **that define how** a JMS client accesses the facilities of an enterprise messaging product"*

- **JMS definisce uno standard** che consente ai programmi Java di **inviare e ricevere messaggi tramite un prodotto MOM**.
- I programmi scritti con JMS potranno essere eseguiti *con un qualsiasi MOM* che implementi le interfacce JMS:
  - i prodotti MOM che intendo fornire le funzionalità JMS devono implementare le interfacce definite dallo standard JMS.

JMS (Java Message Service)

- La specifica e le API JMS* (versione 1.1) sono scaricabili da ("*Download the version 1.1 API Documentation, Jar and Source*"):

  http://www.oracle.com/technetwork/java/docs-136352.html



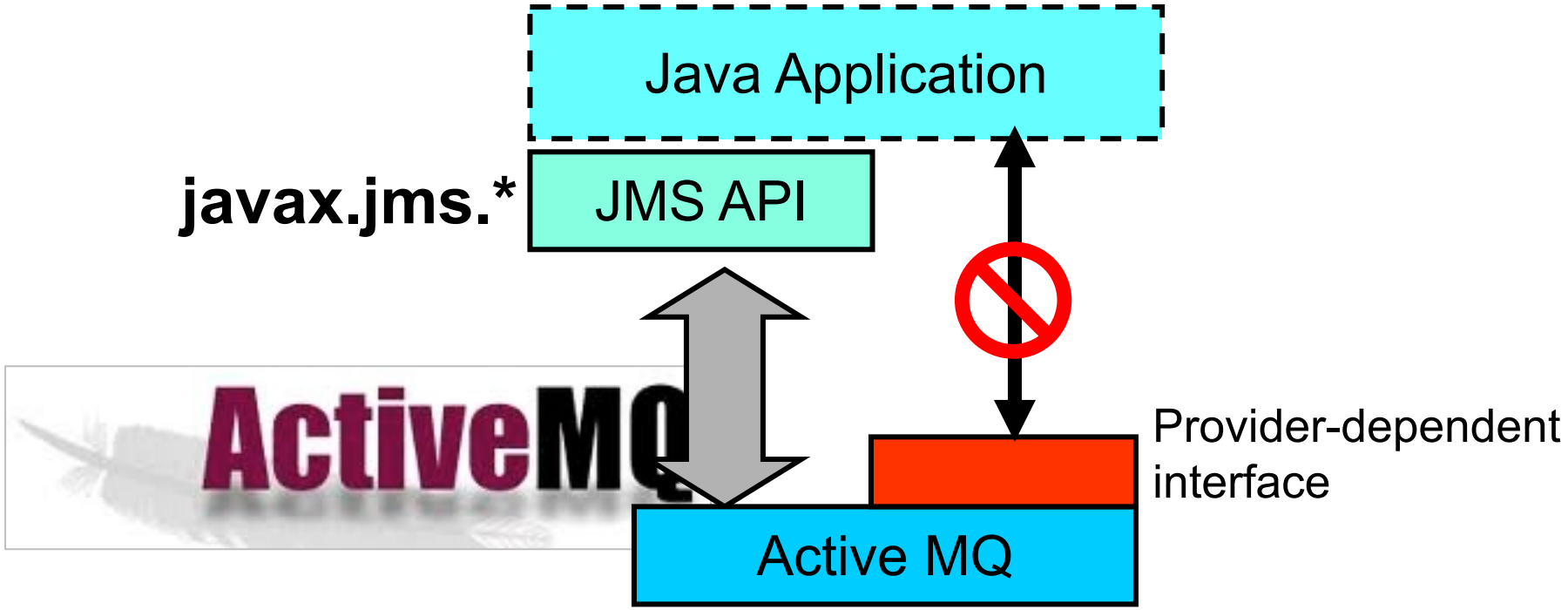
*NOTA: **è comunque necessario installare un provider** che implementi lo standard JMS.

Il provider: Apache ActiveMQ

- Il provider JMS utilizzato durante il corso è Apache ActiveMQ, scaricabile da:

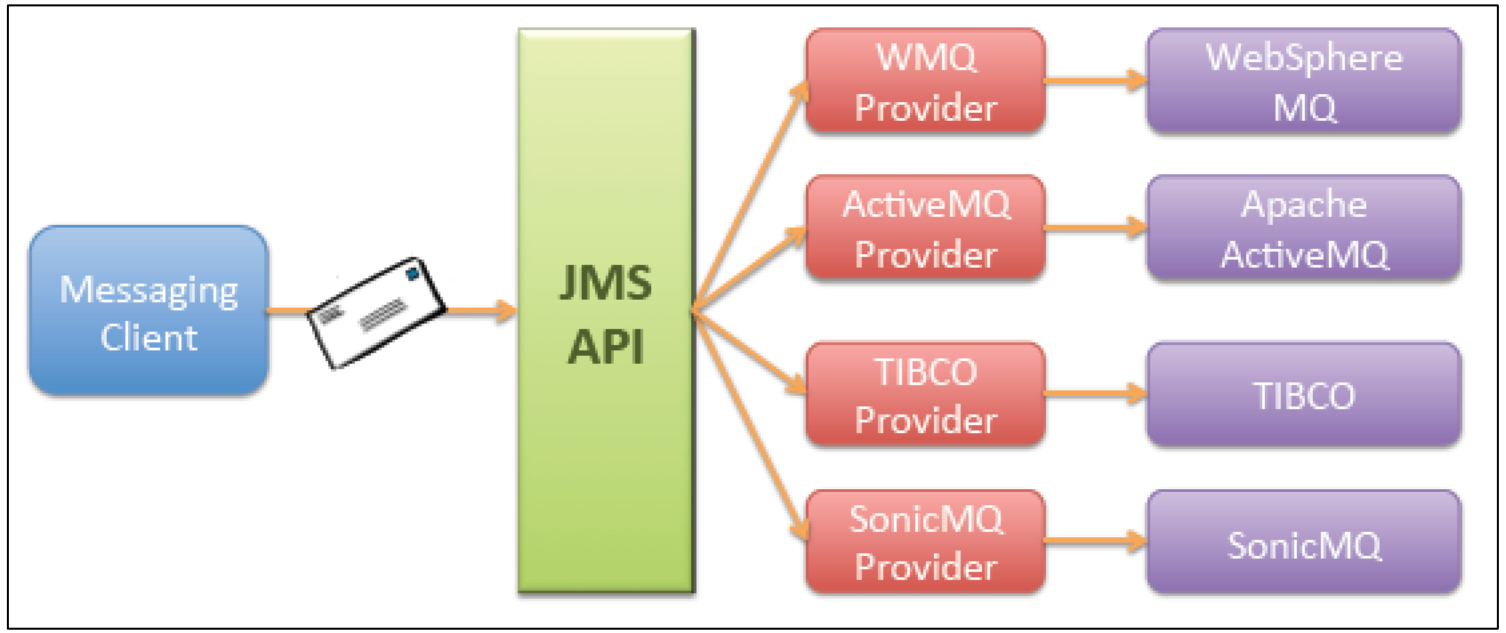
  http://activemq.apache.org/components/classic/download/

- NOTA: scaricare la versione di Apache ActiveMQ che supporta la versione Java installata!



JMS e provider

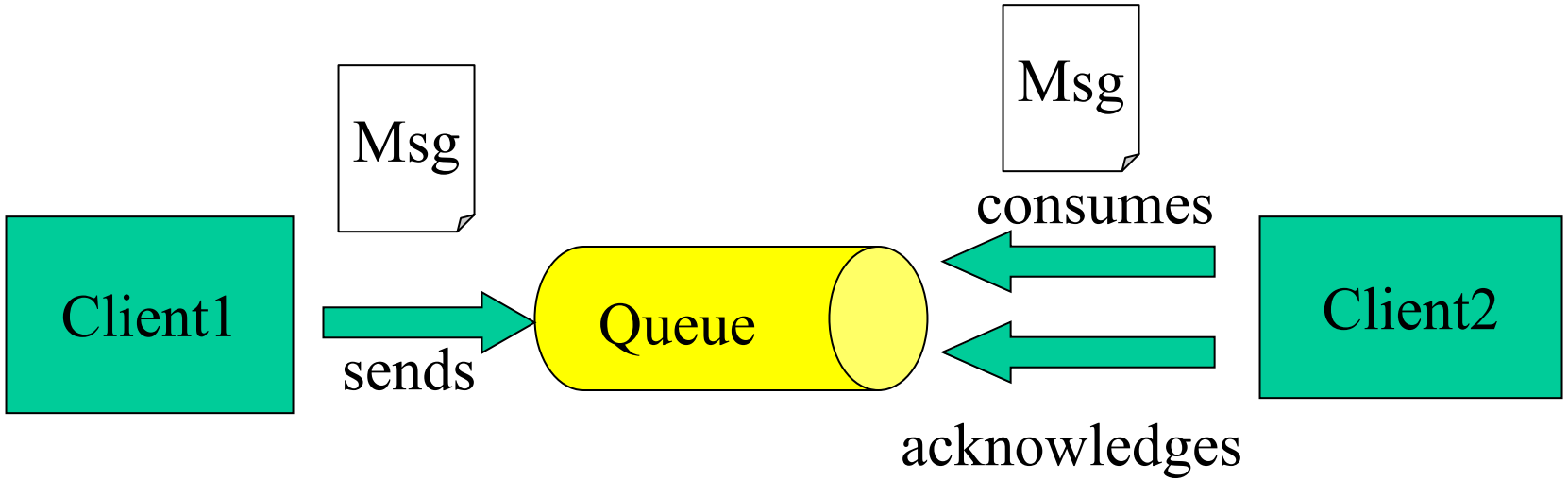
- La specifica JMS consente il **disaccoppiamento tra i client** (sender e receiver di messaggi) e la specifica **implementazione definita del provider**



