IBM TRAINING

# M32

# Introduction to JMS and XMS Application Programming

Stephen Rowles – rowles@uk.ibm.com

IBM
TRANSACTION & MESSAGING
TECHNICAL CONFERENCE

Atlanta, GA

June 12-16, 2006

RETURN TO INDEX

N

O

T

E

S

Stephen Rowles is a Software Engineer within the WebSphere MQ department at Hursley. His current role is team lead and developer of the WMQ Explorer.

## Agenda

- **JMS Overview,**
  - **specification version 1.02**
  - **specification version 1.1**
- **JMS with WebSphere MQ**

## Agenda

N O T E S

This presentation is an introduction to programming messaging applications using the Java Message Service (JMS) API and a non-Java equivalent XMS.

We start with an overview of JMS, the interfaces provided and how they are used to perform messaging, we will then also be looking into how WebSphere MQ implements JMS, and what you need to do to set up WebSphere MQ to support a JMS application.

We will then look at the similar API XMS.

You should not need a detailed understanding of MQ to follow this presentation but knowledge of MQ would certainly help.

## JMS – Java Message Service

- JMS is the standard Java API for messaging
  - point-to-point messaging domain
  - publish/subscribe messaging domain
- Vendor-independent Messaging API in Java
  - Specification owned by Sun
  - Managed by The Java Community Process
  - Expert Group includes IBM
- Part of Java 2 Enterprise Edition 1.3 and 1.4
- Defines package of Java Interfaces
  - Provides provider-independence
  - Does not provide provider interoperability

## JMS – Java Message Service

N
O
T
E
S

JMS is the standard Java API for applications to use in order to perform messaging.

It treats messaging as being in two “domains” corresponding to two ways of using messaging: point-to-point and publish/subscribe.

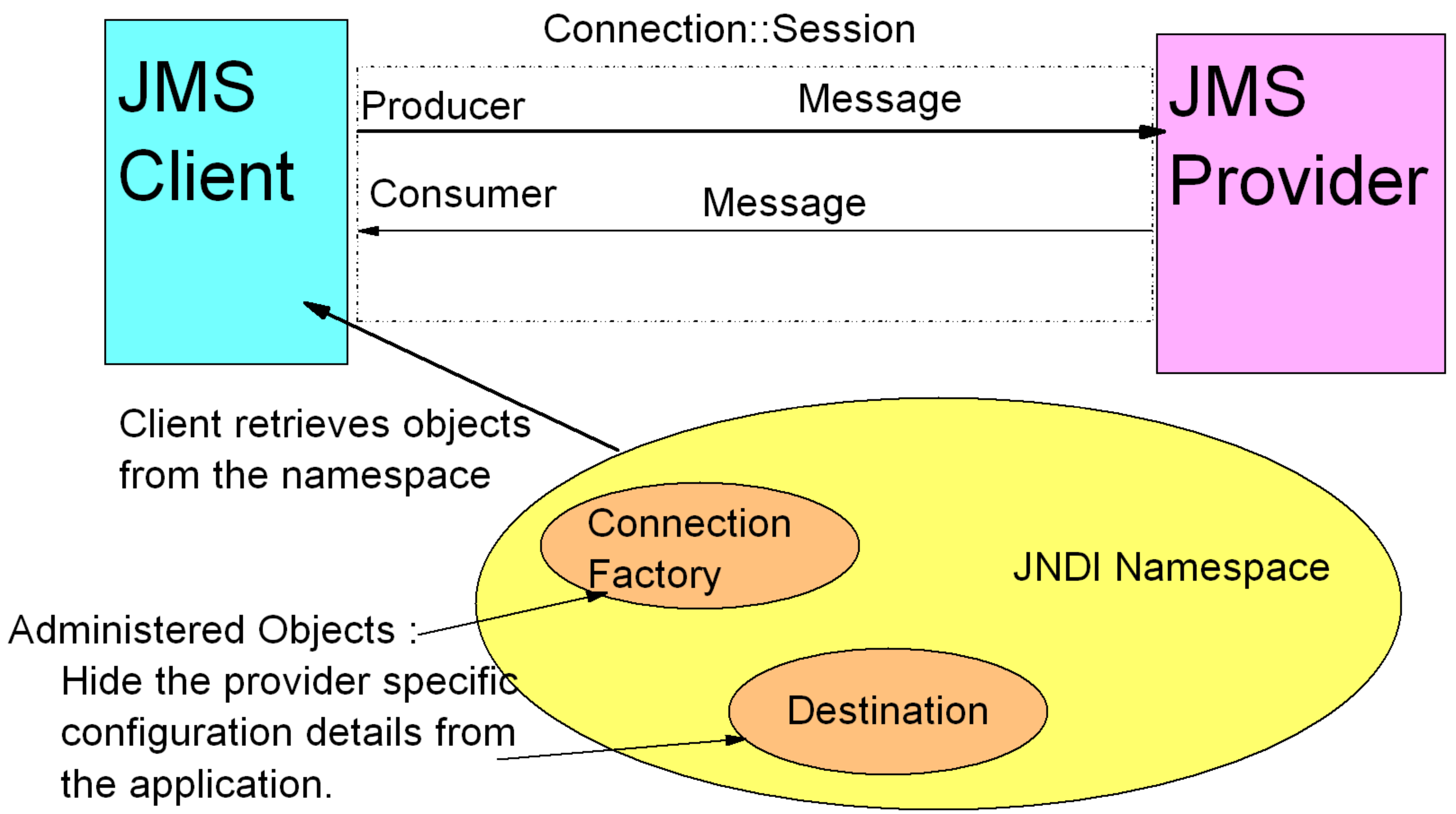
The specification was developed by Sun Microsystems with help from IBM, other messaging vendors, and other application server vendors.

The standard defines a package of Java Interfaces, to be implemented by the messaging vendor, or “provider” to use the J2EE terminology. This means that applications written to the JMS API are provider-independent, and can be redeployed between vendor products without modification.

The standard does not define the communications protocol to be used by the vendor to implement the messaging service. There is therefore no interoperability between vendor products: if you want to send a JMS message to another application, then that application must be using the same JMS product as you.

A JMS implementation is a mandatory part of Java 2 Enterprise Edition 1.3 onwards (J2EE) Compliant application server vendor’s are obliged to include a JMS implementation in their product.

## JMS Key Concepts


JMS Client
Connection::Session
Producer
Message
Consumer
Message
JMS Provider
Client retrieves objects from the namespace
Connection Factory
JNDI Namespace
Administered Objects : Hide the provider specific configuration details from the application.
Destination

## JMS Key Concepts

N O T E S

This slide provides an overview of the key concepts in JMS.

The application using the JMS API is a JMS Client. The product providing the JMS implementation is termed a JMS Provider.  The application connects to the JMS Provider by creating a Connection, and then, using this Connection, opens a Session.

Having opened a Session, the Client can then send and receive messages to Destinations – Topics and or Queues. Sending messages is achieved by using an object called a Producer, and receiving messages is done using a Consumer object.

JMS achieves provider-independence, so that applications can be ported across vendor products without change, by storing objects that have vendor-specific properties in a directory. These objects are ConnectionFactories and Destinations.

The ConnectionFactory, which is used to generate Connection objects, holds the properties that the Connection objects will have. These properties are provider specific and define the endpoint details for the Connection.

