Dipartimento di Informatica
e Sistemistica Antonio Ruberti

Sapienza
Università di Roma

Architetture Software Orientate ai Servizi

Java Message Service - JMS

"Magnifico ambasciatore. «Tarde non furon mai grazie divine». Dico questo perché mi pareva di aver non perduto, ma smarrito la grazia Vostra, essendo stato Voi molto tempo senza scrivermi; ed ero nel dubbio chiedendomi donde potesse nascerne la causa.„

[N. Machiavelli, *Lettera a Francesco Vettori in Roma*]

Java Message Service

- Java message service **(JMS)** è l'insieme di API che consentono lo scambio di messaggi tra applicazioni Java distribuite sulla rete
- La prima specifica JMS è stata rilasciata nel 1998, mentre l'ultima versione delle API è la 1.1, pubblicata nel 2002

Messaggio

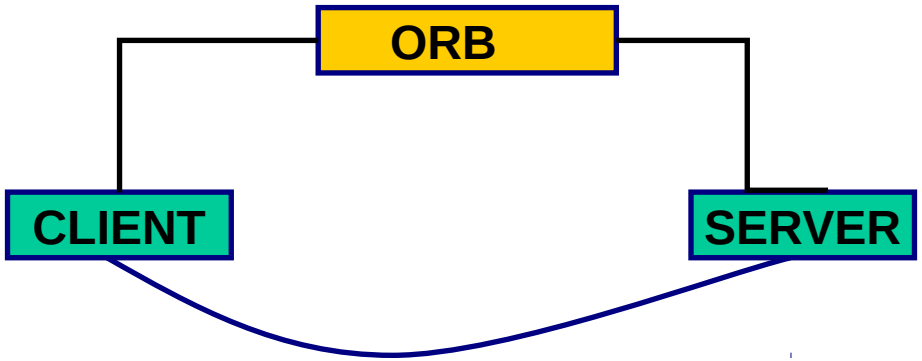
- In ambito JMS, un messaggio è un raggruppamento di dati che viene inviato da un sistema ad un altro
  - I dati sono solitamente utilizzati per notificare eventi ed informazioni
  - Sono pensati per essere utilizzati da software

Messaging System

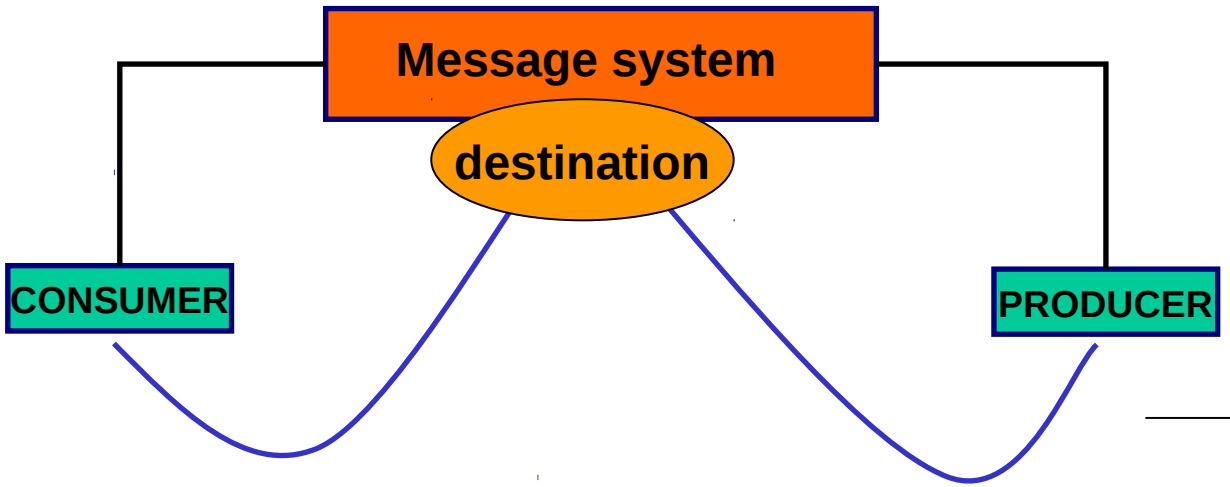
- Con il termine ***messaging*** ci si riferisce ad un meccanismo che consente la comunicazione **asincrona** tra client remoti:
  - Un ***producer*** invia un messaggio ad uno o più *consumer*
  - Il ***consumer*** riceve il messaggio ed esegue le operazioni correlate, **in un secondo momento**
- *Producer* e *consumer* sono definiti entrambi ***JMS client***.
  - Occorre non confondersi con il paradigma client/server
- I *JMS client* sfruttano l'infrastruttura fornita dal ***JMS Provider***

