
# MOM
## Message Oriented Middleware

&

# JMS
## Java Message Service

Ada Diaconescu
ada.diaconescu@telecom-paris.fr

# Overview

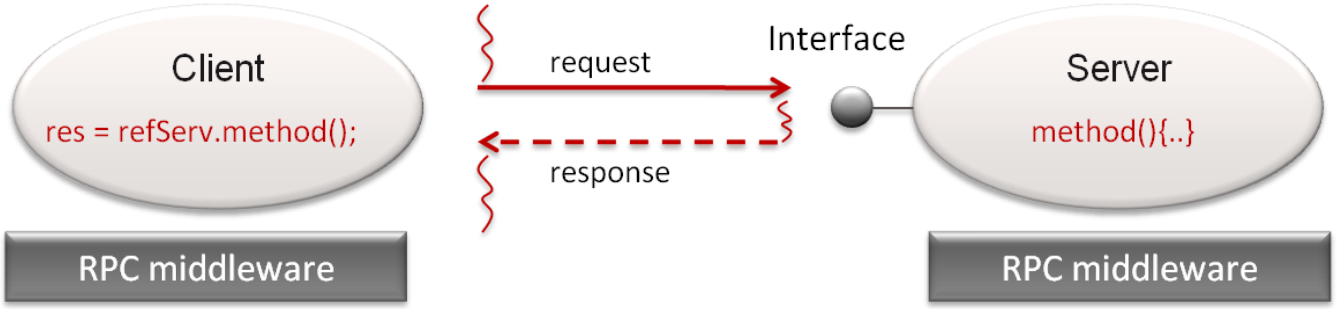
- Message Oriented Middleware – MOM
- Java Message Service – JMS
- Code Examples (using the JMS API)


# Part 1

MOM – Message Oriented Middleware

# Motivation – Why *messages*? (1 /2)

## RPC / RMI – Reminder



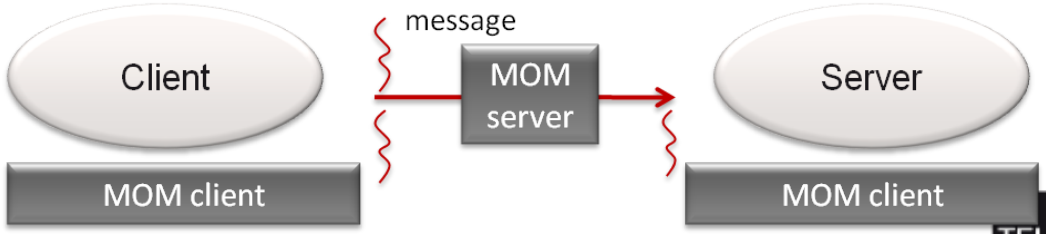

- Strong-coupling
  - The Client depends on the Server's Interface
  - The client must "know" the Server (Reference)
- Time dependency
  - The Client and the Server must be available simultaneously
- Synchronous/blocking communication (usually)
  - The Client is blocked after sending the request and until receiving the response

# Motivation – Why *Messages*? (2 /2)

## Different Applications have different constraints

- No simultaneous availability
  - Different application components are not always available at the same time (especially for large-scale, distributed applications)
- Need for asynchronous / non-blocking communication
  - Business logic allows components to send information to other components and then continue executing in the absence of an immediate response
- Need for loose-coupling
  - Developers wish to avoid interface dependencies between components and even having to have direct references between components => facilitate component replacement

⇒ Solution:
Message-based Communication



# Message Oriented Messages (MOM)

- Offer a simple & reliable model for message-exchanges in distributed systems
- Employ one of the most ancient communication models
- Used for large systems
  - Banking networks
  - Telecommunications
  - Online booking and commerce
  - … etc.