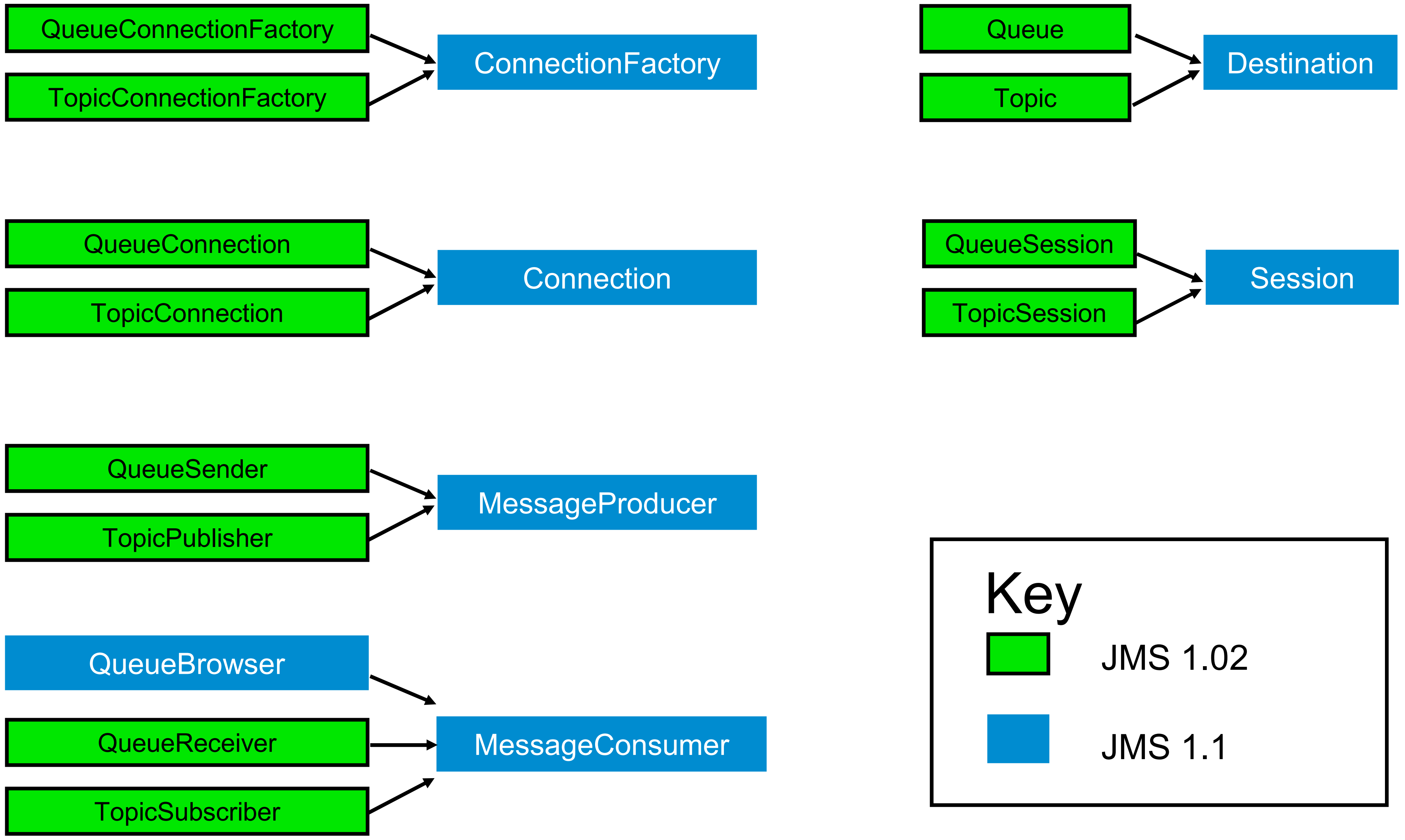
A Destination object is either a Queue or a Topic and holds provider specific information about the address of this destination in the provider messaging implementation.

A Queue is used for point to point messaging and a Topic is used for Publish/Subscribe.

A JMS client can retrieve ConnectionFactory and Destination objects from the directory using another standard Java API: Java Naming and Directory Interface (JNDI).

A JMS Client is not obliged to use JNDI; it can create provider-specific ConnectionFactory and Destination objects directly in the application code; however, in this case the application code will obviously need to change if it is to use a different Provider.

# JMS Interfaces (Unified Domain)



## JMS Interfaces

N O T E S

This slide shows the main Java Interfaces in JMS.

The JMS Provider supplies classes which implement these interfaces.

As described on the previous slide, a JMS Client starts with a ConnectionFactory, which is usually created by an administrator, and stored in a JNDI directory.

All the JMS objects are either retrieved from the JNDI directory (ConnectionFactory and Destinations), or are created from objects initially retrieved. Thus the JMS Client is able to code using only the JNDI API and the JMS Interfaces, it never needs to know the implementation classes; this is how provider-independence is achieved.

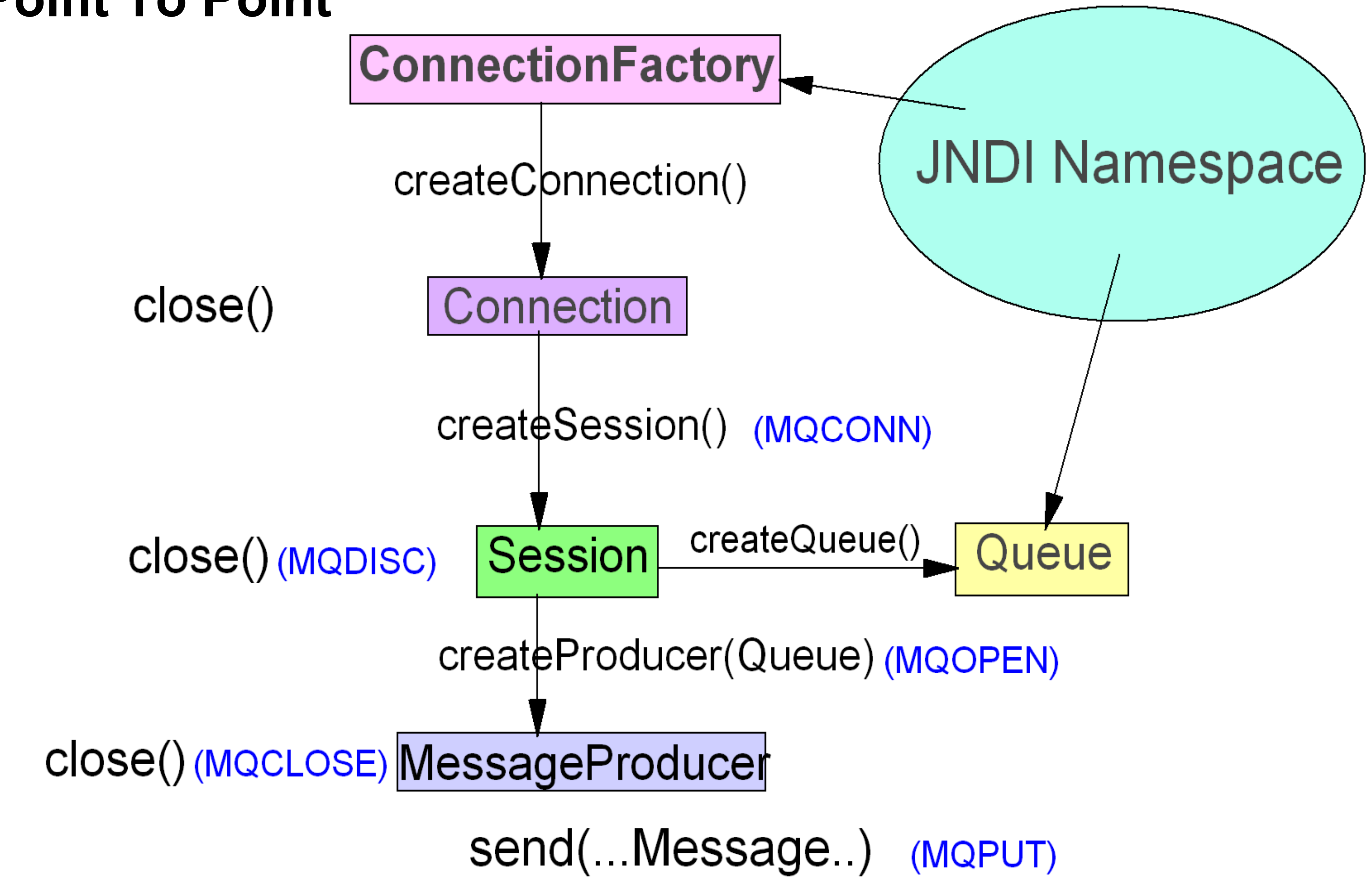
The previous specification of JMS (Version 1.02b) defined the 2 messaging domains, point-to-point and publish/subscribe as being completely independent of each other.

The latest version JMS 1.1 includes these interfaces for doing point-to-point and publish subscribe messaging independently, but also specifies use of the parent interfaces as 'cross messaging domain' interfaces. These parent interfaces allow JMS applications to create Connections and Sessions to a single provider endpoint, as in one connection. Therefore sessions created on this connection can use either one or both of the messaging domains.

The type of messaging that is actually used with a cross domain JMS Connection and Session, is decided ultimately by the Destination type used by the Consumer and Producer objects.

Use of a Queue object implies point to point messaging and a Topic implies Publish/Subscribe messaging.

## oint To Point



## Point To Point

N O T E S

This slide shows an application flow in more detail using the JMS 1.1 cross domain interfaces.

It shows the method calls made against each java object and also the corresponding MQI call that corresponds to the JMS object.