Messaging System (2)

DIPARTIMENTO DI INFORMATICA
E SISTEMISTICA ANTONIO RUBERTI
SAPIENZA
UNIVERSITÀ DI ROMA



Modello ORB (RMI)



MODELLO MESSAGE ORIENTED

JMS non include…

- Un sistema di Load balancing/Fault Tolerance: la API JMS non specifica un modo in cui diversi client possano cooperare al fine di implementare un unico servizio critico
- Notifica degli Errori
- JMS non prevede API per il controllo della privacy e dell'integrità del messaggio

Architettura JMS

- **JMS Client**: programma che manda o riceve i messaggi JMS
- **Messaggio**: ogni applicazione definisce un insieme di messaggi da utilizzare nello scambio di informazioni tra due o più client
- **JMS provider**: sistema di messaggistica che implementa JMS e realizza delle funzionalità aggiuntive per l'amministrazione e il controllo della comunicazione attraverso messaggi
- **Administered objects**: sono oggetti JMS preconfiguarti, creati da un amministratore ad uso dei client. Incapsulano la logica specifica del JMS provider nascondendola ai client, garantendo maggiore portabilità al sistema complessivo.

Administered Objects

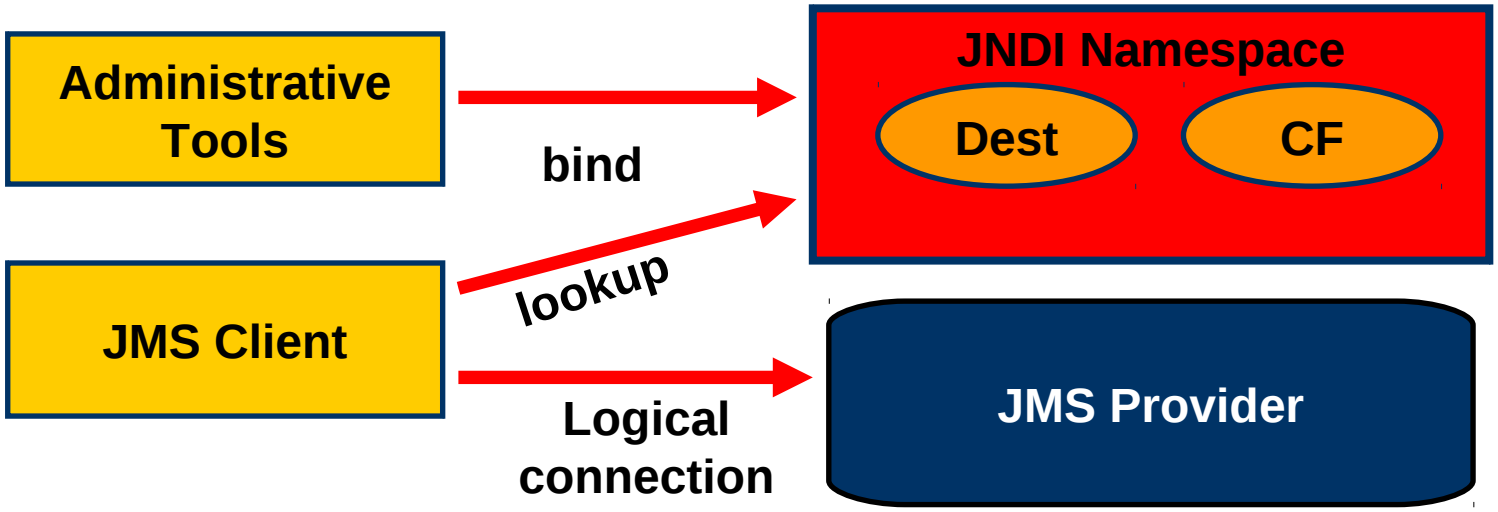
- **Connection Factory:** oggetto utilizzato da un client per realizzare la connessione con il provider
- **Destination (queue/topic):** oggetto utilizzato da un client per specificare la destinazione dove il messaggio è spedito/ricevuto.