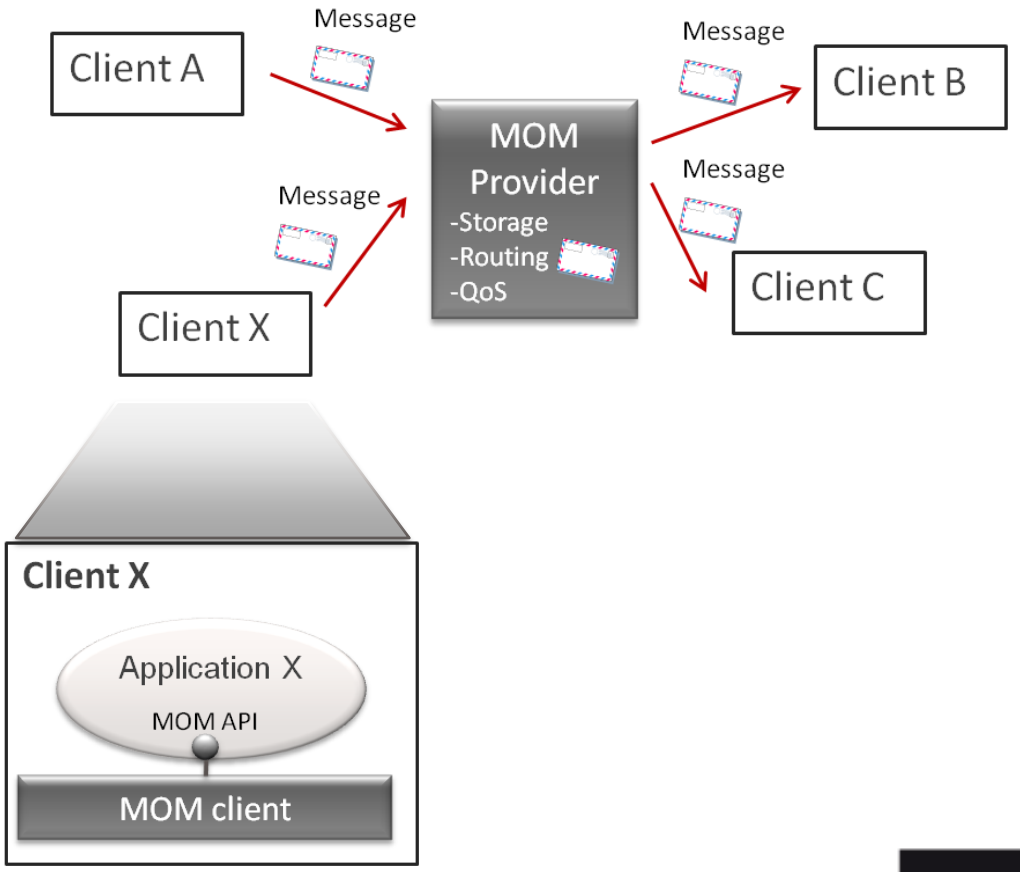
# MOM – Typical Characteristics

- Message delivery guarantees
- Transactions support
- Routing management
- Large-scale support
- Configuration support (QoS policies)
- Loosely-coupled components (generally!)
  - No interface dependency
  - No direct references to components
  - No time dependency – the sender is not interrupted if the receiver is unavailable
  - Asynchronous / non-blocking communication (no implicit/compulsory reply; except ack.)

# MOM – General Architecture

## Main Entities:

- Client – MOM user
  - Sends and receives messages
- Provider – MOM support
  - Routing, message storing, configuration, ...
- Message – transmitted data
  - MIME type - text, sound, image, ..
  - QoS - priority, delivery date, ..