The application starts by calling createConnection, against a ConnectionFactory, this returns a Connection object. The point to point specific interfaces and methods to use are QueueConnectionFactory, createQueueConnection(). This will create a QueueConnection, which is the same as the Connection object but can only be used for point to point messaging.

The application then calls createSession against the Connection to get a Session object. To use the point to point messaging domain interfaces you would call createQueueSession() on the QueueConnection object. Note that if you start with a ConnectionFactory object that is specific to a messaging domain, every object that you will need to create from its connections, will also be specific to that messaging domain.

Creating a Session is equivalent to MQCONN in the MQI – it results in a physical connection to the queue manager being opened. The application now needs to take a Destination object – This will be either a Queue or a Topic and will define the messaging domain behaviour of any objects created on this Session.

A Queue object as shown in the diagram means we will be using the point to point messaging domain provided by WMQ.

Destination objects are normally obtained using JNDI – and passed it to the createProducer() method of the Session. This is equivalent in MQI terms to MQOPEN: the specified Queue is opened for PUT. Alternatively, the application could have called createConsumer(), in which case the queue would have been opened for GET.

The application can now call send, passing Message objects, to do the equivalent of an MQPUT.

When the application has finished sending messages to that Queue, the application can call the close() method on the MessageProducer, equivalent to an MQCLOSE of the Queue.

When it’s done sending messages to any Queue, the application can call close on the Session (equivalent to an MQDISC), followed by close on the Connection.

There are a couple more steps here than in the MQI, but the basic pattern is the same. The explanation for the extra steps is:

The ConnectionFactory has no equivalent in the MQI, but is needed to allow JMS applications to be provider-independent.

The MQI has no separation of Connections and Sessions. In JMS, a Connection is typically JVM-wide, and supports concurrent access, whereas a Session does not support concurrent access, and so is typically used only from a single thread.

## JMS Point To Point Sample Code

```
//Either
                                        //retrieve connection factory from context (JNDI)
ConnectionFactory factory = (ConnectionFactory)context.lookup(conn_name);
//or
                                        //create and initialize administered object in code
MQConnectionFactory localFactory = new MQConnectionFactory();
localFactory.setQueueManager(qmgrname);
                                        // create connection
Connection connection = factory.createConnection();
                                        //start connection otherwise no messages can be received
connection.start();
                                        //Obtain session
Session session = connection.CreateSession(transacted, acknowlegeMode);
                                        //retrieve destination from context
Queue queue = (Queue)context.lookup(queue_name);
                                        //create sender/receiver
MessageProducer messageProducer = session.CreateProducer(queue);
MessageConsumer messageConsumer = session.CreateConsumer(queue);
                                        //send and receive message
TextMessage outMessage = session.createTextMessage();
messageProducer.send(outMessage);
Message inMessage = messageConsumer.receive();
```

# JMS Point To Point Sample Code

N O T E S

Here is the code for what that flow may look like.

The ConnectionFactory is normally obtained using the Context.lookup() call. The return type of the lookup call is Object so it must be explicitly cast to the expected ConnectionFactory interface.

The actual type of the object will be the class that is the providers implementation of the ConnectionFactory interface.

You can create a Connection factory directly in application code, but this requires use of the provider specific class to implement the ConnectionFactory interface.

You then call the createConnection method on your factory object to create a Connection. The JMS specification defines that connections are created in the stopped state and so must be explicitly started.

A Session is then created. The two parameters are to say whether this session is to be transacted or not, and then the type of message acknowledgement that will be used.

Before you can create the message producer and consumer, we need to get a destination object. This application is using the point to point messaging domain so we need to retrieve a Queue with another lookup call. This queue destination is passed in to the createProducer() and createConsumer() methods on the Session.

Now we need a Message to send. I will talk more about JMSMessages in a minute, but you just need to call the relevant createMessage() method on the session. You can then pass the Message to the send method of the MessageProducer.

To receive messages, you would call the receive method on the message consumer.

## JMS Sessions

- Session / QueueSession / TopicSession

- Connection.createSession(boolean Transacted,
  int Acknowledge_Mode);
  - Trasacted attribute
    - true / false
  - Acknowledge_Mode :
    - AUTO_ACKNOWLEDGE
    - DUPS_OK_ACKNOWLEDGE
    - CLIENT_ACKNOWLEDGE
    - SESSION_TRANSACTED

## JMS Sessions

N

O

T

E

S

The JMS Session is created from the Connection object.

The createSession() method takes 2 parameters : transacted and AcknowledgeMode.

Transacted state for this entire session is either true or false and affects all messageProducers and MessageConsumers created on this session.

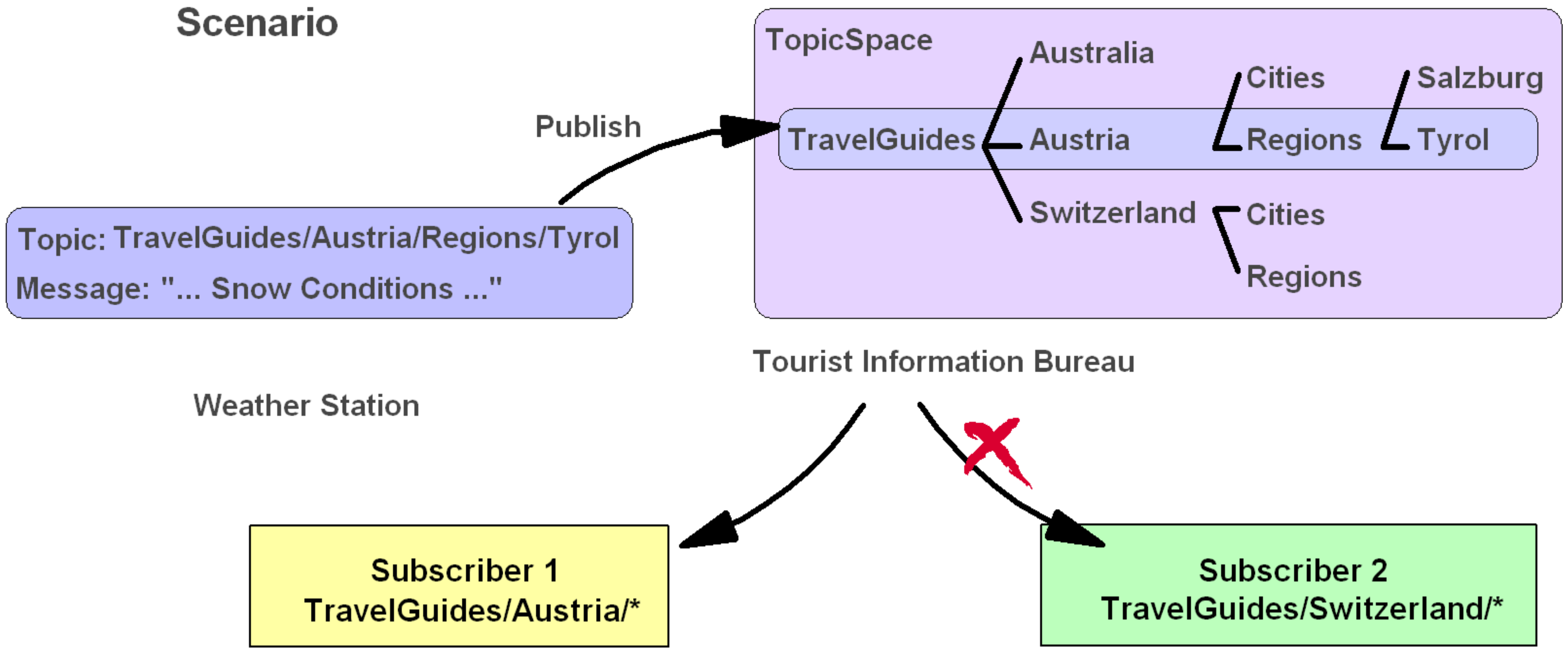
If you decide to use a transacted session, you must call the commit() or rollback() method call to complete/undo units of work. No messages sent will be hardened to queues if a commit is  not called after sending messages, and if you do not commit after receiving messages, those message will be available again on the queue when your Connection is closed.

The acknowledge mode relates to how messages received by this sessions messageConsumers are to be acknowledged.

AUTO_ACKNOWELDGE and DUPS_OK_ACKNOWLEDGE both will automatically acknowledge the receipt of messages on the message consumers (sessions) behalf. The difference between them is that there is a small window with DUPS_OK where a message may be delivered twice. This is because the acknowledge call is issued by the client implementation on behalf of the MessageConsumer when it receives the next message, whereas AUTO_ACK acknowledges the message just before it returns it on the receive() method call.