Administered Objects (2)

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

- Le Connection Factory e le Destination sono legati a *nomi simbolici*
- I servizi **JNDI** (***Java Naming and Directory Service***) sono tipicamente resi disponibili dall'Application Server per ottenere gli oggetti remoti registrati.
  - I nomi degli oggetti sono completamente indipendenti dal luogo fisico in cui gli stessi sono resi disponibili.
  - Il "Naming" degli oggetti in JMS è gestito dall'amministratore di sistema e *non* dai client (come in RMI)
- Una applicazione che usa JMS adopera JNDI per ottenere i riferimenti a tali oggetti.
  - Per esempio, un sender o un receiver.

Bind e lookup



- Attraverso gli *Administrative Tools*, gli amministratori responsabili del dispiegamento dei componenti effettuano il deploy ed associano (***bind***) un nome JNDI agli oggetti *Destination* e *Connection Factory*
- I client JMS ricercano gli Administered Objects (***lookup***) ed instaurano una connessione logica ad un JMS Provider

Modelli di messaggistica (message domain)

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

- **Point-to-point (PTP):**
  - Il *producer* si definisce ***sender***
  - Il *consumer* si definisce ***receiver***
  - Più JMS Client possono condividere la stessa *Destination* (**queue**)
  - Un messaggio può essere consumato da **un solo receiver**
  - Consente una comunicazione **uno-a-uno**
- **Publish/Subscribe (Pub/Sub):**
  - Il *producer* si definisce **publisher**
  - I *consumer* si definiscono **subscriber**
  - Più JMS Client possono condividere la stessa *Destination* (**topic**)
  - Il messaggio pubblicato viene ricevuto da **tutti i subscriber** che si siano dichiarati **interessati**
  - Consente una comunicazione **uno-a-molti**

Point-to-point domain

Dipartimento di Informatica
e Sistemistica Antonio Ruberti
Sapienza
Università di Roma

- Ogni JMS Client spedisce e riceve messaggi mediante canali virtuali conosciuti come **queue** (coda)
- È un modello di tipo "*pull-based*"
  - spetta ai receiver prelevare i messaggi dalle code

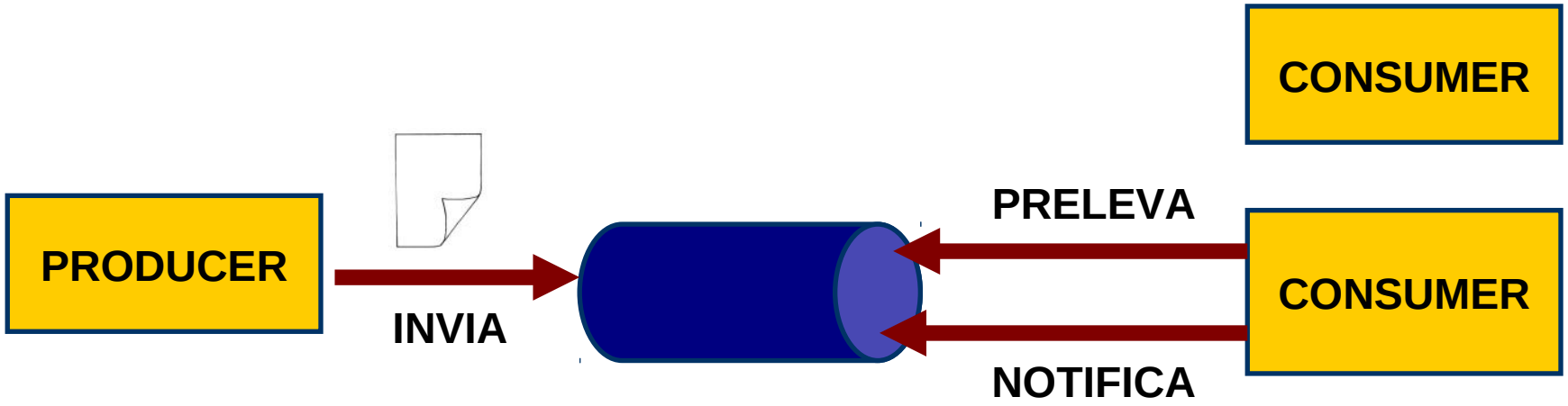
Point-to-point domain (2)