# MOM – Configuration Parameters

## MOM Administration

- Deployment
  - Node positioning
  - Resource allocation
- Message Queue Configurations
  - Queue size
  - Persistence
  - Message filtering
- Administration tools

## Client Configuration

- MOM access point
  - Identification
  - Connexions set-up
- Message transmission/reception mode
  - Connection type
  - MOM access priorities
  - Reception filtering


# Part 2

JMS – Java Message Service

API

# MOM – need for standardisation

- A unique definition of the distribution model
  - “distribution model based on message exchanges amongst the nodes of a distributed application”
- Multiple implementation solutions
  - Different semantics and offered services
    - Blind message transmission or acknowledgement support; transactions support, ...
    - Managing node mobility, dynamic reconfiguration, ...
  - Different architectures and implementations
    - Communication based on TCP/IP, IP multicast, SSL, HTTP, a lower-layer of RPC, …
    - Various implementations of message queues and topics
    - Various supported topologies – centralised, decentralised, hybrid
  - => Need for standardisation
    - E.g.: Java Message Service (JMS), OMG COS Events/Notification, WS-Reliable Messaging, AMQP, MQTT…

# MOM – Standardisation Efforts

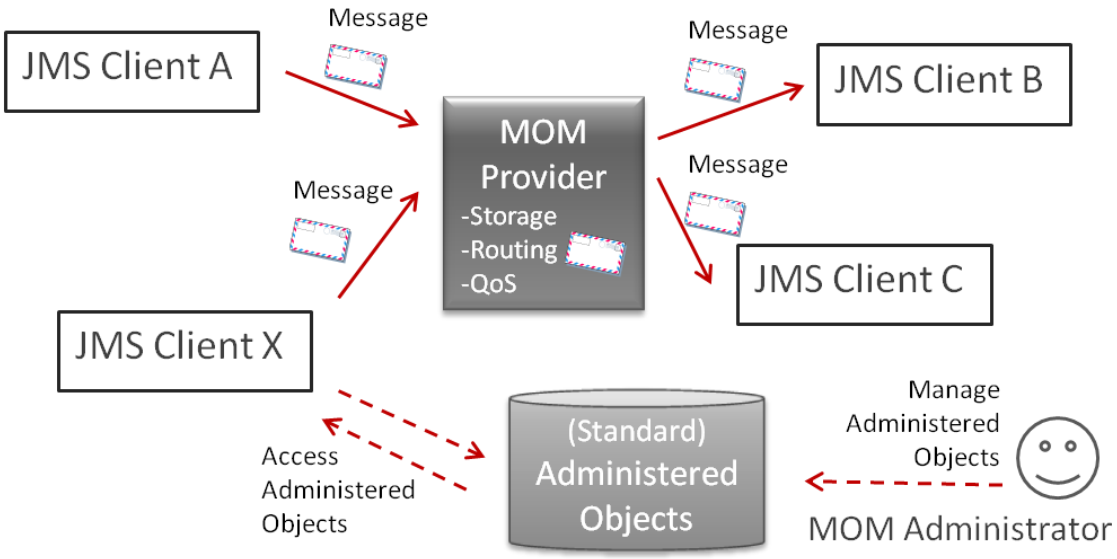
- Until ~2001, little or no normalisation efforts
  - One API per MOM vendor
  - Different designs (e.g. Resource usage)
  - Different functionalities
- Difficulties
  - Limited interoperability (critical)
  - Maintenance and evolution problems
- Evolutions
  - Java Messaging Service (JMS) – a standard API for Clients
  - CORBA COS Notification - a Client API, infrastructure description (objects & API)
  - Advanced Message Queueing Protocol (AMQP) – an open standard for interoperability of MOMs based on different languages and platforms
  - MQTT – lightweight open standard protocol for Internet of Things (IoT) and Machine To Machine (M2M) systems
  - …

# (Sun/Oracle's) JMS: Java Message Service

- MOM API Specification
- Integrated within J2EE 1.3 ++, coupled to EJB (Message-Driven Bean)
- First publicly-accessible MOM specification
  - Implemented by the main MOMs
  - Adaptable to other languages (C++, Ada)
  - Few restrictions: a synthesis of existing MOMs => authorises rather than constraining
- JMS : 1.1 specification
  - Specification for Clients
    - P-t-P, Pub/Sub, call-backs,
    - Filtering (SQL-like syntax) & transactions
    - Message types
  - Does not specify the infrastructure
    - Protocol, representation, transport
    - Configuration process
    - Error handling, failure management
    - Administration interfaces
    - Security