Richiamo: Domini di Messaging JMS

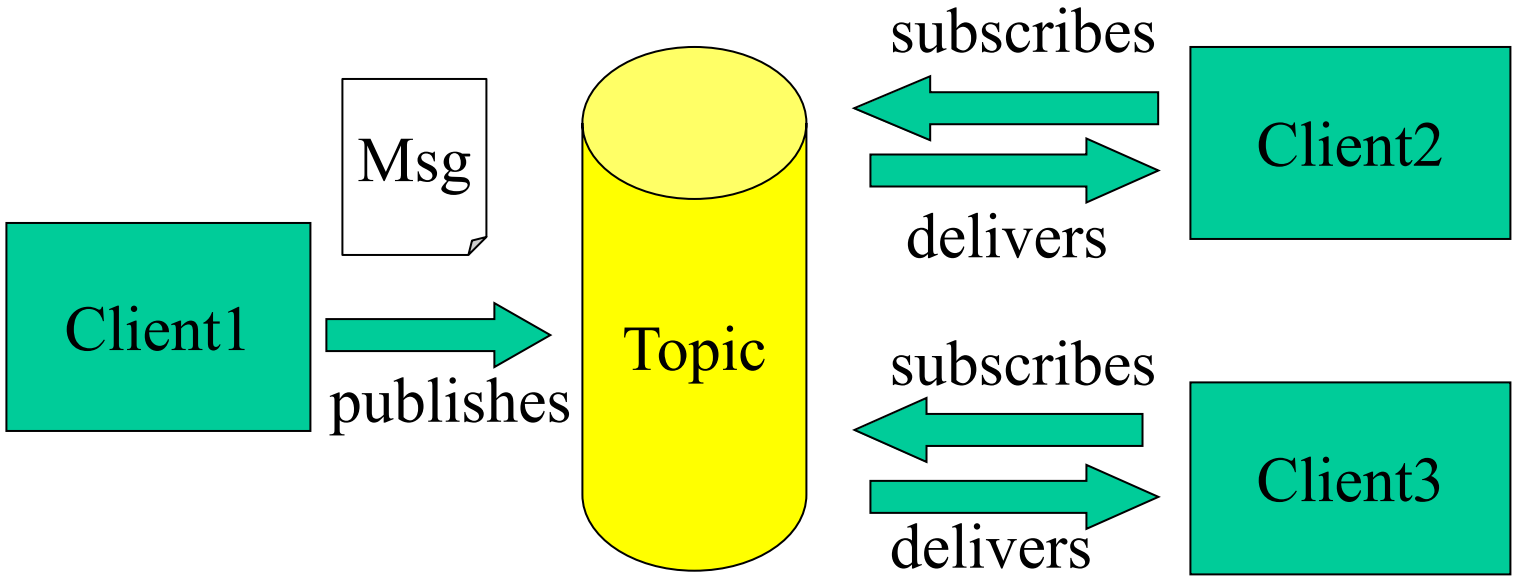
- **Point-to-Point (PTP)**
  - Il dominio di messaging PTP ruota intorno al concetto di **coda di messaggi**;
  - *ogni messaggio ha un solo consumer*
- **Publish-Subscribe**
  - ogni messaggio è associato ad un **topic**;
  - utilizza il concetto di *topic* per l'invio e la ricezione dei messaggi;
  - *ogni messaggio può avere più consumer*

Richiamo: Messaging Point-to-Point



- Una coda **conserva** un determinato **messaggio** finché esso ***non scade*** o ***viene consumato***;
- Il **receiver conferma (*ack*)** la corretta **ricezione** del messaggio.

Richiamo: Messaging Publish/Subscribe



- I ***topic*** **conservano un messaggio** finché esso **non viene rilasciato ai subscriber *correnti***.
- I **messaggi** possono essere **processati** da **0..N subscriber**.

Interfacce JMS

- JMS definisce un insieme di **interfacce** di alto livello che consentono l'accesso ai domini di messaging PTP e PUB/SUB.
- **`ConnectionFactory`:** necessaria per la creazione di una Connection:
  - `QueueConnectionFactory` o `TopicConnectionFactory`;
- **`Destination`:** incapsula la destinazione, ovvero "dove" i messaggi saranno inviati, o "da cui" i messaggi sono ricevuti:
  - `Coda` o `Topic`
- Istanze di `ConnectionFactory` e `Destination` prendono il nome di **administered object**.

Interfacce JMS

- **`Connection`**: connessione verso il provider JMS;
- **`Session`:** contesto *single-threaded* per inviare o ricevere messaggi:
- **`MessageProducer`**: utilizzato per inviare messaggi;
- **`MessageConsumer`**: utilizzato per ricevere messaggi.

Interfacce JMS

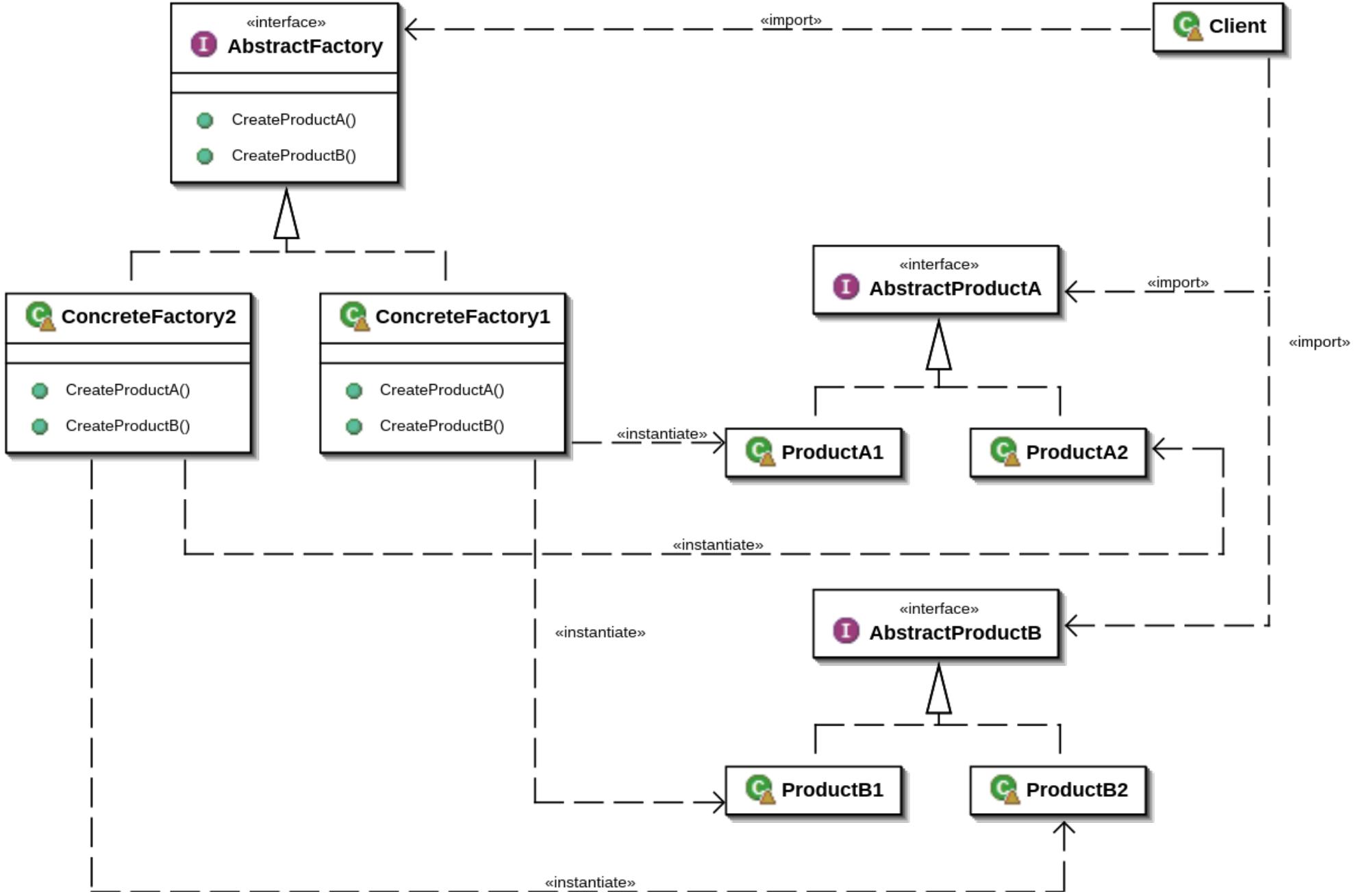
- Le interfacce di alto livello citate in precedenza, sono specializzate a seconda dello specifico dominio di *messaging*.