Dipartimento di Informatica
e Sistemistica Antonio Ruberti
Sapienza
Università di Roma

- Più producer e più consumer possono condividere la stessa coda, ma…
- Ogni messaggio può essere letto da un solo consumer
- Sender e receiver non hanno alcuna dipendenza temporale rispetto ai messaggi
- Il receiver notifica l'avvenuta ricezione e processamento del messaggio (**acknowledgement**)
- Il PTP si rivela utile quando è necessario garantire che un messaggio arrivi ad un solo destinatario che notifichi la corretta ricezione

Point-to-point domain (3)

DIPARTIMENTO DI INFORMATICA
E SISTEMISTICA ANTONIO RUBERTI
SAPIENZA
UNIVERSITÀ DI ROMA



Publish/Subscribe domain

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

- Ogni producer spedisce messaggi a molti consumer mediante canali virtuali conosciuti come **topic** (argomento)
- È un modello di tipo "*push-based*"
  - i messaggi vengono automaticamente inviati in **broadcast** ai consumer
- Per ricevere i messaggi, i consumer devono *sottoscriversi* ad un topic
- Qualsiasi messaggio spedito al topic viene consegnato **a tutti** i consumer sottoscritti, ciascuno dei quali riceverà una copia identica di ciascun messaggio inviato al topic

Publish/Subscribe domain (2)

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

- Il publisher ed i subscriber hanno una dipendenza temporale:
  - Un consumer che si sottoscrive ad un topic può consumare solamente messaggi pubblicati *dopo* la sua sottoscrizione.
  - Il subscriber può continuare a consumare messaggi solo nel periodo in cui rimane attivo.
    - Durante il periodo di inattività i messaggi che dovrebbe ricevere andrebbero persi.
- Per ovviare al problema il client JMS può dichiararsi **Durable Subscriber**
  - Questo gli consente di disconnettersi e riconnettersi in un secondo momento, ricevendo tutti i messaggi pubblicati durante il periodo di disconnessione

Publish/Subscribe domain (3)