If you specify CLIENT_ACKNOWLEDGE, then you/the application will need to call the acknowledge() method on the message that is received to acknowledge it receipt. If you do not do this the message will be available again on the queue when you close your Connection object.

# Publish Subscribe



Messages delivered to all interested subscribers

Subscriptions can be "durable" or "non-durable"

## Publish Subscribe

N O T E S

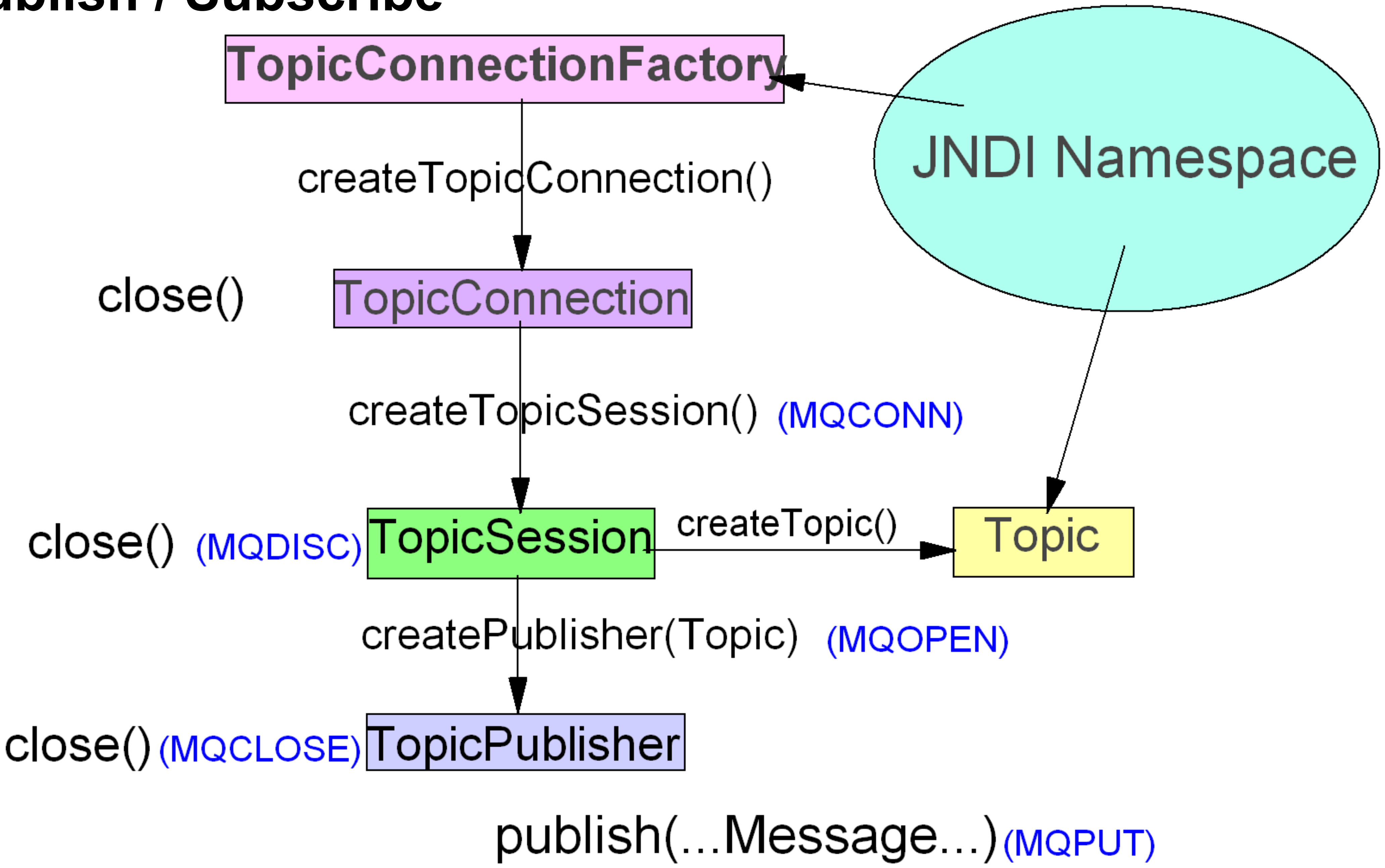
So, far we have talked only of point-to-point messaging, so now let’s turn to publish/subscribe. In publish/subscribe instead of sending a message to a Queue destination, you send – or publish it to a Topic.

Topics are organized hierarchically into a Topic Space. Instead of issuing individual receive calls to retrieve messages, you issue a subscribe. You will then be passed messages for your subscription until you unsubscribe. You can subscribe to a particular Topic, or you can use wildcards, as in the two examples here, to subscribe to all Topics at a particular point in the hierarchy.

In JMS, subscriptions can be either “durable” or “non-durable”. A non-durable subscription is one that lasts as long as the subscriber is available to receive messages (or until it unsubscribes). You can think of a non-durable subscription is being one in which an unsubscribe is automatically issued on behalf of the subscriber when it terminates, either normally or abnormally.

For a durable subscription, the subscriber passes a name that uniquely identifies the subscriber. If the application terminates, then on restart it can reissue the subscribe using the same identification. The JMS Provider must then deliver any messages to the subscriber which where published during its absence.

## ublish / Subscribe



## Publish / Subscribe

N O T E S

The application flow for a publish subscribe application is very similar to that for the point to point messaging.

This flow is using interfaces from the Publish/Subscribe messaging domain but to use Publish subscribe messaging with the cross domain interfaces, as shown in the previous point to point flow slide, all you need to do is use the exact same objects and methods, but with a Topic object instead of a Queue.

The underlying MQI call equivalents remain the same and the Broker Subscription and publish headers are compiled by JMS on the applications behalf.

# JMS Publish/Subscribe Sample Code

```
                                        //retrieve connection factory from context (JNDI)
TopicConnectionFactory topicfactory = (TopicConnectionFactory)context.lookup(tcf_name);
                                        //alternatively create and initialize administered object in code
MQTopicConnectionFactory localFactory = new MQTopicConnectionFactory();
localFactory.setQueueManager(qmgrname);
localFactory.setClientId("Client_ID");
                                        // create connection
TopicConnection topicConnection = factory.createTopicConnection();
                                        //start connection otherwise no messages can be received
topicConnection.start();
                                        //Obtain session
TopicSession topicSession = connection.CreateTopicSession(transacted, acknowlegeMode);
                                        //retrieve destination from context
Topic topic  = (Topic)context.lookup(Topic_name);
Topic topic2 = topicSession.createTopic("TOPIC_NAME");
                                        //create sender/receiver
TopicPublisher publisher = session.CreatePublisher(topic);
TopicSubscriber durableSubscriber = session.CreateDurableSubscriber(topic, "Sub_name");
                                        //send and receive message
TextMessage outMessage = session.createTextMessage();
messageProducer.publisher(outMessage);
Message inMessage = durableSubscriber.receive();
topicSession.unsubscribe("Sub_name");
```

## JMS Publish/Subscribe Sample Code

N O T E S

Here is the code for what that flow may look like.

The TopicConnectionFactory (a publish/subscribe specific ConnectionFactory) is normally obtained using the Context.lookup() call. The return type of the lookup call is Object so it must be explicitly cast to the expected TopicConnectionFactory interface.

The actual type of the object will be the class that is the providers implementation of the TopicConnectionFactory interface.

You can create a Connection factory directly in application code, but this requires use of the provider specific class to implement the TopicConnectionFactory interface.

You then call the createTopicConnection method on your factory object to create a Connection. The JMS specification defines that connections are created in the stopped state and so must be explicitly started.

A Session is then created. The two parameters are to say whether this session is to be transacted or not, and then the type of message acknowledgement that will be used.

Before you can create the message producer and consumer, we need to get a destination object. This application is using publish/Subscribe with the JMS1.02b domain specific publish subscribe interfaces so we can only use a topic. This topic destination is passed in to the createProducer() and createConsumer() methods on the Session.

Now we need a Message to send. I will talk more about JMSMessages in a minute, but you just need to call the relevant createMessage() method on the session. You can then pass the Message to the publish method of the TopicPublisher.

To receive messages, you would call the receive method on the durableSubscriber.

With durableSubscribers you must explicitly unsubscribe them. The unsubscribe() method is provided on the session object.