# JMS Architecture

- **JMS Client** – uses the JMS API
  - To send / receive messages
  - For "Administered Objects"
- **JMS Provider** – impl. The JMS API
  - Queues and Topics management
  - Message transmission
  - Global MOM configuration
  - Source of heterogeneity – e.g. Administration mechanisms, used protocol, error handling, …
- **JMS Message** – objects that contain communicated information
  - Header
  - Properties (optional)
  - Body (optional)




- **Administered Objects**
  - Connection Factories
  - Destinations: Queues & Topics

# JMS Message

- **Three parts: header, properties, body**
- **Header** : message dientification and routing
  - Pairs (name, value):

    `JMSDestination, JMSDeliveryMode, JMSMessageID, JMSTimestamp, JMSExpiration, JMSRedelivered, JMSPriority, JMSReplyTo, JMSCorrelationID, JMSType`

- **Properties** (optional) – application-specific, used ofr message-filtering

  E.g. – sender : `message.setStringProperty("Username","John Doe");`

  E.g. – receiver :
  ```
  TopicSubscriber sub =
  session.createSubscriber(chatTopic,"Username !='William'");//avec filtre!
  ```

- Body (optional) – contains the communicated application data

  E.g.: `message.setText(MSG_TEXT + " " + (i + 1));`

  - TextMessage: chain of characters
  - MapMessage: pair set (name, value)
  - ByteMessage: byte stream
  - StreamMessage: value stream
  - ObjectMessage: serializable object

# JMS Message Reception

- *Careful using the terms:“synchronous” & ”asynchronous” !*

- **Synchronous** reception – mode “pull”, blocking
  - The consumer gets the message from the Destination by calling the `receive` method
  - This method blocks until a message becomes available at that Destination, or until the request expires.

- **Asynchronous reception** – mode “push ”, non-blocking, with time dependency
  - The consumer registers a `MessageListener` with the targeted Destination
  - When a message arrives at that Destination the MOM provider delivers it to the registered Message Listener/s, by calling its `onMessage` method

- *NOTE: whether a message is delivered to a single consumer or to several consumers depends on the Destination type (see next)*

# JMS Destination Types

- Queue
  - Persistent messages (stored until consumed)
  - Time decoupling between message producers & consumers
  - Usually employed for point-to-point communication

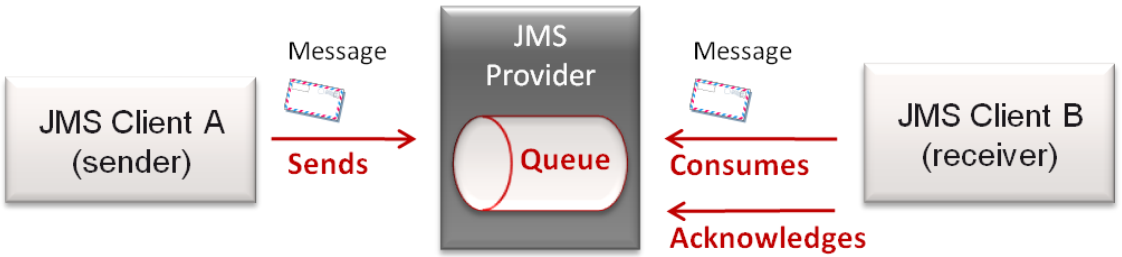
- Topic
  - Non-persistent messages (not stored, forwarded upon arrival; unless specific config.)
  - Time coupling between message producer & consumer
    (unless using the configuration option: **durable subscription**)
  - Usually emplyed for Publish-Subscribe communication

- *Note: JMS allows using all combinations of the two Destination types (queue & topic) and the two message reception types (sync. & async.)*