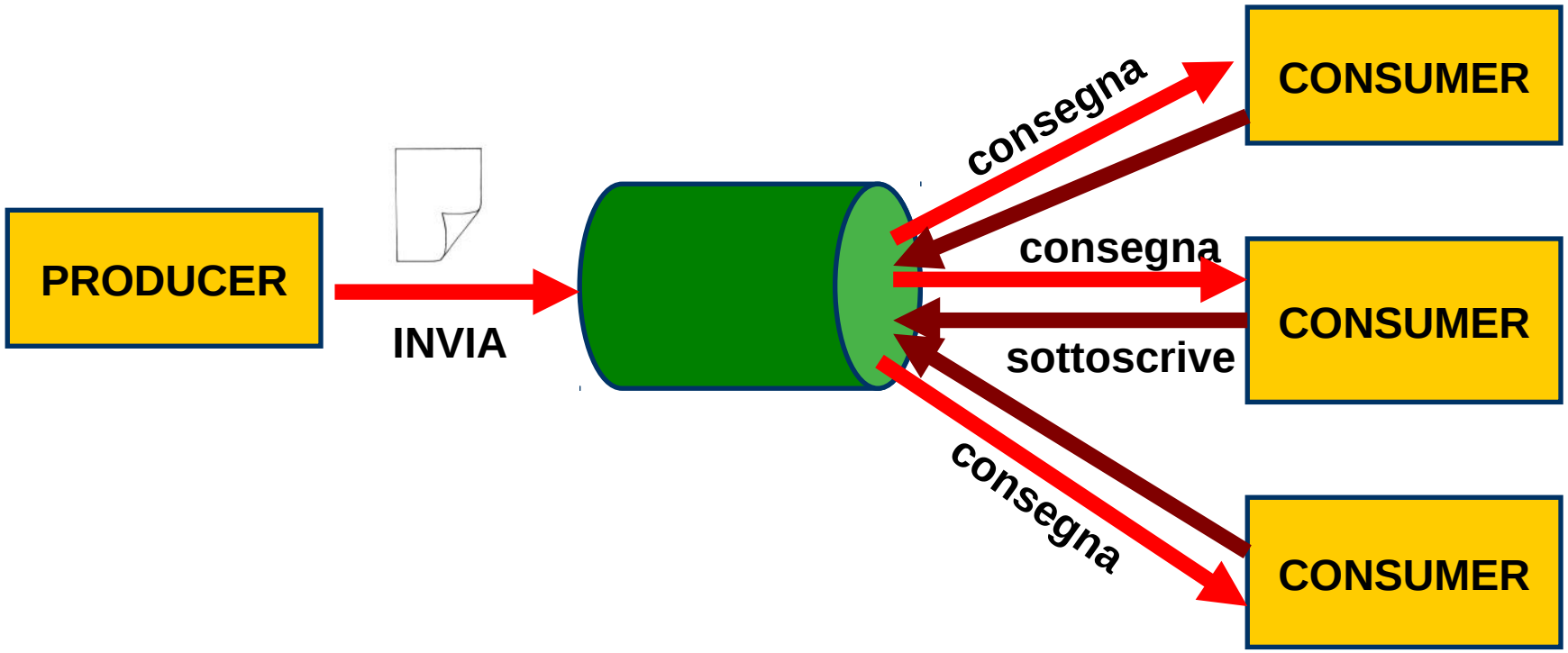
DIPARTIMENTO DI INFORMATICA
E SISTEMISTICA ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

- Il modello **pub/sub** prevede due tipi di **sottoscrizione**:
  - **Topic-based**:
    - le informazioni vengono suddivise in argomenti (*topic* o *subject*)
    - le sottoscrizioni e le pubblicazioni vengono effettuate scegliendo come discriminante un argomento (letteralmente, *topic*)
  - **Content-based**:
    - utilizza dei **filtri** (*message selector*) per una selezione più accurata delle informazioni da ricevere sul topic

Publish/Subscribe domain (4)

DIPARTIMENTO DI INFORMATICA
E SISTEMISTICA ANTONIO RUBERTI
SAPIENZA
UNIVERSITÀ DI ROMA



Modalità di ricezione

- Un messaggio JMS può essere consumato secondo due modalità:
  - **Modalità sincrona:** il subscriber (o il receiver) prelevano direttamente il messaggio dalla coda, tramite l'invocazione del metodo **`receive().`**
    - Il client rimane bloccato finché non arriva il messaggio o fino allo scadere di un timeout
  - **Modalità asincrona:** il client registra un *message listener* su una destination.
    - Ogni qual volta un messaggio è pronto per essere consegnato ad un client, il JMS Provider lo consegna ed il listener associato viene invocato.