## JMS Message Interfaces

- **BytesMessage : Unformatted binary data**
  - Session.createBytesMessage(new byte[]);
- **TextMessage : Character data**
  - Session.createTextMessage(“String data”);
- **StreamMessage : Sequence of typed data fields**
  - Session.createStreamMessage();
- **MapMessage : Collection of typed data fields**
  - Session.createMapMessage();
- **ObjectMessage : Serialized Java Object**
  - Session.createObjectMessage();

N
O
T
E
S

# JMS Message Interfaces

A nice feature of JMS is the set of message classes used. These isolate an application from the internal representation of its data.  These are the 5 message interfaces defined by the JMS specification: BytesMessage, TextMessage, StreamMessage, MapMessage and ObjectMessage.

BytesMessages hold unformatted binary data. They have read<type> and write<type> methods for each of the Java built in types : boolean, int, long, double etc. They also have read/writeBytes methods that operate on an array of bytes, and read/writeObject methods which work on any built in types.

JMS does not define any interpretation of a bytesMessage contents so it is up to the sending /receiving applications to agree on the correct order of read and write calls.

TextMessages hold character data, they have methods setText and getText, which take and return a String object.

StreamMessages hold a sequence of typed data fields. It is similar to a BytesMessage except that it does hold an interpretation of the contents of the message and will throw an exception if the consumer does not issue a valid sequence of reads – But the consumer does not have to issue the reads in the same sequence as the producer issued write method calls : Stream message will not throw an excepiton if there is a valid Java type conversion between the write and read types eg: writeInt() and readLong()

MapMessages hold a collection of typed data fields. The fields are stored as a set of name-value-pairs where the names are strings and the values are of Java primitive types, Producers and consumers do not have to agree on the order of reads and writes with MapMessages but they must agree on the names and the types of objects to be represented.

ObjectMessages can be used to send any serializable Java object. It just has methods setObject and getObject(). This is convenient if the receiving application is a JMS application and very inconvenient if not !

## Message Headers and Properties

- Fixed Header

| | |
|---|---|
| •JMSMessageID | JMSCorrelationID |
| JMSTimestamp | JMSDeliveryMode |
| JMSPriority | JMSExpiration |
| JMSRedelivered | JMSReplyTo |
| JMSDestination | JMSType |

- Properties
  - JMSXUserid, JMSXAppId, JMSXDeliveryCount
  - Provider Extensions
  - User-defined

## Message Headers and Properties

N O T E S

Although JMS does not define the wire format of a message, it does define messages as being composed of a fixed header, a variable length header and a body.

The body holds the data represented by the message types detailed on the last slide.

However it chooses to represent the message on the wire, the JMS provider must carry the defined fixed header fields in the message. I won't go into too much detail on each of the fields but they do relate to the MQMD header in WMQ Messages.

The properties part of the message holds name-value-pairs, like the body of a MapMessage. There are 3 different kinds of properties:

- JMS defines several standard properties that are in effect optional header fields.

- Providers can add their own specific properties

- Applications can add any additional properties they choose

The generic message interface, from which all these messages inherit, provides the methods required for setting and getting all of these properties.

Many of the header fields and properties, for example JMSTimestamp are set automatically by the provider when the message is sent.

## Message Selectors

- Request filtering of messages by provider

- Based on message properties or header fields
  - outMessage.setStringProperty(“Postcode”, “SO21 2JN”);

- Specified by MessageConsumer
  - Session.createReceiver(queue, “Postcode LIKE ‘SO%’”);

## Message Selectors

N O T E S

An important feature of JMS is the ability to set message selectors, which allow MessageConsumers to instruct JMS to filter messages, and only deliver messages arriving at the Destination which meet the selector expression.

The syntax for the selector expression is defined by the JMS specification based on a subset of SQL92.

The expression can refer to message properties and to JMS header fields.

Note that the selector associated with a Receiver is set when the receiver is first created and cannot be changed.

Depending on the way the provider manages the destination, and the complexity of the selector, receiving a message using a message selector can be an expensive operation in terms of performance.

## Asynchronous Message Delivery

- Application implements MessageListener
  - Provider calls onMessage() method
- MessageListener registered with MessageConsumer
- ExceptionListener registered on Connection

## Asynchronous Message Delivery

N O T E S

Another feature of JMS, with no direct equivalent in the MQI, is the asynchronous message delivery.

Rather than an application always having to call the MessageConsumer.receive() method, it can ask the JMS provider to call back to the application code when a message becomes available.

To do this the application must provide a class which implements the JMS MessageListener interface and has a method "onMessage()".

Once the application has registered an instance of the MessageListener implementation with the MesssageConsumer, then whenever a message arrives at the destination for this messageConsumer, the onMessage() method will be called by the provider, passing the message as the argument on the method call.

There are a couple of points to watch when using MessageListeners:

When registered, a message listener will mark the session as asynchronous. The application should not use 'synchronous' message consumers with this session.

Since there is no application code surrounding the receive call (the provider effectively makes the receive call on your behalf) the application cannot code a try catch block to catch Exceptions resulting from a receive failure. This is overcome by implementing another JMS interface ExceptionListener, and registering an instance of this ExceptionListener with the Connection. The provider will then call the ExceptionListeners onException() method if an exception occurs.

## Message Listener Sample Code

```
import javax.jms.*;
public class MyListener implements MessageListener
{
   //called by JMS when a message is available
   public void onMessage(Message message)
   {
     //application specific processing here
            doStuff();
   }
}
//in main program, register MessageListener
MyListener msgListener = new MyListener();
queueReceiver.setMessageListener(msgListener);

//Register ExceptionListener, implementation details as MessageListener,
//implement the onException method
MyExceptionListener expListener = new MyExceptionListener();
connection.setExceptionListener(expListener);

//main program can continue with other application specific behaviour
```

# Message Listener Sample Code

N O T E S

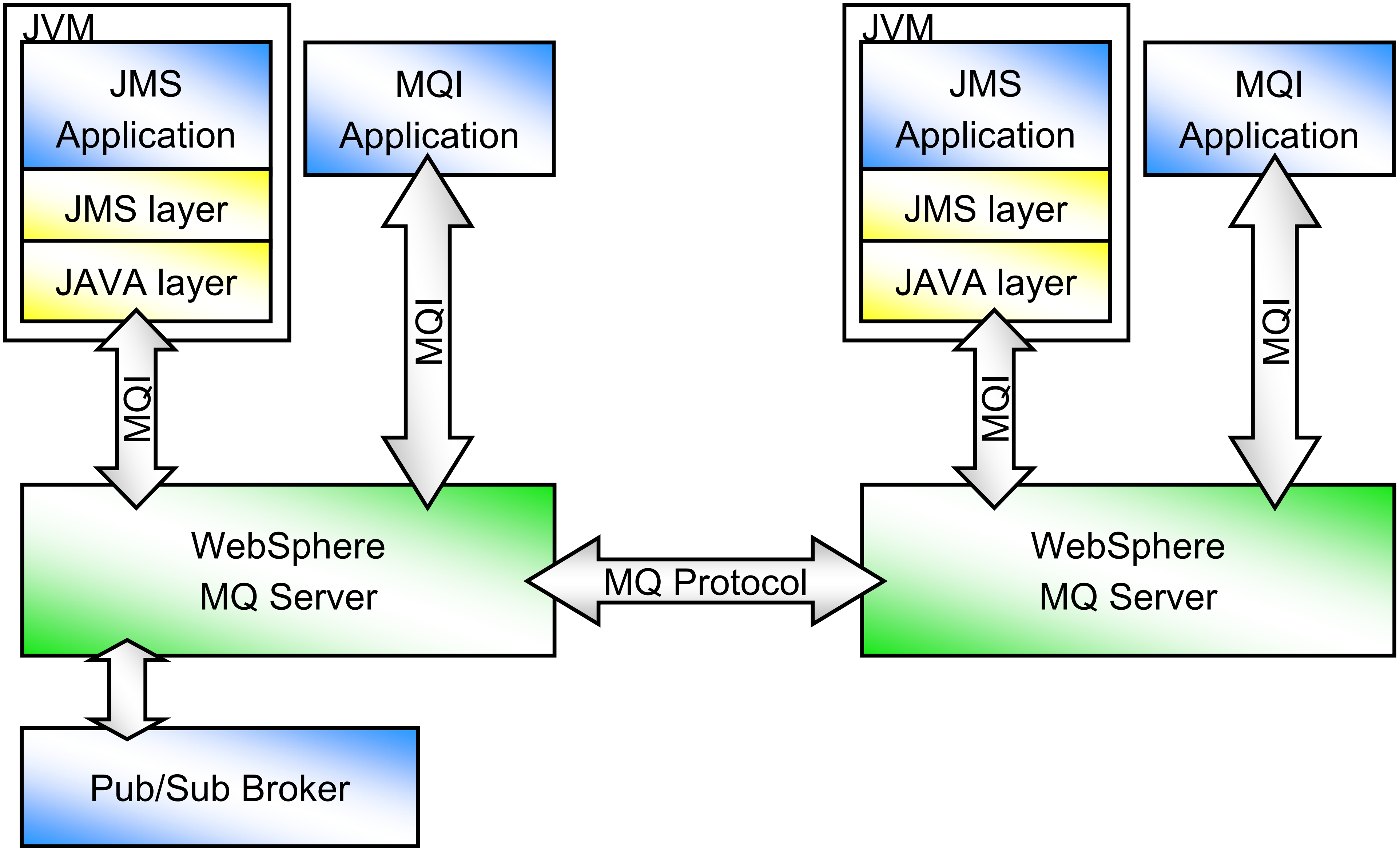
Here is some sample code to show how a MessageListener can be used.