# JMS Communication Models (1 /2)

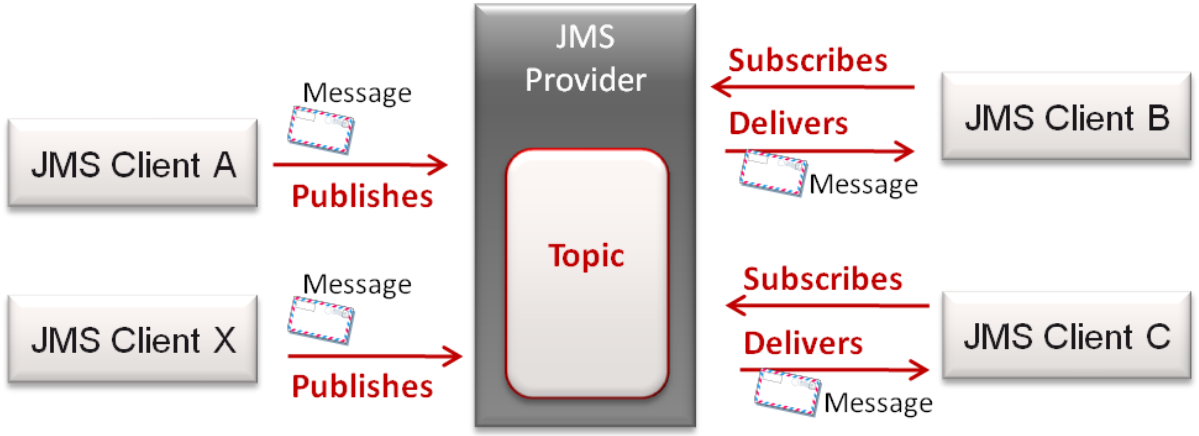
## Point-to-Point

- Each message is stored in a Queue until its targeted receiver reads it (or it expires)
- Each message is consumed once and only once, by a single receiver
- No time dependency between the message sender and receiver
- The receiver acknowledges the received message



# JMS Communication Models (2 /2)

- Publish / Subscribe
  - Each message can have several consumers
  - Time dependency between the message producer and consumers
    - Consumers only receive messages produced after their subscription
    - Consumers must remain available to receive messages (from the topics they subscribed to) (consumers can also create “durable” subscriptions to avoid this availability constraint)



# JMS Client – How to initialise communication?

- Use a Connection Factory
  - Handles the connection with the JMS/MOM provider.
  - Encapsulates connection parameters as defined by the administrator.
- Start a Session
  - A mono-task context
  - Used for sending and receiving messages
  - Manages several message consumers and producers