Connection Factory

- È l'interfaccia utilizzata dal JMS Client per creare connessioni con il JMS Provider onde accedere al servizio JMS.
  - `javax.jms.ConnectionFactory`
- `ConnectionFactory` è estesa delle interfacce:
  - **`QueueConnectionFactory`**
  - **`TopicConnectionFactory`**

Connection Factory (2)

DIPARTIMENTO DI INFORMATICA
E SISTEMISTICA ANTONIO RUBERTI
SAPIENZA
UNIVERSITÀ DI ROMA

- Nel caso di PTP Domain si utilizza l'interfaccia `javax.jms.QueueConnectionFactory`

```
Context ctx = new InitialContext();
QueueConnectionFactory queueConnectionFactory =
(QueueConnectionFactory)ctx.lookup("ConnectionFactory");
```

- Context è l'interfaccia base per la specifica di un naming context di JNDI.
- Si crea un oggetto di tipo context
- Si recupera il riferimento logico all'oggetto con il metodo lookup

Connection Factory (3)

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

- Nel caso di dominio publish/subscribe si utilizza l'interfaccia TopicConnectionFactory

```
Context ctx = new InitialContext();
TopicConnectionFactory topicConnectionFactory =
(TopicConnectionFactory)ctx.lookup("ConnectionFactory");
```

Connection

- Dopo aver effettuato il *lookup* per recuperare lo handle ad un'implementazione di Connection Factory, è possibile creare un oggetto di tipo **`javax.jms.Connection`**
- PTP Domain:
  - Si ottiene lo handle all'interfaccia **`QueueConnection`** invocando il metodo **`createQueueConnection()`** sull'oggetto **`QueueConnectionFactory`**:
- Pub/Sub Domain:
  - Si ottiene lo handle all'interfaccia **`TopicConnection`** invocando il metodo **`createTopicConnection()`** sull'oggetto **`TopicConnectionFactory.`**

```
QueueConnection queueConnection =
queueConnectionFactory.createQueueConnection();
```

```
TopicConnection topicConnection =
topicConnectionFactory.createTopicConnection();
```

Session

- Un oggetto che implementi l'interfaccia `javax.jms.Session` può essere creato a partire da istanze `Connection`
- Permette l'istanziazione di
  - producer,
  - consumer,
  - messaggi.
- L'interfaccia è estesa, rispettivamente per la gestione di topic e queue, da
  - `javax.jms.TopicSession`
  - `javax.jms.QueueSession`

Destination (1)

- È l'interfaccia verso gli *Administered Object* che astraggono *queue* e *topic*
  - **`javax.jms.Destination`**
- Nel PTP Domain, la destination è specializzata dall'interfaccia **`javax.jms.Queue`**
- Nel Pub/Sub Domain, la Destination è specializzata dall'interfaccia **`javax.jms.Topic`**
- Anche in questo caso il client identifica la destinazione mediante l'utilizzo delle API JNDI

Destination (2)

- In JBoss, la definizione, configurazione ed installazione di destination è gestita dall'amministratore dell'Application Server
  - In JBoss 5.1, il sistema di messaggistica, bundled nell'Application Server, era *JBoss Messaging*
    - Si potevano adoperare, tramite interfaccia JMX, file `<nomecoda>-service.xml` dispiegati nella directory `$JBOSS_HOME/server/default/deploy`
  - Da JBoss 6, il sistema di messaggistica, bundled nell'Application Server, è *HornetQ* (http://www.jboss.org/hornetq)
    - Nuove destination possono essere specificate alterando il file `$JBOSS_HOME/server/default/deploy/hornetq/hornetq-jms.xml`

Esempio per JBoss 5.1 (quotazioni-service.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE server>
<server>
  <mbean code="org.jboss.mq.server.jmx.Topic"
  name="jboss.mq.destination:service=Topic,name=quotazioni">
    <depends
    optional-attribute-name="DestinationManager">
      jboss.mq:service=DestinationManager
    </depends>
  </mbean>
</server>
```

Esempio per JBoss 6 (hornetq-jms.xml)

```
<configuration
xmlns="urn:hornetq"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:hornetq /schema/hornetq-jms.xsd">