High-level Interface	PTP Domain	Pub/Sub Domain
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

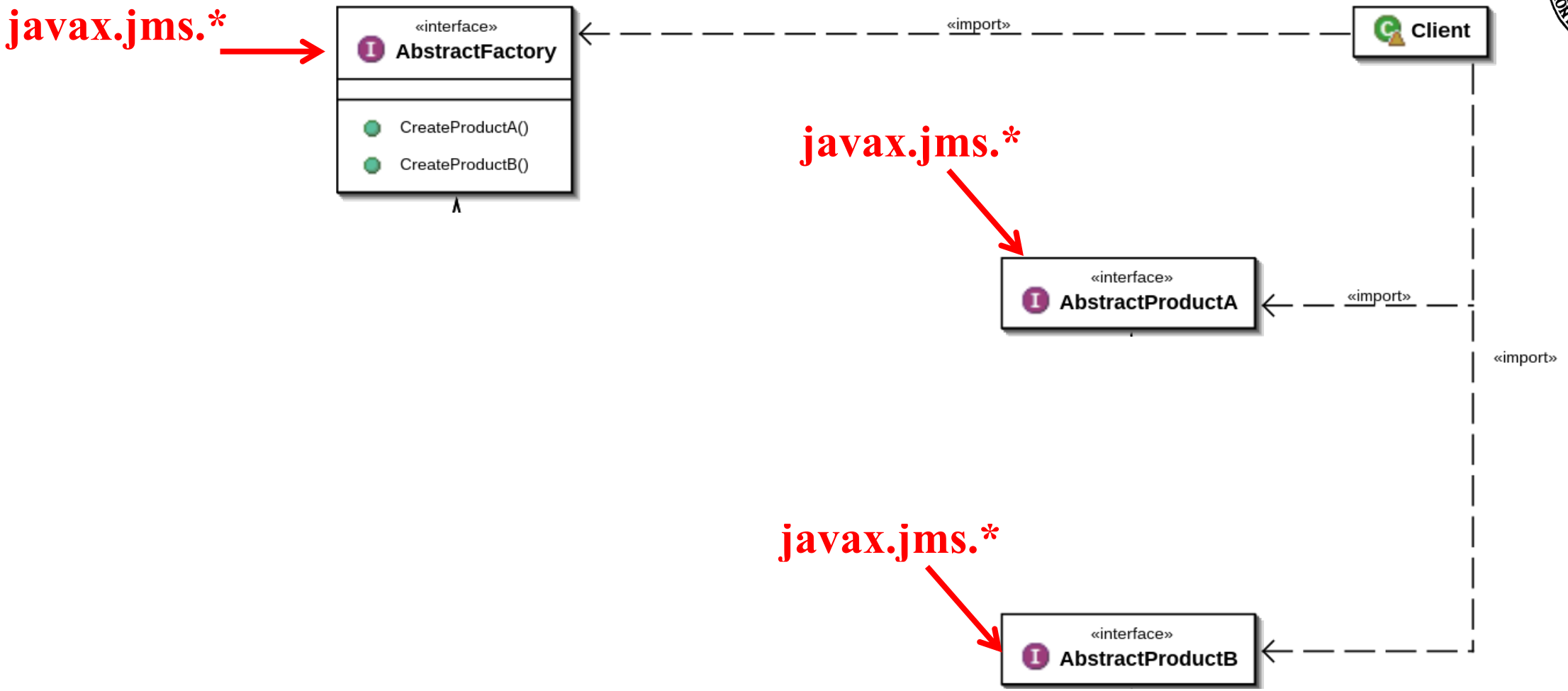
JMS e il pattern Abstract Factory

- Il pattern **Abstract Factory** è utilizzato quando:
  - un sistema deve essere **indipendente da come i suoi prodotti sono creati o realizzati**;
  - un sistema deve poter essere configurato con **una di tante famiglie di prodotti**;
  - si vuole fornire una libreria di prodotti e **rivelarne l'interfaccia**, **ma non l'implementazione.**
- Per esempio, `QueueConnectionFactory`, `QueueConnection`, `Queue`, `QueueSession` (analogamente per `Topic`*) sono interfacce che punteranno **ad implementazioni di classi fornite dal provider**.