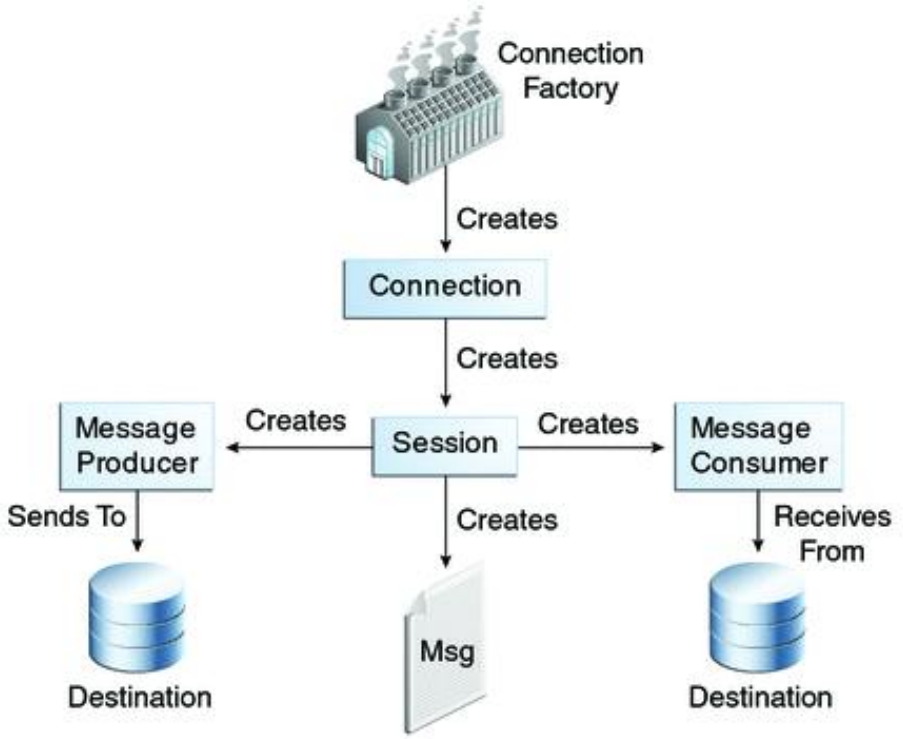


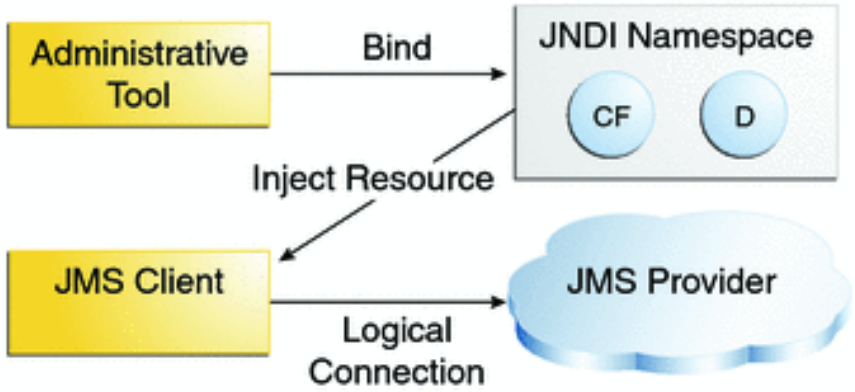

Image from Oracle's JMS tutorial
http://download.oracle.com/javaee/6/tutorial/doc/bnceh.html

# JMS Client – How to Obtain the Connection Factory? and the Destinations?

- The MOM administrator:
  - Creates the Administered Objects: Connection Factories (CF) & Destinations (D)
  - Registered the Administered Objects with a naming & directory service JNDI (bind operation)

  via an Administration Tool

  (specific to each MOM provider)




- JMS Clients:
  - Inquire the JNDI service to obtain the Administered Objects– FC, D
  - Use the Connection Factories to get MOM connections
  - Use Destinations to send/receive messages

Image from Oracle's JMS tutorial
http://download.oracle.com/javaee/6/tutorial/doc/bncdx.html

# JMS – Link with other Java APIs

- JNDI: Java Naming and Directory Interface
- JTA: Java Transaction API

# JMS Providers

- **Sun Java System Message Queue** (Sun → Oracle)
  - JMS integrated within Sun's Enterprise Application Server (GlassFish)
  - http://www.oracle.com/us/products/middleware/application-server/oracle-glassfish-server/index.html
- **MQ JMS** (IBM)
  - One of the market leaders
  - http://www7b.software.ibm.com/wsdd/library/techtip/0112_cox.html
- **WebLogic JMS** (BEA → Oracle)
  - Enterprise Application Server – can support JMS
  - http://download.oracle.com/docs/cd/E13222_01/wls/docs92/messaging.html
- **JMSCourier** (Codemesh)
  - Applications in C++ and JMS
  - http://www.codemesh.com/en/AlignTechnologyCaseStudy.html
- **TIBCO** http://www.tibco.com/products/soa/messaging
- **Fiorano Software** http://www.fiorano.com
- **JRUN Server** http://www.allaire.com
- **GemStone** http://www.gemstone.com
- **Nirvana** http://www.pcbsys.com
- **Oracle** http://www.oracle.com
- **…**
- **Vendor lists**
  - http://www.techspot.co.in/2006/06/jms-vendors.html
  - http://adtmag.com/articles/2003/01/31/jms-vendors.aspx