The onMessage() method will contain the application code to deconstruct the message – and possible to send a reply message.

The message Listener is registered with the QueueReceiver object ( – a point to point specific MessageConsumer object.)

The Exception Listener will always be registered with the underlying Connection object, though it usual that the class that's implements a MessageListener will also implement an ExceptionListener to catch any exceptions that may occur whilst JMS is receiving the message on behalf of the application.

## WebSphere MQ and JMS



## WebSphere MQ and JMS

N
O
T
E
S

WebSphere MQ JMS support is actually implemented by mapping the JMS API to the pre existing WMQ API, MQI.

JMS applications can connect direct to the queue manager using “bindings” via the Java Native interface (JNI), or they can connect using TCP/IP sockets.

With these java sockets – known as a “client” connection, the JMS application is pure Java and can be used in web applications like applets.

The key point to be made on this slide is that JMS applications, when using WebSphere MQ as the provider, can interoperate with other WMQ applications, that is applications written in other programming languages that use other WMQ API’s.

The other WMQ application need not be aware that its communication partner is written to the JMS API.

## JMS vs MQ Classes for Java

- Some MQ functions not present in JMS
  - eg: Distribution Lists
- JMS has features not present in other MQ
  - eg: Message Selectors
- MQ API is more complex, but offers more control
- JMS is a simpler, higher-level API
- JMS is portable between providers

# JMS vs MQ Classes for Java

N O T E S

As well as JMS, the traditional MQ API (MQI) is also available to Java developers in the MQ Classes for Java.

These Java classes offer some functions that are not available in JMS such as distribution lists and message segmentation.

JMS can only offer functionality that is common across all providers but where possible MQ specific support has been added via the administered objects – connection factories and destinations, in order to keep applications vendor neutral.

**JMS features that are not available in MQ include:**

Message selectors

Direct support for publish/subscribe – in MQ you will need to manually construct the publication and subscription messages.

Message classes and Java serialization

Asynchronous delivery

Administered objects

Vendor independence

Generally MQ API is more complex, but can offer more control of the underlying message service.

JMS is a simpler, high level API which requires less detailed expertise on messaging, thereby allowing developers to concentrate on writing business logic.

It can be that this abstraction can have some overheads in terms of required memory and machine cycles

## WebSphere MQ JMS Implementation

- Built into WebSphere MQ 5.3 onwards
- AIX, HP, iSeries, Linux (Intel & zSeries), Windows, z/OS, Solaris
- Publish Subscribe with the addition of
  - Product extension MA0C
  - WebSphere MQ Integrator
  - WebSphere Business Integration Event and Message Broker

## WebSphere MQ JMS Implementation

N O T E S

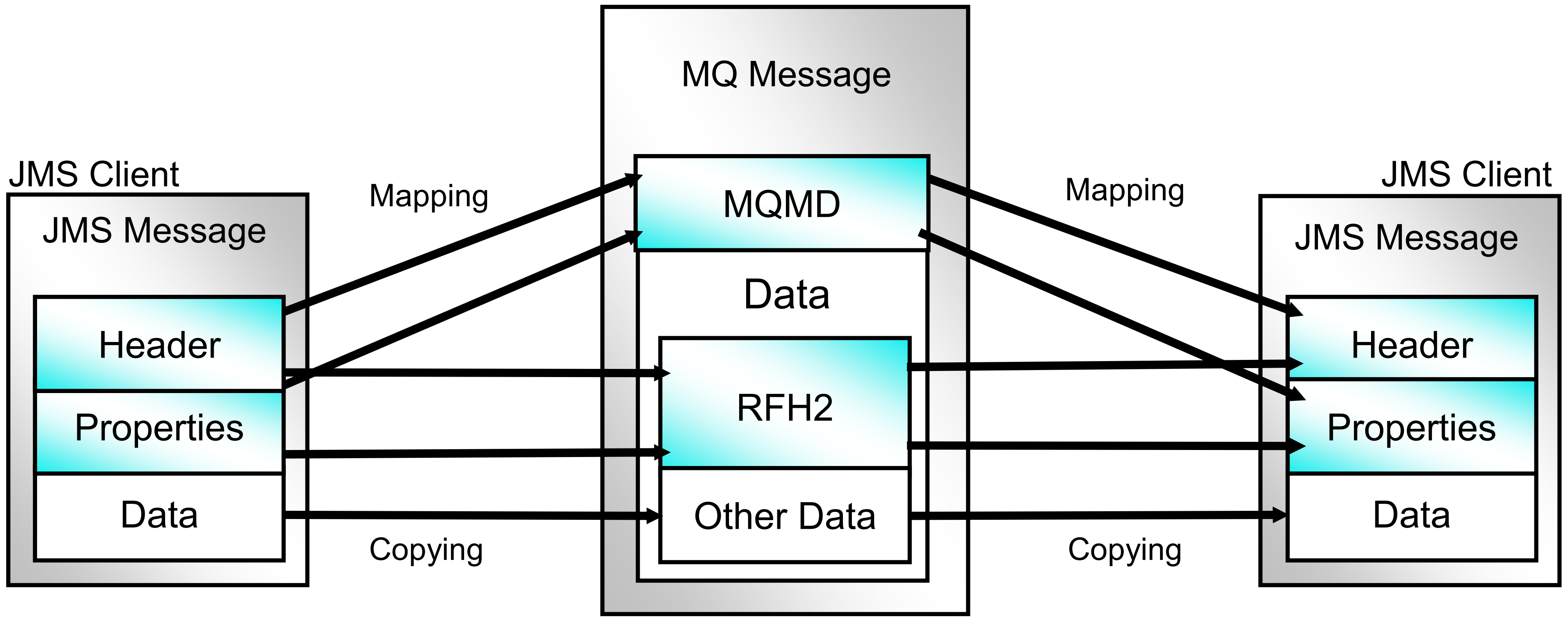
Prior to WebSphere MQ version Java support was provided by SupportPac MA88.

In WebSphere MQ verison 5.3, the installation on windows requires the user to select a custom install and select the Java/ JMS option.

For version 6 JMS and MQ classes for Java will always be installed as part of the server.

The Product extension MAOC is also known as the Fuji broker and provides publish subscribe capability on the Queue Manager.

## Message Header Mapping



## Message Header Mapping

N O T E S

JMS Message objects are represented as MQ Messages as shown in this picture.

The JMS Header and properties are mapped to the MQMessage MQMD header where they have equivalents. The other properties, that have no MQMD equivalent, are mapped to a header in an internal format known as the RFH2 header, this immediately follows the MQMD header.

RFH2 is an XML format structure, it is also used by WebSphere MQ Integrator.

When receiving the message the reverse happens, where there are MQMD properties that have no JMS equivalent, they are mapped into JMS provider specific properties.

The JMS Message body is copied unchanged to the MQMessage body following the RFH2 header.

Many existing non JMS applications are not able to interpret the RFH2. For this reason it is possible to not use this header format by setting a flag on the Destination object. The Destination being an administered object can hold provider specific information without affecting the portability of the JMS application.

## JMS Message Persistence

- **WebSphere MQ Persistence**
  - **Messages will be delivered once and only once.**
- **JMS 1.1 specification Persistence**
  - **Somewhere between WMQ Persistence and WMQ Non Persistence**
  - **Option where WMQ Non persistent messages are able survive the restart of the QueueManager.**
- **JMS1.1 specification Non Persistence**
  - **Uses WMQ non persistent messages**

## JMS Message Persistence

N
O
T
E
S

The JMS specification defines 2 modes of persistence for JMS Messages, Persistent and Non persistent.