<!-- ... -->
<!-- in fondo al file -->

 <queue name="MoMIMAPCrawler">
  <entry name="/queue/MoMIMAPCrawler"/>
 </queue>

</configuration>
```

Lookup delle Destination

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

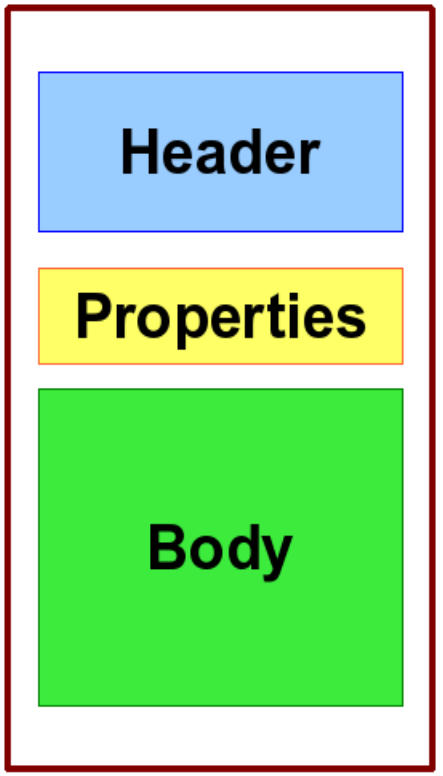
- Un consumer deve ottenere il riferimento alla destination invocando il metodo lookup
- L'input è il nome della destination precedentemente dispiegata
  - Nell'assegnazione dello handle, occorre non dimenticare il cast al tipo opportuno (`javax.jms.Queue` o `javax.jms.Topic`)

```
Queue queue = (Queue)
ctx.lookup("/queue/myQueue");

Topic topic = (Topic)
ctx.lookup("/topic/myTopic");
```

Message

- Cositutito da:
- **Header (obbligatorio):** contiene informazioni sul messaggio
  - e.g., il **`JMSCorrelationID`**, adoperato per connettere due messaggi in relazione l'uno con l'altro
- **Property (opzionale):** contiene alcune proprietà opzionali del messaggio solitamente utilizzate per gestire la compatibilità tra sistemi di messaggistica differenti.
  - I suoi campi sono esaminati dai un consumer mediante i **message selector**
- **Body (opzionale):** contiene la parte informativa trasferita all'interno del messaggio



Message Body

- Esistono vari tipi di messaggio ma qui ci limiteremo ai messaggi di testo
  - Un oggetto **`TextMessage`** viene creato per mezzo dei metodi **`createTextMessage()`** forniti dall'interfaccia **`Session`**
- Nel caso **PTP**, per creare un messaggio di tipo **`TextMessage`** utilizziamo il seguente codice:

```
TextMessage txtMsg = queueSession.createTextMessage();
txtMsg.setText("Ciao!");
producer.send(txtMsg);
[…]
```

Si invoca il metodo `createTextMessage()` su un oggetto `queueSession` specifico per il dominio PTP

Message Property

- I campi property possono essere impostati e letti mediante i metodi:
  - `set<Type>Property(String name, <Type> value)`
  - `<Type> get<Type>Property(String name)`

    dove `<Type>` è il tipo di property
    - `Boolean`, `Byte`, `Integer`, `String`, `Object`...
- Il seguente esempio imposta il valore di due property

```
TextMessage txtMsg = queueSession.createTextMessage();
txtMsg.setText("Ciao!");
txtMsg.setStringProperty("Sender", "Massimiliano");
txtMsg.setIntegerProperty("Priority", 2);
```

Message Producer

- Il **`message producer`** è l'oggetto che ha il compito di inviare messaggi ad una destination
- Implementa l'interfaccia
  `javax.jms.MessageProducer`
  ed è generato da un oggetto
  `javax.jms.Session`
  attraverso il metodo

```
MessageProducer createProducer(
Destination destination
)
```

- Come d'uopo, si hanno le specializzazioni per queue e topic
  - **`TopicPublisher createPublisher(`**
    `Topic` `topic`
)
    in
    `javax.jms.`**`TopicSession`**
  - **`QueueSender createSender(`**
    `Queue` `queue`
)
    in
    `javax.jms.`**`QueueSession`**

Message Consumer

- Il **message consumer** è l'oggetto che ha la possibilità di trarre messaggi da una destination
- Implementa l'interfaccia
  javax.jms.MessageConsumer
  ed è generato da un oggetto
  javax.jms.Session
  attraverso il metodo