- **Joram**
  - **http://joram.ow2.org/**
  - **Free software (LGPL licence)**
  - **Developed by ScalAgent (since 1999)**

***Used for the Labs***

# JMS References

- JMS homepage (Sun → Oracle)
  - http://java.sun.com/products/jms → http://www.oracle.com/technetwork/java/jms/index.html
- JMS specifications
  - http://www.oracle.com/technetwork/java/jms-101-spec-150080.pdf
- JMS Tutorial
  - http://download.oracle.com/javaee/1.3/jms/tutorial
- JMS API FAQs
  - http://www.oracle.com/technetwork/java/faq-140431.html
- Gopalan Suresh Raj, September, 1999
  - http://www.execpc.com/~gopalan/jms/jms.html
- Richard Monson-Haefel & David A. Chappell, “Java Message Service”, O’Reilly, January 2001


# Part 3

Code Exemples

using the JMS API
(http://download.oracle.com/javaee/6/api)

# Exemples Overview

- # 1: JMS Producer
  - Producing messages
  - Towards both types of JMS Destination : Queue & Topic
- # 2: JMS Consumer → Receiver
  - *Synchronous / blocking* message reception
  - From any of the two JMS Destination types: Queue ou Topic
- # 3: JMS Consumer → Subscriber & Listener
  - *Asynchronous, non-blocking* message reception
  - From any of the two JMS Destination types: Queue ou Topic

# Example 1 : Producer (1)

```
import javax.jms.*;
import javax.naming.*

public class Producer{

  public static void main(String[] args)
    {
        //admin objects
        Context context = null;//jndi context
        ConnectionFactory factory = null;//jms connection factory

        //naming configs
        String factoryName = "ConnectionFactory";
        String destName = ...;
        Destination dest = null;

        //jms vars
        JMSContext jmsContext = null;//connection to destination
        JMSProducer jmsProducer = null;//producer

        //message vars
        int count = ...;
        String text = "Message ";
```

# Example 1 : Producer (2)

```
try{
  // create the JNDI initial context
  context = new InitialContext();

  // look up the ConnectionFactory
  factory = (ConnectionFactory)context.lookup(factoryName);

  // look up the Destination (queue or topic)
  dest = (Destination)context.lookup(destName);

  //close intitialContext
  context.close();

} catch(NamingException ex){...}
```

# Example 1 : Producer (3)

```
//create the jms context
//-> replaces connection & session in JMSv1.0
jmsContext = factory.createContext(<user-name>, <psw>);

// create the producer
jmsProducer = jmsContext.createProducer();

//send the <text> message for <count> times
for (int i = 0; i < count; ++i) {
      jmsProducer.send(dest, text);
      System.out.println("Sender:: sent message " +
            text + " " + i + " to destination " + destName );
}

//close JMSConext
jmsContext.close();
```

# Example 2 : Consumer via receive() → sync. (1)

```
import javax.jms.*;
import javax.naming.*;

//synchronous consumer from a destination
public class Receiver{

    public static void main(String[] args)
    {
        //admin objects
        Context context = null;//the jndi initial context
        ConnectionFactory factory = null;

        //naming configs
        String factoryName = "ConnectionFactory";
        String destName = ...;
        Destination dest = null;

        //jms
        JMSContext jmsContext;// connection to destination
        JMSConsumer receiver = null;//synchronous receiver

        //
        int count = ...;
```

# Example 2 : Consumer via receive() → sync. (2)

```
try{
  // create the JNDI initial context
  context = new InitialContext();

  // look up the ConnectionFactory
  factory = (ConnectionFactory)context.lookup(factoryName);

  // look up the Destination
  dest = (Destination)context.lookup(destName);

  //close intitialContext
  context.close();
catch(NamingException ex){...}
```