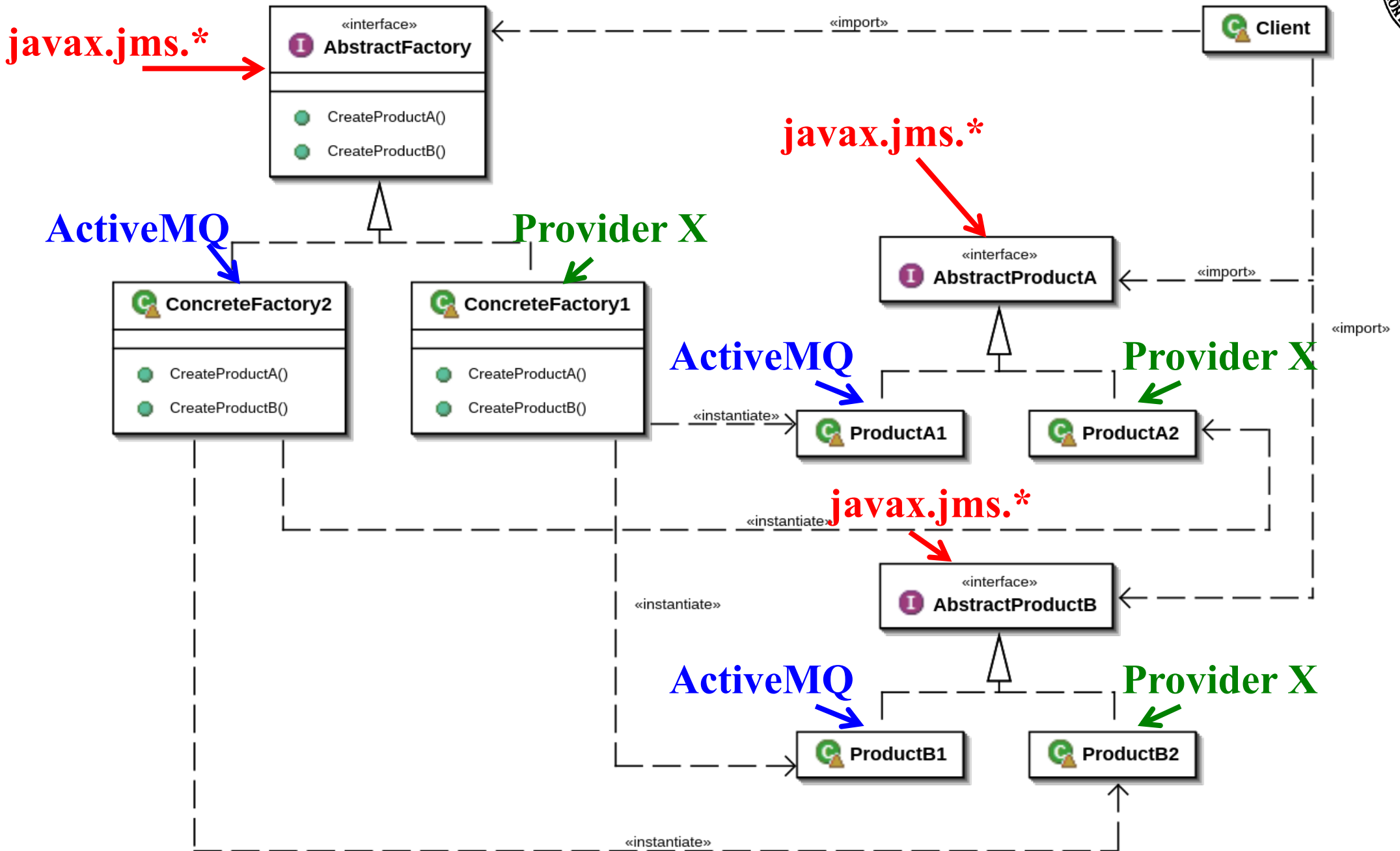
Abstract Factory



JMS e il pattern Abstract Factory



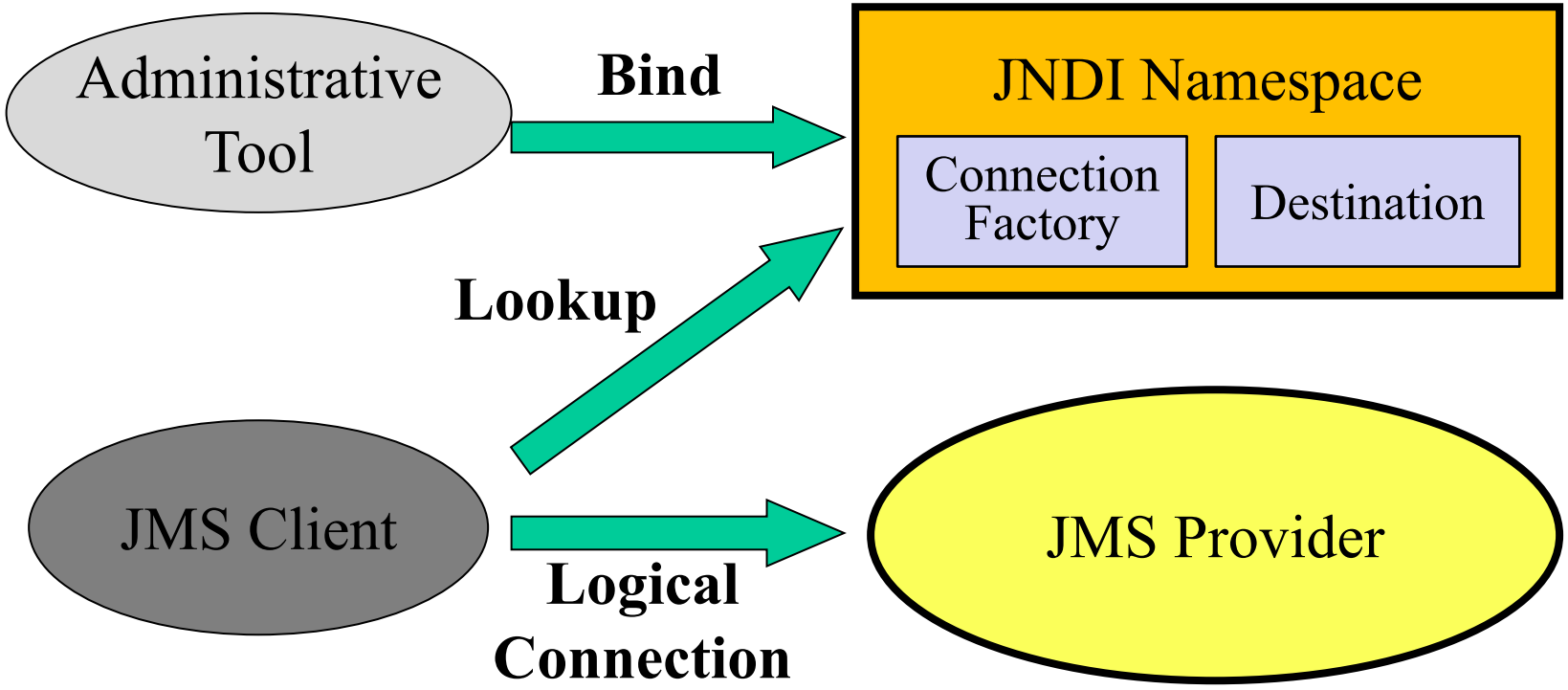
JMS e il pattern Abstract Factory



Entità di un sistema JMS

- **Client:** programmi JAVA che inviano/ricevono messaggi;
- **Provider**: realizzazione "concreta" del sistema di messaging; implementa la specifica JMS, supporta lo scambio messaggi e fornisce funzionalità per la gestione/supervisione del sistema di messaging.
- **Messaggi:** astrazioni delle informazioni scambiate tra entità che usano JMS
- **Administered Objects:** istanze di `ConnectionFactory` e `Destination`
  - oggetti JMS *pre-configurati* che **colmano il gap** tra le interfacce JMS e la **specifica tecnologia** del provider;
  - il provider JMS fornisce i programmi per creare gli **administered objects** e registrarli in **JNDI** (Java Naming and Directory Interface);

Entità di un sistema JMS

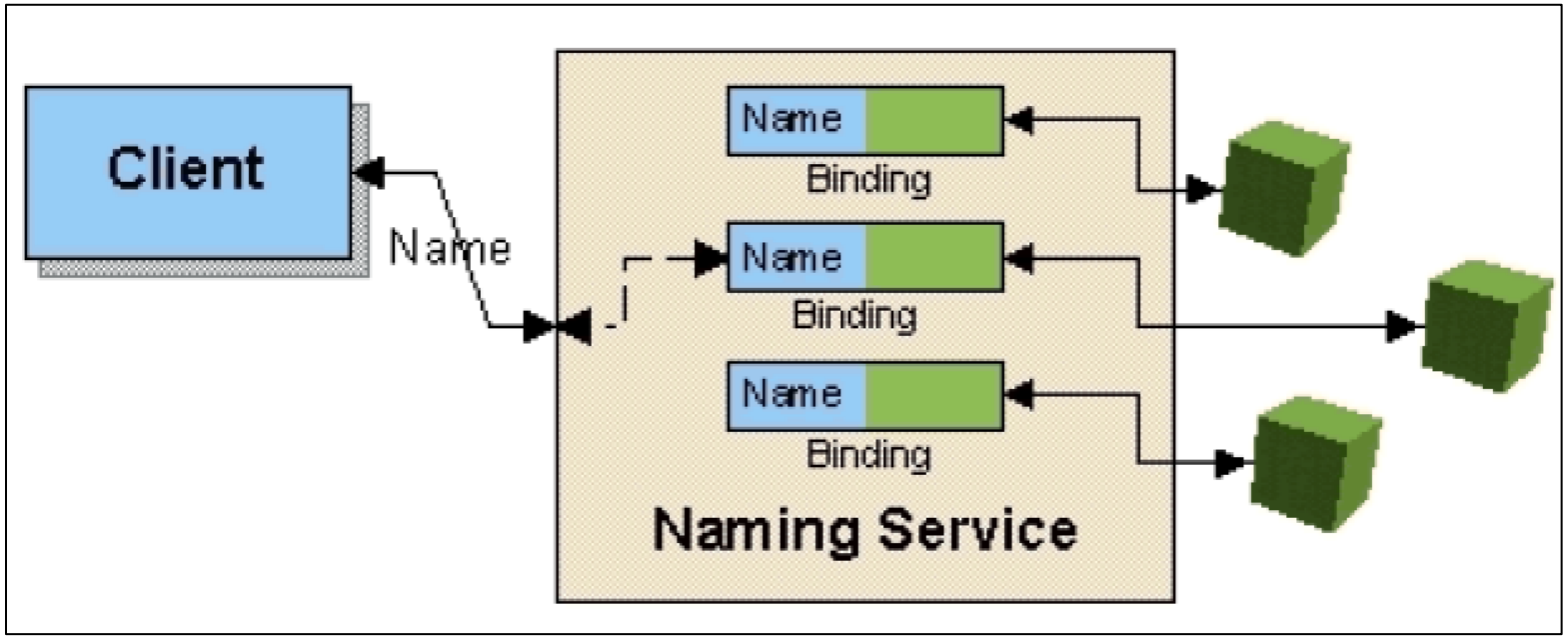


Administered objects

- Gli *administered objects* sono recuperati dai **client JMS** tramite **JNDI** (operazione di **lookup**);
- Esistono **due tipologie** di *administered objects*:
  - **`ConnectionFactory`**
    - creazione di una Connection per il *messaging* (**`QueueConnectionFactory`** e **`TopicConnectionFactory`**)
  - **`Destination`**
    - dove i messaggi saranno inviati, o da cui i messaggi sono ricevuti (`Queue` e `Topic`)
- Sebbene gli *administered objects* siano istanze di classi che incapsulano dettagli dipendenti dal provider, essi sono:
  - recuperati tramite un meccanismo standard (JNDI);
  - acceduti tramite interfacce standard JMS;
- Il programma JMS deve conoscere il **nome JNDI** e l'**interfaccia JMS degli administered objects**, ma **non** i dettagli dipendenti dal provider.

Naming service e JNDI

- Un **naming service** mantiene un insiemi di *binding* tra nomi e riferimenti a oggetti (o direttamente a oggetti).
- ***Java Naming and Directory Interface (JNDI)*** è un'interfaccia che supporta le funzionalità comuni di un *naming service*.