Both of these are implemented with the WMQ JMS, JMS non persistent messages relate directly to WMQ Non Persistent messages.

The persistence option for JMS Messages, will transfer that message to a WebSphere MQ persistent message. This may provide more availability than is required for some JMS applications, so as well as this there is an option for JMS Messages using MQ to get the minimum persistence as defined by the JMS specification.

The main difference between these persistent options is that in the JMS specification, it can be interpreted that for a JMS Message to be persistent it must survive the restart of the provider – ie. the WMQ Queue Manager.

In WMQ Persistent messages are highly available and will be delivered once and only once. This can have an effect on the performance of persistent message throughput as many writes to logs are required to ensure this delivery.

So another persistence option is available to be set via the Queue Destination object that allows the persistent messages to conform to the persistence as defined the JMS Specification.

## Setting up WebSphere MQ and JMS

- Classpath
  - To include : jms.jar providerutil.jar com.ibm.mq.jar com.ibm.mqjms.jar
  - Optionally depending on functionality required, include : jndi.jar ldap.jar  fscontext.jar jta.jar
- Path
  - To include the location of the bindings libraries
    - Windows default c:\Program Files\IBM\WebSphere MQ\Java\lib
    - Unix /opt/mqm/java/lib
- JNDI (jndi.jar)
  - File System  (fscontext.jar)
  - LDAP (ldap.jar)

# Setting up WebSphere MQ and JMS

N O T E S

The steps required to use JMS with WMQ.

Before you can run a JMS application, there are a number of environment setup tasks that must be performed

The classpath needs to include the following jars files:

| | |
|---|---|
| jms.jar, | from Sun defines the JMS interfaces |
| com.ibm.mqjms.jar | The IBM Provider JMS Implementation |
| com.ibm.mq.jar | the underlying MQ API |
| jndi.jar | for JNDI along with fscontext.jar and ldap.jar |
| jta.jar | for transactions |

The full list can be found in the WebSphere MQ Using Java book.

The system Path must be update to include the lib subdirectory where MQJMS was installed: /opt/mqm/java/lib or C:\Program Files\IBM\WebSphere MQ\Java\lib

This is for the JNI bindings support.

Finally you will need to setup a JNDI directory to host ConnectionFactory and Destination objects.

Supported directory services are:

The Sun reference implementation of JNDI (fscontext.jar) which reads and instantiates directory contents from a flat file. This is the easiest.

LDAP : You would use LDAP if you wanted to manage the JMS configuration centrally with other server resources.

## WebSphere MQ Administered Objects

- MQConnectionFactory
- MQXAConnectionFactory
- MQQueueConnectionFactory
- MQXAQueueConnectionFactory
- MQTopicConnectionFactory
- MQXATopicConnectionFactory
- MQQueue
- MQTopic

# WebSphere MQ Administered Objects

N O T E S

These are the WMQ JMS administered objects.

MQConnectionFactory is the MQ implementation of the JMS interface ConnectionFactory.

It defines the connection parameters for the underlying MQ Connection, such as the Queue manager name, channel name, port number and hostname.

For publish and subscribe applications it will also hold parameters for connecting to a broker such as the broker name and broker QueueManager.

Queue and TopicConnectionFactory are the messaging domain specific versions of ConnectionFactory and only hold the details relative to their messaging domain.

MQQueue implements the JMS Queue interface. It names the underlying MQ Queue to be used and other messaging properties such as persistence, default priority etc.

MQTopic implements the JMS Topic interface and holds the publish/subscribe specific destination address details.

MQXAConnectionFactory is the implementation of the XAConnectionFactory interface and be used by application servers to include WMQJMS in their global transactions.

## JMS Admin command line tool

- Use to populate the JNDI directory with the JMS administered objects
  - ConnectionFactories
  - Destinations (Queues / Topics)

- Syntax is verb + object (+ properties)
  - Define q(aQueue) qmgr(QM_NAME)
  - Alter QCF(QCF) transport(CLIENT)
  - Display t(topic)

# JMS Admin command line tool

N O T E S

WebSphere MQ JMS provides a command line tool called JMSAdmin to administer JNDI objects.

The administration tool enables administrators to define the properties of eight types of WebSphere MQ JMS object and to store them within a JNDI namespace. Then, JMS clients can use JNDI to retrieve these administered objects from the namespace and use them. The JMS objects that you can administer by using the tool are:

| | |
|---|---|
| MQConnectionFactory (JMS 1.1 only) | MQXAConnectionFactory (JMS 1.1 only) |
| MQQueueConnectionFactory | MQXAQueueConnectionFactory |
| MQTopicConnectionFactory | MQXATopicConnectionFactory |
| MQQueue | JMSWrapXAQueueConnectionFactory |
| MQTopic | JMSWrapXATopicConnectionFactory |

Invoking the administration tool

The administration tool has a command line interface. You can use this interactively, or use it to start a batch process. The interactive mode provides a command prompt where you can enter administration commands. In the batch mode, the command to start the tool includes the name of a file that contains an administration command script.

To start the tool in interactive mode, enter the command: JMSAdmin [-t] [-v] [-cfg config_filename]  where:

-t Enables trace (default is trace off)

-v Produces verbose output (default is terse output)

-cfg config_filename Names an alternative configuration file (see “Configuration” on page 42)

A command prompt is displayed, which indicates that the tool is ready to accept administration commands. This prompt initially appears as: InitCtx>

## JMSAdmin Command Line Tool Config

- JMSAdmin.config file provided to determine how to connect to the directory.

  - INITIAL_CONTEXT_FACTORY
    - eg: com.sun.jndi.fscontext.RefFSContextFactory
  - PROVIDER_URL
    - eg: file:C:\\JNDI-Directory
  - SECURITY_AUTHENTICATION
    - eg: CRAM-MD5   (LDAP only)

# JMSAdmin Command Line Tool Config

N O T E S

**INITIAL_CONTEXT_FACTORY**

The service provider that the tool uses. There are three explicitly supported values for this property: v com.sun.jndi.ldap.LdapCtxFactory (for LDAP) v com.sun.jndi.fscontext.RefFSContextFactory (for file system context) v com.ibm.websphere.naming.WsnInitialContextFactory (to work with WebSphere Application Server's CosNaming repository)

On z/OS and OS/390, com.ibm.jndi.LDAPCtxFactory is also supported and provides access to an LDAP server. However, this is incompatible with com.sun.jndi.ldap.LdapCtxFactory, in that objects created using one InitialContextFactory cannot be read or modified using the other.

**PROVIDER_URL**

The URL of the session's initial context; the root of all JNDI operations carried out by the tool. Three forms of this property are supported: ldap://hostname/contextname (for LDAP)

file:[drive:]/pathname (for file system context)

iiop://hostname[:port] /[?TargetContext=ctx] (to access base WebSphere Application Server V4 CosNaming namespace)

This support for the WebSphere Application Server version 4 Namespace is available only for that environment.

WebSphere Application Server Version 5 provides its own administration tools to set up the JMS administered object JNDI entries.

JMSAdmin will only be required to be used with WAS5 if you plan to use the Generic JMS Provider. It is also used to define JMS administered objects for the WebLogic V8 Foreign JMS provider.

**SECURITY_AUTHENTICATION**

Whether JNDI passes security credentials to your service provider. This property is used only when an LDAP service provider is used. This property can take one of three values:

none (anonymous authentication)