# Example 2 : Consumer via receive() → sync. (3)

```
//create the jms context
jmsContext = factory.createContext(<username>, <psw>);

// create the message receiver
receiver = jmsContext.createConsumer(dest);

//receive <count> number of messages
String textMessage;
for (int i = 0; i < count; ++i){
        //receive message synchronously -> block until msg or timeout
        //receiver params: message type, time-out in ms
        textMessage = receiver.receiveBody(String.class, 10000);
        System.out.println("Received: " + textMessage + " " + i);
}

//close JMSConext
jmsContext.close();
```

# Exeample 3 : Consumer via subscribe → async. (1)

```
import javax.jms.*;
import javax.naming.*;

//asynchronous consumer from a destination
public class Subscriber {

  public static void main(String[] args){

        //admin objects
         Context context = null;//jndi initial context
         ConnectionFactory factory = null;

         //naming configs
         String factoryName = "ConnectionFactory";
         String destName = ...;
         Destination dest = null;

         //jms
         JMSContext jmsContext;//connection to destination
         JMSConsumer subscriber = null;
```

# Example 3 : Consumer via subscribe → async. (1)

```
try{
  // create the JNDI initial context
  context = new InitialContext();

  // look up the ConnectionFactory
  factory = (ConnectionFactory)context.lookup(factoryName);

  // look up the Destination
  dest = (Destination)context.lookup(destName);

  //close intitialContext
  context.close();
catch(NamingException ex){...}
```

# Example 3 : Consumer via subscribe → async. (1)

```
//create the jms context
jmsContext = factory.createContext(<username>, <psw>);

// create the subscriber
subscriber = jmsContext.createConsumer(dest);

//set listener
subscriber.setMessageListener(new MsgListener());

System.out.println("Subscriber Ready ...");

//close JMSConext
//jmsContext.close();
```

What is this?...

# Example 3 : Consumer via subscribe → async. (1)

```
import javax.jms.*;

public class MsgListener implements MessageListener {

  @Override
  public void onMessage(Message message) {
      try {
            System.out.println(
                  message.getBody(String.class));
      } catch (JMSException e) {
            e.printStackTrace();
      }
  }
}
```


# Conclusion

MOM & JMS

# Actors & Usage

## Numerous products

- IBM WebsphereMQ (MQSeries)
- SonicMQ (→ Progress Software)
- BEA WebLogic (→ Oracle)
- Microsoft Message Queuing (MSMQ)
- Amazon Simple Queue Service (SQS)
- Joram (→ ScalAgent)
- ~ZeroMQ (not a MOM)
- …

## Usage

- Large Information Systems (IS) – 'historic' users
  - E.g. Banks, insurance companies, ...
  - Need to scale-up
- EAI (Enterprise Application Integration)
  - Integrating existing systems
  - Connexion to Data Bases (DB)
  - Managing several protocols
- …

# Conclusion

- MOMs are
  - ~ Easy to implement
  - ~ Easy to deploy
  - ~ Highly configurable
  - ~ Scale-up
- BUT
  - Are still tool-boxes
  - Rather complex to use


# Annexes

...

# Various examples of message-based communication

- Sockets
- MOM
- Other approaches – ex. ZMQ (Zero MQ), Akka, …

# Minimal implementation

- *nc* (netcat) : “TCP/IP swiss army knife”
  - Sockets manipulation tool
  - Allows setting-up message-based communication with shell command lines ell
  - Using pipes, sockets & shell language

On one terminal :

```
nana.enst.fr$ nc -l 2222
```

On another terminal :

```
yse.enst.fr$ echo bla | nc nana.enst.fr 2222
```

- Then perl, awk, .. To construct/analyse messages

# ZeroMQ

- Intermediary solution between:
  - Direct socket manipulation
    => flexibility
  - MOM usage
    => facilitates usage but set-up difficulties and performance costs
- Library that facilitates socket manipulation for setting-up a message-oriented distributed system
- http://www.zeromq.org

- ...