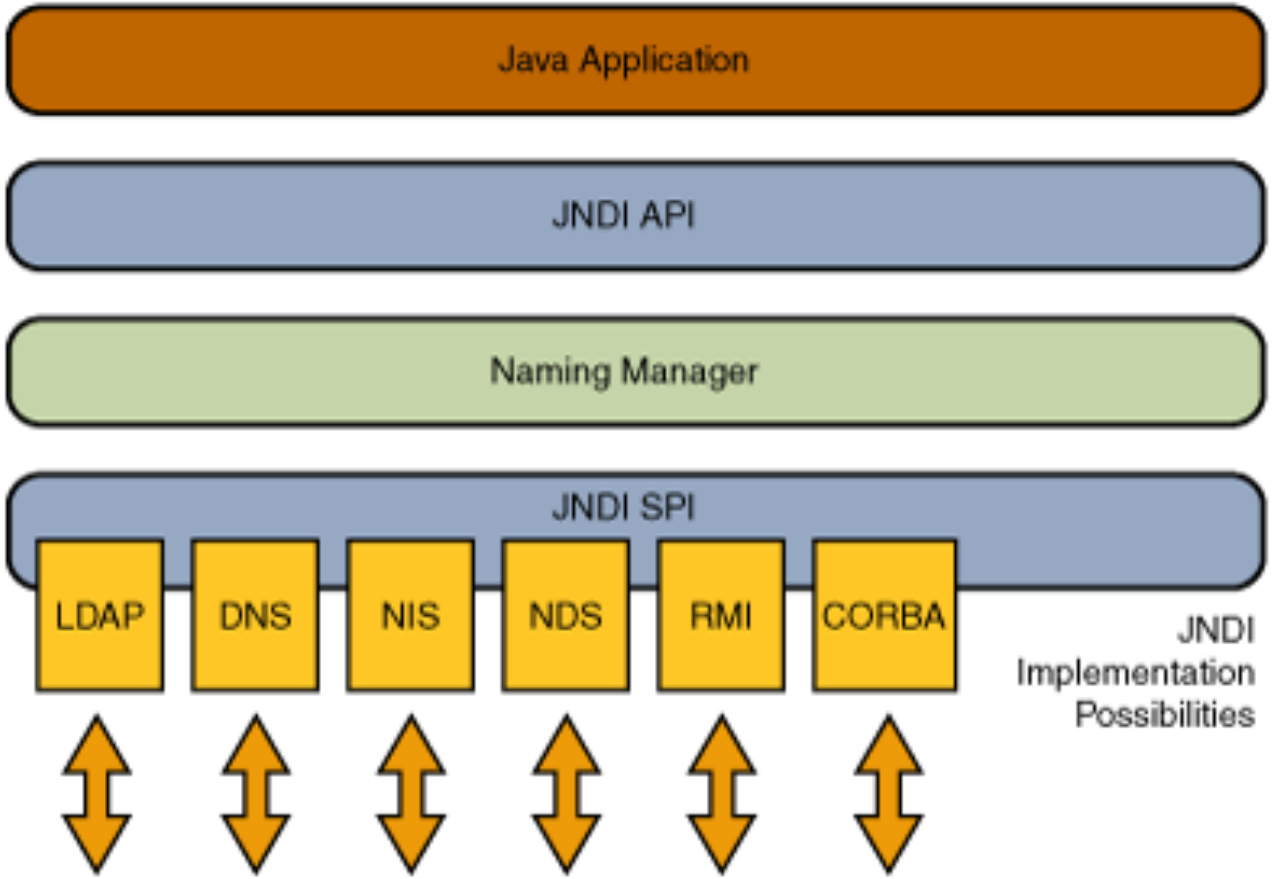


Alcuni concetti

- **Name**: nome dato all'oggetto registrato;
- **Binding**: associazione nome-oggetto;
- **Reference**: puntatore a un oggetto;
- **Context**: insieme di associazioni nome-oggetto.

Naming service e JNDI

- JNDI è **indipendente** dallo specifico servizio di naming
  - il servizio di naming è acceduto tramite **plugin** chiamati **Service Provider**



Utilizzare JNDI

- Un client individua il provider di un servizio JNDI configurando le seguenti **proprietà**:
  - **`java.naming.factory.initial`**
  - **`java.naming.provider.url`**
- Si utilizza **`Context`** che è un'**interfaccia** che specifica i metodi per *aggiungere*, *cancellare*, *cercare* **oggetti** tramite un servizio JNDI:
  - rappresenta un insieme di binding;
  - tutte le operazioni JNDI sono svolte in relazione a un **`Context`**.
- **`InitialContext`** è un'implementazione di **`Context`**.

Metodi principali di Context

- `void bind (String stringName, Object object)`
  - il nome non deve essere associato già ad alcun oggetto.
- `void rebind (String stringName, Object object)`
- **`Object lookup(String stringName)`**
- `void unbind(String stringName)`
- `void rename (String stringName, String newName)`

Modello di programmazione generico JMS

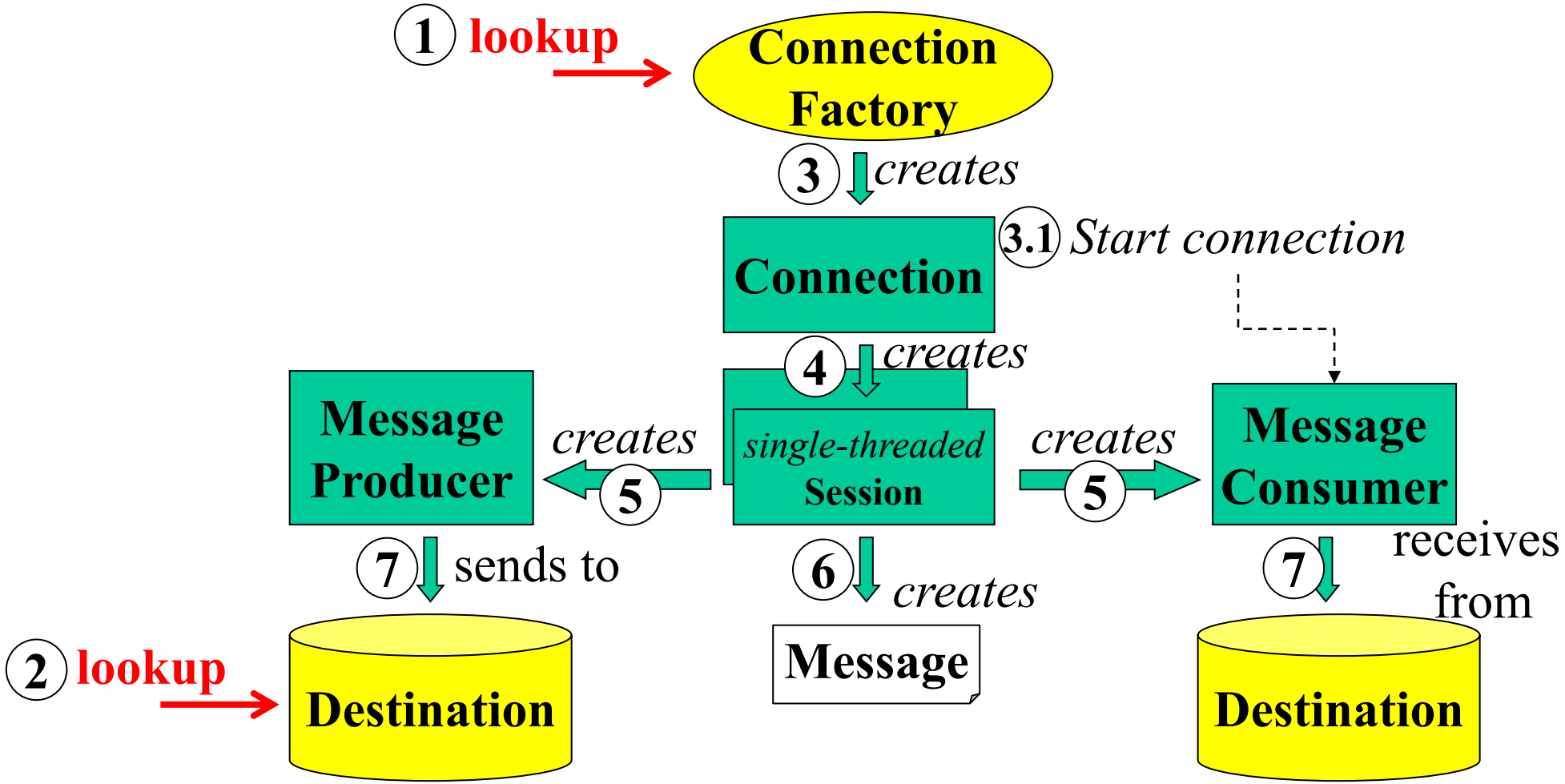
- Lo sviluppo di un **client JMS** prevede i seguenti passaggi:
  1. **lookup** di una **`ConnectionFactory`** da JNDI
  2. **lookup** di una **`Destination`** da JNDI
  3. utilizzo della `ConnectionFactory` per la creazione di una **`Connection`**
     3.1. Devo avviare la **`Connection`** se voglio abilitare la ricezione dei messaggi
  4. utilizzo della **`Connection`** per la creazione di una (o più) **`Session`**
  5. utilizzo di una **`Session`** e della **`Destination`** per la creazione dei **`MessageProducer`** e **`MessageConsumer`**
  6. utilizzo di una **`Session`** per la creazione di un **`Message`**
  7. Invio e ricevo messaggi
  8. *cleanup* delle risorse

- Prima dell'avvio di un client JMS è **necessario avviare il provider**:
  - l'avvio del provider consente l'attivazione del servizio di naming JNDI da cui il client potrà recuperare **`ConnectionFactory`** e **`Destination`**.