```
MessageConsumer createConsumer(
Destination destination[, String messageSelector]
)
```

- Come d'uopo, si hanno le specializzazioni per queue e topic
  - **TopicSubscriber createSubscriber(**
    Topic topic[, String messageSelector, boolean noLocal]
)
    in
    javax.jms.**TopicSession**
  - **QueueReceiver createReceiver(**
    Queue queue[, String messageSelector]
)
    in
    javax.jms.**QueueSession**

Modalità Asincrona

- È la modalità standard di trasmissione di messaggi in JMS
- Il producer non è tenuto ad attendere che il messaggio sia ricevuto per continuare il suo funzionamento
- Per supportare il consumo asincrono dei messaggi, deve essere utilizzato un oggetto *listener* che implementi l'interfaccia

  `javax.jms.MessageListener`
- Tramite overriding del metodo

  `void onMessage(Message message)`

  si possono definire le operazioni da effettuare all'arrivo di un messaggio.

Message Listener

- Passi:
  - Generare l'oggetto listener

```
TopicListener topicListener = new TextListener();
QueueListener queueListener = new TextListener();
```

  - Passare l'oggetto listener creato come input al metodo
    `void setMessageListener(`
    `MessageListener listener`
    `)`
    degli oggetti **`MessageConsumer`**

Message Listener

- Il listener, dopo essere stato connesso **si mette in ascolto** di un messaggio, avviando la connessione

```
queueConnection.start();
topicConnection.start();
```

- All'arrivo di un messaggio viene invocato il metodo **`onMessage()`**

```
public class TextListener implements MessageListener
{
        public void onMessage(Message message)
        {
                // Codice per consumare il messaggio
        }
}
```

Modalità Sincrona

- I prodotti di messaging sono intrinsecamente asincroni ma un message consumer può ricevere i messaggi anche in modo sincrono
- Il consumer **richiede esplicitamente** alla destinazione di prelevare il messaggio (*fetch*) invocando il metodo

  `Message receive([long timeout])`

  offerto dall'interfaccia

  `javax.jms.MessageConsumer`

  – È sospensivo, ovverosia rimane bloccato fino alla ricezione del messaggio, a meno che non si espliciti un **timeout**, scaduto il quale il metodo termina

Modalità Sincrona (2)

```
while(<condition>) {
  Message m = queueReceiver.receive();
  if( (m != null) && (m instanceof TextMessage) ) {
    message = (TextMessage) m;
    System.out.println("Rx: " + message.getText());
  }
}
```

Indipendentemente dalla modalità di ricezione, la comunicazione viene conclusa invocando un'operazione di `close()` sulla connessione

```
queueConnection.close();
topicConnection.close();
```

Message Selector

- I message selector sono parametri utilizzati per filtrare messaggi in arrivo, permettendo ad un consumer di specificare quali siano quelli di suo interesse, sulla base delle informazioni contenute all'interno del messaggio
  - La selezione avviene ***solo*** a livello di ***header*** **e** ***property***
    non di ***body***
- Assegnano il lavoro di filtraggio al JMS Provider anziché all'applicazione
  - Il filtraggio avviene a livello di server e permette di inoltrare ai client lungo la rete i messaggi strettamente necessari o utili, risparmiando così la banda del canale
- Un message selector è un oggetto di tipo String che contiene un'espressione
  - la sintassi appartiene ad un sottoinsieme dello standard SQL92
- Il metodo **`createConsumer()`** ammette opzionalmente di specificare un message selector

```
MessageConsumer createConsumer(
Destination destination[, String messageSelector]
)
```

Esempio

```
ctx = new InitialContext(properties);
cf = (TopicConnectionFactory)ctx.lookup("ConnectionFactory");
destination = (Topic)ctx.lookup("topic/allarme");
connection = cf.createTopicConnection();
session = connection.createTopicSession(
        false,  Session.AUTO_ACKNOWLEDGE
);

selector = "sender = 'Massimiliano' AND priority > 0";
subscriber =
session.createSubscriber(destination,selector,false);

subscriber.setMessageListener (new TextListener());
connection.start();
```