Modello di programmazione generico JMS

- Una **`Connection`** rappresenta una connessione tra il client JMS ed il provider;
- Una **`Destination`** incapsula l'identità della **destinazione** di un messaggio:
  - il "dove" il messaggio sarà inviato (Queue o Topic)
- Una **`Session`** è un contesto *single-threaded* per l'**invio** e la **ricezione** di un messaggio
  - *le Sessioni **non sono pensate** per l'utilizzo concorrente da parte di più thread.*
- I **`MessageProducer`** e **`MessageConsumer`**, inviano e ricevono messaggi nel contesto di una `Session`.

Modello di programmazione generico JMS



Lookup degli administered objects

- Per effettuare il **lookup** degli *administered objects*, è necessario configurare il client per accedere al servizio JNDI di ActiveMQ.

```
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);
        // omesso //
        // slide successiva …
```

Esempio (dominio PTP)

- Tramite le operazioni di lookup, è quindi possibile **ottenere gli administered objects**:

```
// 1. Lookup administered objects, ConnectionFactory e Destination per queue
QueueConnectionFactory queueConnFactory =
(QueueConnectionFactory)jndiContext.lookup("QueueConnectionFactory");

// 2. Lookup administered objects, Destination per queue
Queue queue = (Queue)jndiContext.lookup("test"); // la mia Destination è Queue
//il prefisso "queue." non fa parte del nome jndi

// 3. Creo la Connection
QueueConnection queueConn = queueConnFactory.createQueueConnection();

// 4. Creo la Session (single-threaded)
QueueSession queueSession = queueConn.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
```

Esempio Sender (dominio PTP)

- Creazione sender ed invio dei messaggi

```
// 5. Creo il sender (Message Produce)
QueueSender sender = queueSession.createSender(queue);

// 6. Creo il messaggio che userò per l'invio
TextMessage message = queueSession.createTextMessage();

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

        // 7. Invio il messaggio (send asincrona)
        sender.send( message );
}

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

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

// 8. clean up risorse
sender.close();
queueSession.close();
queueConn.close();
```

Esempio Receiver (dominio PTP)

- Il receiver è **identico** per le fasi di **lookup** e **creazione** connessione/sessione, tranne per l'avvio di una connessione (vedi step 3.1)
- **Creazione di un receiver**

```
// 3. Creo la Connection
QueueConnection queueConn = queueConnFactory.createQueueConnection();

// 3.1. Avvio la connection
queueConn.start();  //NOTE: abilita il delivery dei messaggi!

// 4. Creo la Session (single-threaded)
QueueSession queueSession = queueConn.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);

// next slide ...
```

Esempio Receiver (dominio PTP)

- Il receiver è **identico** per le fasi di **lookup** e **creazione** connessione/sessione, tranne per l'avvio di una connessione (vedi step 3.1)
- **Creazione di un receiver**

```
// 5. Creo il receiver (Message Consumer)
QueueReceiver receiver = queueSession.createReceiver(queue);

// 6. Creo il messaggio che userò per la ricezione
TextMessage message;
do{
        System.out.println ("In attesa di messaggi!");
        // 7. Ricezione (bloccante)
        message = (TextMessage)receiver.receive();
        System.out.println ("      + messaggio ricevuto: " +
        message.getText());
}while (message.getText().compareTo("fine") != 0);

// 8. clean up risorse
receiver.close();
queueSession.close();
queueConn.close();
```

Esempio Publisher (dominio Pub-Sub)

```
// 1. Lookup administered objects, ConnectionFactory per TOPIC
TopicConnectionFactory connFactory = (TopicConnectionFactory)
jndiContext.lookup("TopicConnectionFactory");

// 2. Lookup administered objects, Destination per TOPIC
Topic topic = (Topic)jndiContext.lookup("test");

// 3. Creo la Connection
TopicConnection topicConn = connFactory.createTopicConnection();

// 4. Creo la Session (single-threaded)
TopicSession topicSession = topicConn.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);

// 5. Creo il publisher (Message Producer)
TopicPublisher pub = topicSession.createPublisher(topic);

// 6. Creo il messaggio che userò per la pubblicazione
TextMessage text = topicSession.createTextMessage();

for ( int i =0; i<5; i++){
          text.setText("hello_" + i);
          // 7. Pubblico il messaggio
          pub.publish( text );
}

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

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

// 8. clean up risorse
pub.close();
topicSession.close();
topicConn.close();
```

Esempio Subscriber (dominio Pub-Sub)

```
// 1. Lookup administered objects, ConnectionFactory per TOPIC
TopicConnectionFactory connFactory = (TopicConnectionFactory)
jndiContext.lookup("TopicConnectionFactory");

// 2. Lookup administered objects, Destination per TOPIC
Topic topic = (Topic)jndiContext.lookup("test");

// 3. Creo la Connection
TopicConnection topicConn = connFactory.createTopicConnection();

// 3.1 Avvio la Connection (sono un Message Consumer)
topicConn.start();

// 4. Creo la Session (single-threaded)
TopicSession topicSession = topicConn.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);

// 5. Creo il subscriber (Message Consumer)
TopicSubscriber sub = topicSession.createSubscriber(topic);

// 6. Creo il messaggio che userò per la sottoscrizione
TextMessage msg;

do{
        System.out.println ("In attesa di messaggi!");
        // 7. Sottoscrivo (ricevo) il messaggio
        msg = (TextMessage)sub.receive();
        System.out.println ("       + messaggio ricevuto: " + msg.getText());
}while (msg.getText().compareTo("fine") != 0);

// 8. clean up risorse
sub.close();
topicSession.close();
topicConn.close();
```

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 “ ; “ per i sistemi Windows) 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 VS Code/VS Codium, come *referenced library*

Avviare applicazione di esempio JMS

```
// start activemq daemon in terminal
# cd /PATH/TO/ActiveMQ/bin
// ./activemq start to start activemq as background process
# ./activemq start

// ./activemq stop to terminate daemon
// ./activemq status to get current daemon status
// ./activemq console to start activemq as foreground process
```

Avviare applicazione di esempio JMS

```
// start the receiver (new terminal)
# cd /PATH/TO/APPLICATION/bin
// NOTE: il : finale è solo per SO Unix-based (Linux, Mac OSX, etc.)
# java -cp /PATH/TO/ActiveMQ/activemq-all-X.XX.X.jar:  queueapp.Receiver
In attesa di messaggi!
       + messaggio ricevuto: hello_0
In attesa di messaggi!
. . .

// start the sender (new terminal)
# cd /PATH/TO/APPLICATION/bin
# java -cp /PATH/TO/ActiveMQ/activemq-all-X.XX.X.jar:  queueapp.Sender
I messaggi sono stati inviati!
```

Consumo dei messaggi

- I messaggi JMS possono essere **consumati in due modalità**.
- **Sincrona:**
  - un *receiver* o un *subscriber* prelevano esplicitamente un messaggio dalla destinazione utilizzando il metodo **receive**;
  - la ***receive* si blocca** finché arriva un messaggio, oppure allo scadere di un timeout.
- **Asincrona**:
  - Il client JMS registra un **`MessageListener`** su un consumer;
  - ogni volta che un messaggio arriva a destinazione, il provider JMS invoca il metodo **`onMessage`** del listener.

Receive sincrona

- I messaggi JMS sono ricevuti in modalità sincrona utilizzando uno dei seguenti metodi:
  - **`receive()`**: si blocca finché non c'è un messaggio disponibile;
  - **`receive( long timeout ):`** riceve un messaggio che arriva prima dello scadere del timeout specificato;
  - **`receiveNoWait():`** riceve il messaggio se immediatamente disponibile. Il comportamento di questo metodo equivale a quello di una *receive* con timeout molto piccolo.

Receive asincrona e MessageListener

- Il *listener* è un'istanza di una classe che implementa l'interfaccia JMS **`MessageListener`**:
  - contiene **un solo metodo** denominato **`onMessage`**;
  - nel metodo **`onMessage`** viene scritto il codice che elabora il messaggio al suo arrivo.

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

Receive asincrona e MessageListener

- Il client JMS istanzia il listener e lo registra sul receiver
- Esempio:

```
QueueReceiver receiver = queueSession.createReceiver(queue);

TextMsgListener  msglistener = new TextMsgListener ();

receiver.setMessageListener( msglistener );
```

- Sia il dominio **PTP** che quello **PUB/SUB supportano** la ricezione **asincrona** dei messaggi.

Messaggi JMS

- Tutti i messaggi sono **strutturati** nel seguente modo:

- I **campi header** consentono l'invio e l'identificazione dei messaggi
- Esempi di campi *header*:
  - ***JMSMessageID***: identificativo univoco del messaggio;
  - ***JMSCorrelationID***: identificativo che collega un messaggio ad un altro (per esempio, collegare una richiesta a una risposta);
  - ***JMSReplyTo***: contiene una Destination, definita dal Client, verso cui deve essere inviato un eventuale messaggio di risposta;
  - ***JMSPriority***: contiene la priorità del messaggio (da 0 a 9 –la più alta-);
  - Si accede ai campi header con i metodi del tipo **getJMS*** e **setJMS***

Messaggi JMS

- Sezione ***Properties***:
  - può contenere proprietà specifiche dell'applicazione, proprietà standard JMS, o proprietà del vendor JMS;
  - le **properties** sono coppie *<String, value>*, dove *value* è un tipo built-in Java o una *String*;
  - Una *property* si gestisce con i metodi **set*Property** e **get*Property**;
  - per un messaggio *ricevuto*, le proprietà sono in modalità read-only;
  - il metodo **getPropertyNames ()** restituisce l'elenco di tutti i nomi delle proprietà;
  - supportano una sintassi *SQL-like* per selezionare i messaggi:
    - **createSubscriber (topic, "prop1 > 6 AND prop2 = 'test' ");**
- Sezione ***Body***:
  - **Il contenuto vero e proprio del messaggio**, gestito con i metodi previsti dalla specifica tipologia di *body* (slide seguente).

Messaggi JMS

- JMS supporta diverse tipologie di messaggi (**parte body**):

Message Type	Contains	Some Methods
TextMessage	String	getText, setText
MapMessage	Set of name/value pairs	setString, setDouble, setLong, getDouble, getString
BytesMessage	Stream of uninterpreted bytes	writeBytes, readBytes
StreamMessage	Stream of primitive values	writeString, writeDouble, writeLong, readString
ObjectMessage	Serializable object	setObject, getObject

Acknowledgement

```
QueueSession queueSession = queueConn.
        createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
```

- Il **corretto** consumo di un messaggio avviene in 3 fasi:
  - il client *riceve* il messaggio;
  - il client *processa* il messaggio;
  - il messaggio è **acknowledged.**
- L'acknowledgement può essere effettuato dal ***client*** o dal ***provider JMS***, a seconda di come è configurata la sessione:
  - la **modalità di acknowledgement** è specificata dal **secondo parametro della createSession**.

Acknowledgement

- **AUTO_ACKNOWLEDGE**: *la sessione conferma la corretta ricezione del messaggio* quando il client ritorna correttamente da una **receive** (o quando il MessageListener termina la propria esecuzione):
  - nel caso AUTO_ACKNOWLEDGE con **receive sincrona** la *ricezione* e l'*ackowledgement* avvengono in una fase (seguita poi dal processing).
- **CLIENT_ACKNOWLEDGE**: il client conferma la ricezione del messaggio *invocando il metodo acknowledge sul messaggio*. La conferma avviene a livello di sessione:
  - l'ack di un messaggio consumato conferma automaticamente tutti i messaggi consumati nella sessione.

Acknowledgement

- **DUPS_OK_ACKOWLEDGE**: una forma di ackowledgement *lasco* che, in caso di malfunzionamento nel provider JMS, potrebbe generare messaggi **duplicati:**
  - per esempio, la sessione invia un messaggio di acknowledgement ogni N messaggi o ad intervalli di tempo prefissati.

Sessioni Transacted

```
QueueSession queueSession =
        queueConn.createQueueSession(true,0);
```

- E' possibile raggruppare una serie di operazioni JMS (send/receive) in **transazioni**.
  - quando la sessione è *transacted*, il secondo parametro (quello relativo alla modalità di ACKNOWLEDGEMENT) è ignorato (può essere impostato a 0).
  - una transazione ha significato nel contesto di una **singola sessione**
  - Una transazione termina quando l'applicazione invoca il metodo di **`commit()`** sulla sessione (se tutto è andato bene), oppure con un operazione di **`rollback()`** se si è verificata qualche anomalia

```
try {
   // per es., sequenza di operazioni send
   session.commit();
} catch(Exception e){
   session.rollback();
}
```

Sessioni Transacted

- Quando una transazione è **`commit()`**
  - tutti i messaggi **inviati** nell'ambito della transazione **diventano disponibili per la consegna** ad altre applicazioni
  - tutti i messaggi **ricevuti** nell'ambito della transazione vengono ***acknowledged*** in modo che JMS non tenti di consegnarli nuovamente all'applicazione
    - In una comunicazione PTP, JMS *rimuove anche i messaggi ricevuti dalle loro code*
- Quando una transazione è **`rollback()`**
  - tutti i messaggi **inviati** all'interno della transazione **vengono scartati da JMS**
  - tutti i messaggi **ricevuti** all'interno della transazione diventano **nuovamente disponibili per l'invio**
    - In una comunicazione PTP, **i messaggi ricevuti vengono rimessi in coda** e tornano visibili alle altre applicazioni.

Sessioni JMS e Concorrenza

- La specifica JMS prevede che sia **gestito l'accesso concorrente** su oggetti di tipo `Destination`, `ConnectionFactory` e `Connection`.
- Gli oggetti `Session`, `MessageProducer` e `MessageConsumer` **non sono progettati** per essere utilizzati in contesto multithread:
  - le Session supportano le **transazioni** per le quali è complesso implementare un supporto multithread;
  - la ricezione asincrona multithread non è supportata.

Persistenza dei messaggi

- JMS consente di specificare **due modalità di delivery**, che indicano se i messaggi debbano essere persi o no a causa di un malfunzionamento del provider JMS.
- La **modalità PERSISTENT** assicura che i messaggi non siano persi in caso di malfunzionamento del provider JMS:
  - Ogni messaggio è registrato su memoria stabile;
  - PERSISTENT è la modalità di **default**.
- Nella **modalità NON_PERSISTENT**, il provider non garantisce la memorizzazione del messaggio in caso di malfunzionamenti.

Persistenza dei messaggi

- La proprietà di **persistenza** può essere **specificata in due modi**
- Impostando la modalità tramite il metodo `setDeliveryMethod`

```
producer.setDeliveryMethod(DeliveryMode.NON_PERSISTENT);
```

- Specificando la modalità durante l'invio (**`send`**) del messaggio

```
producer.send(msg, DeliveryMode.NON_PERSISTENT, 3, 1000);
```

  - Nell'esempio, si specifica il *delivery mode* (ossia, NON_PERSISTENT oppure PERSISTENT), la priorità del messaggio, ed il *time-to-live* del messaggio.

Per maggiori dettagli, consultare
https://docs.oracle.com/javaee/7/api/javax/jms/QueueSender.html

Durable Subscription

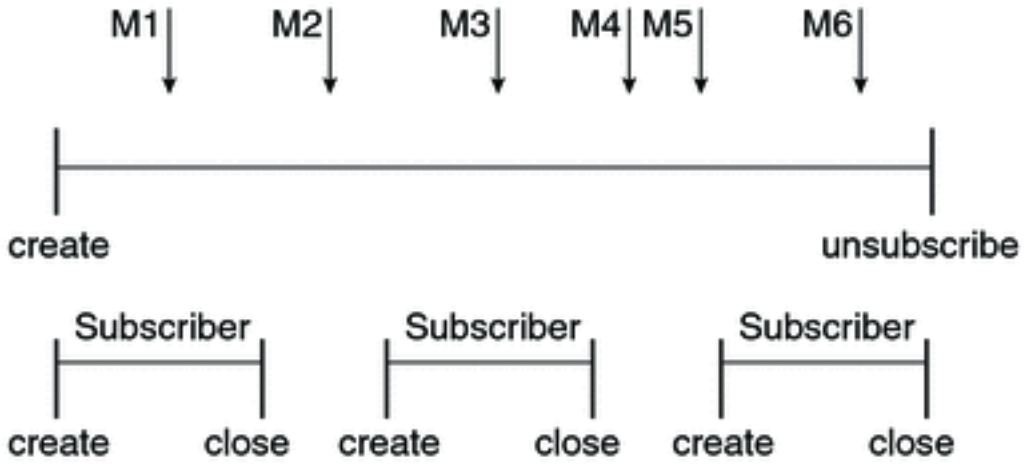
- Di default, un *subscriber* riceve i messaggi solo **se pubblicati quando esso è attivo**:
  - Si dice che il metodo **createSubscriber** crea un subscriber **non-duraturo (non-durable)**
- Il metodo **createDurableSubscriber** disaccoppia invece i concetti di *sottoscrizione* e *subscriber*. Nello specifico esso:
  - registra una **sottoscrizione *durable*** sul provider;
  - la sottoscrizione è specificata da un **nome univoco**;
    - coppia **client ID (connessione)** + **nome della sottoscrizione**.
- Se una sottoscrizione non ha *subscriber* attivi, **i messaggi sono conservati dal provider JMS**

Durable Subscription

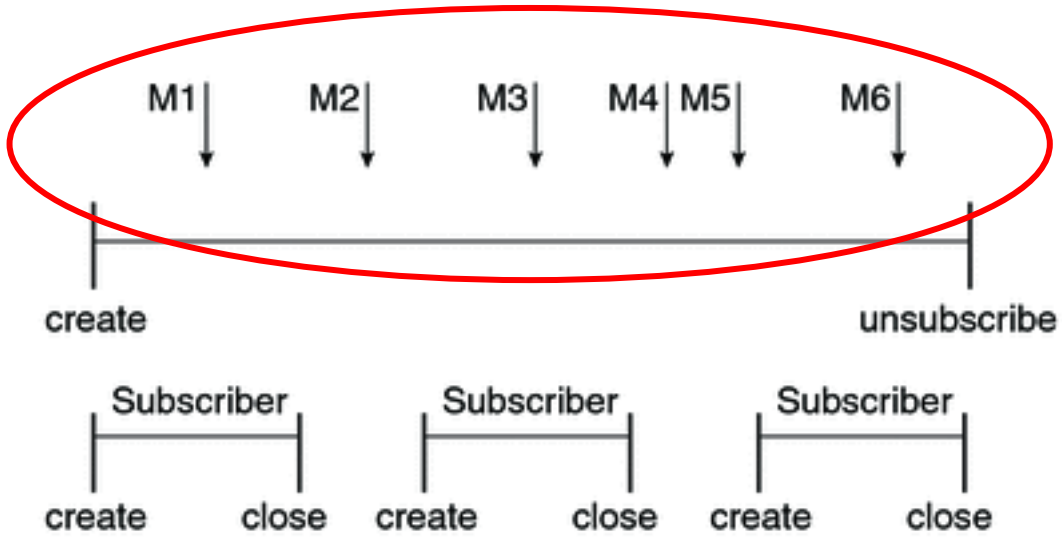
```
[… codice omesso …]

TopicConnection topicConn = connFactory.createTopicConnection();

topicConn.setClientID(”subdurableID");

topicConn.start();
TopicSession topicSession = topicConn.createTopicSession(false,
Session.AUTO_ACKNOWLEDGE);

TopicSubscriber sub =
topicSession.createDurableSubscriber(topic,"MakeItLastConn");

[… codice omesso …]
```

La coppia (ID, nome_subscription) identifica univocamente la sottoscrizione

Durable Subscription



Nel caso di una **durable subscription,** il Subscriber può attivarsi e disattivarsi senza perdere messaggi.



Estensioni STOMP per la semantica dei messaggi JMS

- Nel caso di una comunicazione tra entità JMS e STOMP bisogna tenere conto dei **diversi campi dell'header** utilizzati in JMS e STOMP
- STOMP supporta le seguenti proprietà JMS per i messaggi di tipo **SENT**

STOMP Header	JMS Header	Description
`correlation-id`	`JMSCorrelationID`	Good consumers will add this header to any responses they send.
`expires`	`JMSExpiration`	Expiration time of the message.
`JMSXGroupID`	`JMSXGroupID`	Specifies the Message Groups.
`JMSXGroupSeq`	`JMSXGroupSeq`	Optional header that specifies the sequence number in the Message Groups.
`persistent`	`JMSDeliveryMode`	Whether or not the message is persistent.
`priority`	`JMSPriority`	Priority on the message.
`reply-to`	`JMSReplyTo`	Destination you should send replies to.
`type`	`JMSType`	Type of the message.

Estensioni di ActiveMQ per STOMP

- È possibile ***aggiungere header custom*** ai comandi STOMP per configurare il protocollo ActiveMQ

- Per esempio, è posisbile definire durable subscriber in STOMP (visti nella lezione STOMP)
  - Gli header custom **`client-id`** (comando CONNECT) e **`activemq.subscriptionName`** (comando SUBSCRIBE) devono essere usati in combinazione (in maniera similare ai durable subscriber JSM)
    - Il campo **`client-id`** viene impostato con l'hostname se non esplicitamente impostato
    - La coppia (**`client-id`** , **`activemq.subscriptionName`**) identifica univocamente la sottoscrizione

```
conn.connect(wait=True, headers = {"client-id":"IDtestsub_durable"})
. . .
conn.subscribe(destination='/topic/mytesttopic', id=1    ack='auto',
    persistent=True,
    headers = {"activemq.subscriptionName":"IDtestsubscription"})
```

Estensioni di ActiveMQ per STOMP

Verb	Header	Type	Description
`CONNECT`	`client-id`	`string`	Specifies the JMS clientID which is used in combination with the `activemq.subcriptionName` to denote a durable subscriber.
`SUBSCRIBE`	`activemq.dispatchAsync`	`boolean`	Should messages be dispatched synchronously or asynchronously from the producer thread for non-durable topics in the broker? For fast consumers set this to `false`. For slow consumers set it to `true` so that dispatching will not block fast consumers.
`SUBSCRIBE`	`activemq.exclusive`	`boolean`	I would like to be an Exclusive Consumer on the queue.
`SUBSCRIBE`	`activemq.maximumPendingMessageLimit`	`int`	For Slow Consumer Handling on non-durable topics by dropping old messages - we can set a maximum-pending limit, such that once a slow consumer backs up to this high water mark we begin to discard old messages.
`SUBSCRIBE`	`activemq.noLocal`	`boolean`	Specifies whether or not locally sent messages should be ignored for subscriptions. Set to `true` to filter out locally sent messages.
`SUBSCRIBE`	`activemq.prefetchSize`	`int`	Specifies the maximum number of pending messages that will be dispatched to the client. Once this maximum is reached no more messages are dispatched until the client acknowledges a message. Set to a low value > **1** for fair distribution of messages across consumers when processing messages can be slow. **Note**: if your STOMP client is implemented using a dynamic scripting language like Ruby, say, then this parameter ***must*** be set to `1` as there is no notion of a client-side message size to be sized. STOMP does not support a value of `0`.

Estensioni di ActiveMQ per STOMP

Verb	Header	Type	Description
SUBSCRIBE	activemq.priority	byte	Sets the priority of the consumer so that dispatching can be weighted in priority order.
SUBSCRIBE	activemq.retroactive	boolean	For non-durable topics make this subscription retroactive.
SUBSCRIBE	activemq.subscriptionName	string	For durable topic subscriptions you must specify the same activemq.client-id on the connection and activemq.subcriptionName on the subscribe prior to v5.7.0. **Note**: the spelling subcriptionName NOT subscriptionName. This is not intuitive, but it is how it is implemented in ActiveMQ Classic 4.x. For the 5.0 release of ActiveMQ Classic, both subcriptionName and subscriptionName will be supported (subcriptionName was removed as of v5.6.0).
SUBSCRIBE	selector	string	Specifies a JMS Selector using SQL 92 syntax as specified in the JMS 1.1 specification. This allows a filter to be applied to each message as part of the subscription.

Lavorare con i messaggi JMS Text/Bytes e STOMP

- STOMP è un protocollo molto semplice e **non riconosce i messaggi JMS**, come TextMessage o BytesMessage
- Il protocollo supporta tuttavia un **header** denominato **content-length**
- ActiveMQ permette utilizza questo header per **determinare quale tipo di messaggio creare quando si invia da STOMP a JMS**
  - **Inclusione del content-length header**: il messaggio risultante è di tipo `ByteMessage`
  - **Non inclusione del content-length header**: il messaggio risultante è di tipo `TextMessage`
- La stessa logica può essere seguita anche nel passaggio **da JMS a STOMP**

Lavorare con i messaggi JMS Text/Bytes e STOMP

```
# sender.py
import stomp

conn = stomp.Connection([('127.0.0.1', 61613)], auto_content_length=False)
conn.connect(wait=True)

conn.send('/topic/mytesttopic', 'test message')

conn.disconnect()
```

E' necessario utilizzare il `physical-name` del topic/queue specificato lato JMS

```
//      jndi topic name    physical topic-name
jndiProperties.put( key:"topic.test", value:"mytesttopic" );
```

L'inclusione/esclusione dell'header `auto_content_length` specifica il tipo di messaggio da creare quando si invia un messaggio da STOMP a JMS

Inclusion of content-length header	Resulting Message
yes	`BytesMessage`
no	`TextMessage`