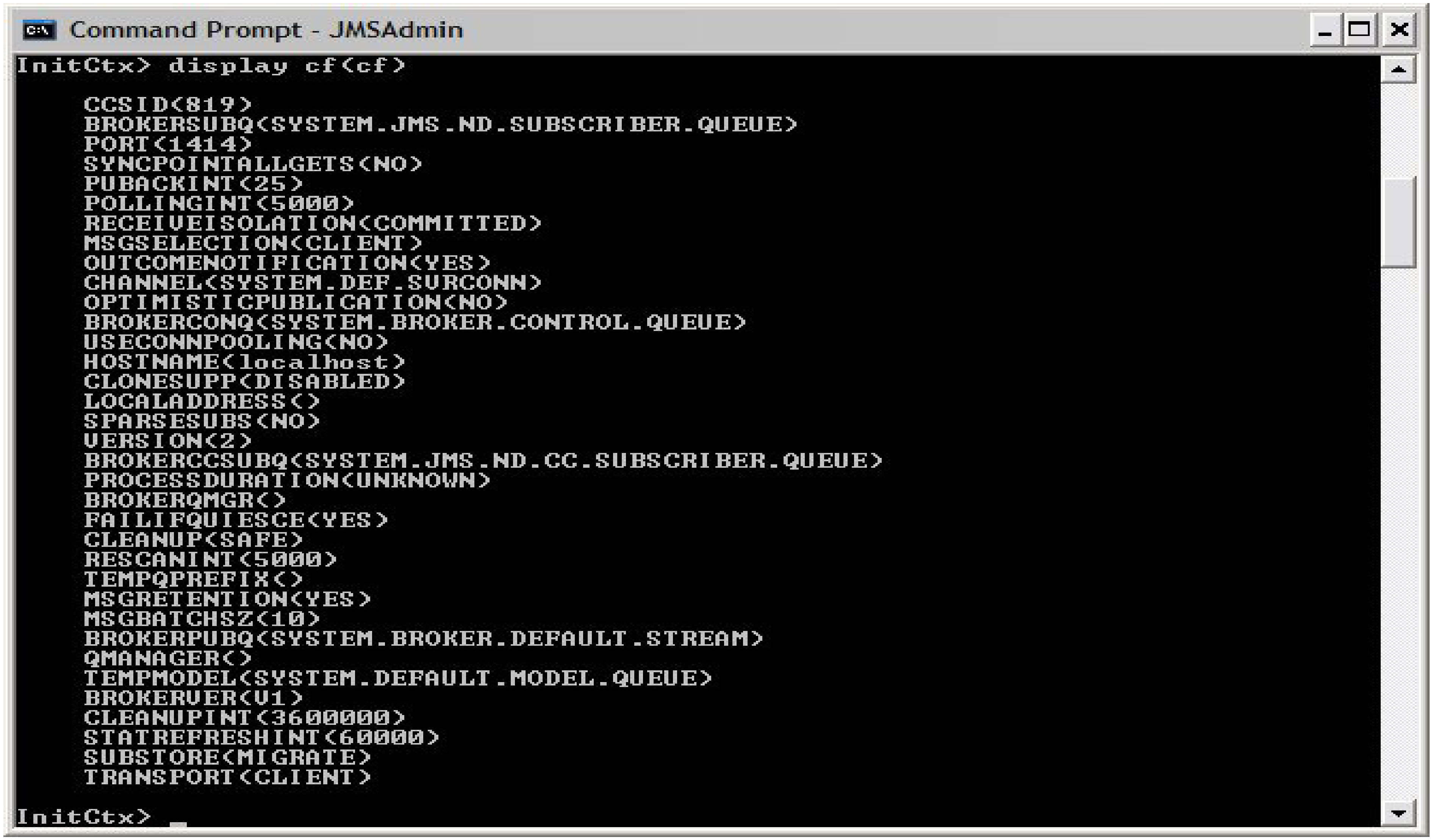
simple (simple authentication)

CRAM-MD5 (CRAM-MD5 authentication mechanism)

If a valid value is not supplied, the property defaults to none.

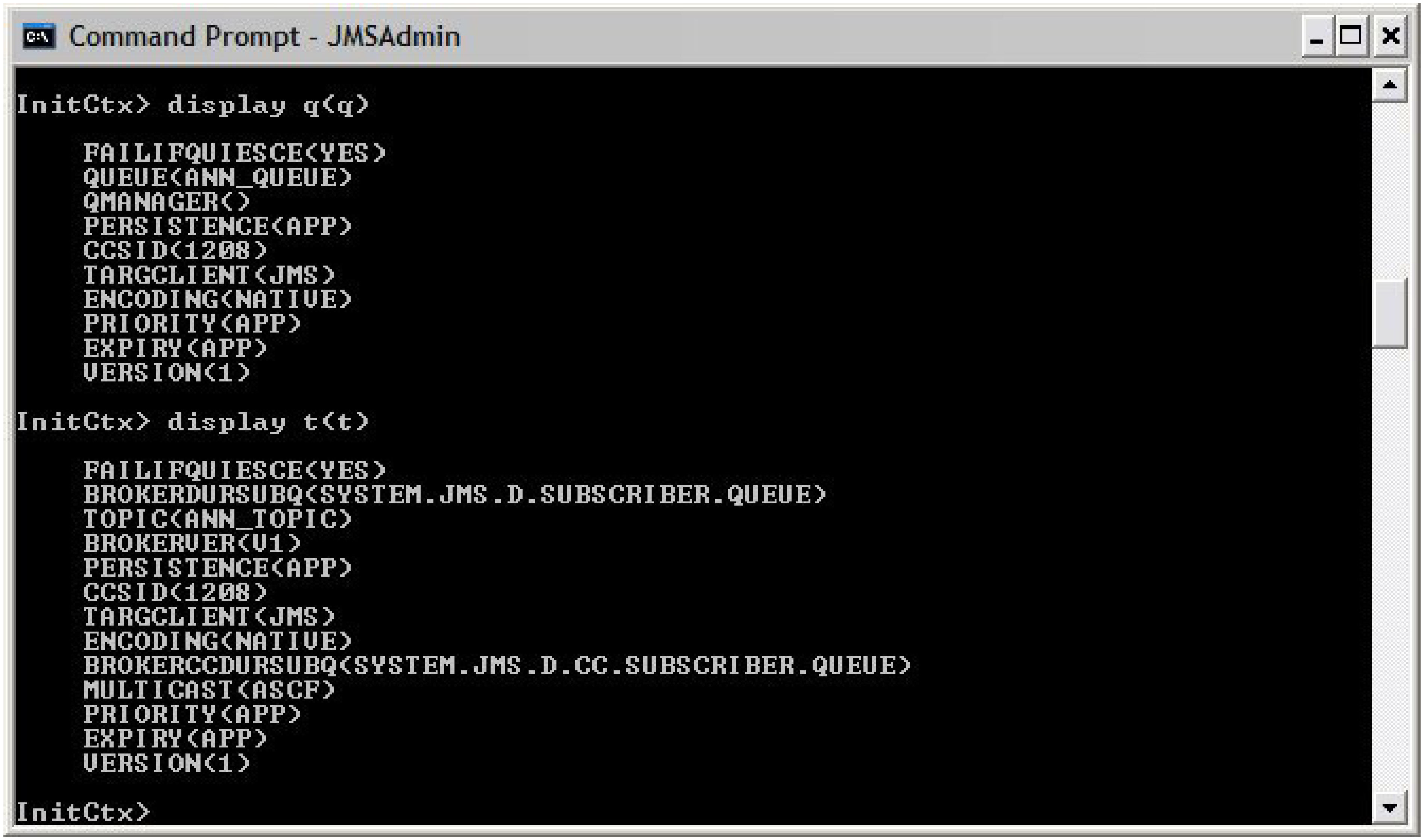
Full information on this tool can be found in the WebSphere MQ Using Java book Chapter 5.

# JMSAdmin ConnectionFactory



```
InitCtx> display cf(cf)

    CCSID(819)
    BROKERSUBQ(SYSTEM.JMS.ND.SUBSCRIBER.QUEUE)
    PORT(1414)
    SYNCPOINTALLGETS(NO)
    PUBACKINT(25)
    POLLINGINT(5000)
    RECEIVEISOLATION(COMMITTED)
    MSGSELECTION(CLIENT)
    OUTCOMENOTIFICATION(YES)
    CHANNEL(SYSTEM.DEF.SVRCONN)
    OPTIMISTICPUBLICATION(NO)
    BROKERCONQ(SYSTEM.BROKER.CONTROL.QUEUE)
    USECONNPOOLING(NO)
    HOSTNAME(localhost)
    CLONESUPP(DISABLED)
    LOCALADDRESS()
    SPARSESUBS(NO)
    VERSION(2)
    BROKERCCSUBQ(SYSTEM.JMS.ND.CC.SUBSCRIBER.QUEUE)
    PROCESSDURATION(UNKNOWN)
    BROKERQMGR()
    FAILIFQUIESCE(YES)
    CLEANUP(SAFE)
    RESCANINT(5000)
    TEMPQPREFIX()
    MSGRETENTION(YES)
    MSGBATCHSZ(10)
    BROKERPUBQ(SYSTEM.BROKER.DEFAULT.STREAM)
    QMANAGER()
    TEMPMODEL(SYSTEM.DEFAULT.MODEL.QUEUE)
    BROKERVER(V1)
    CLEANUPINT(3600000)
    STATREFRESHINT(60000)
    SUBSTORE(MIGRATE)
    TRANSPORT(CLIENT)

InitCtx> _
```

## JMSAdmin Destination Queue/Topic

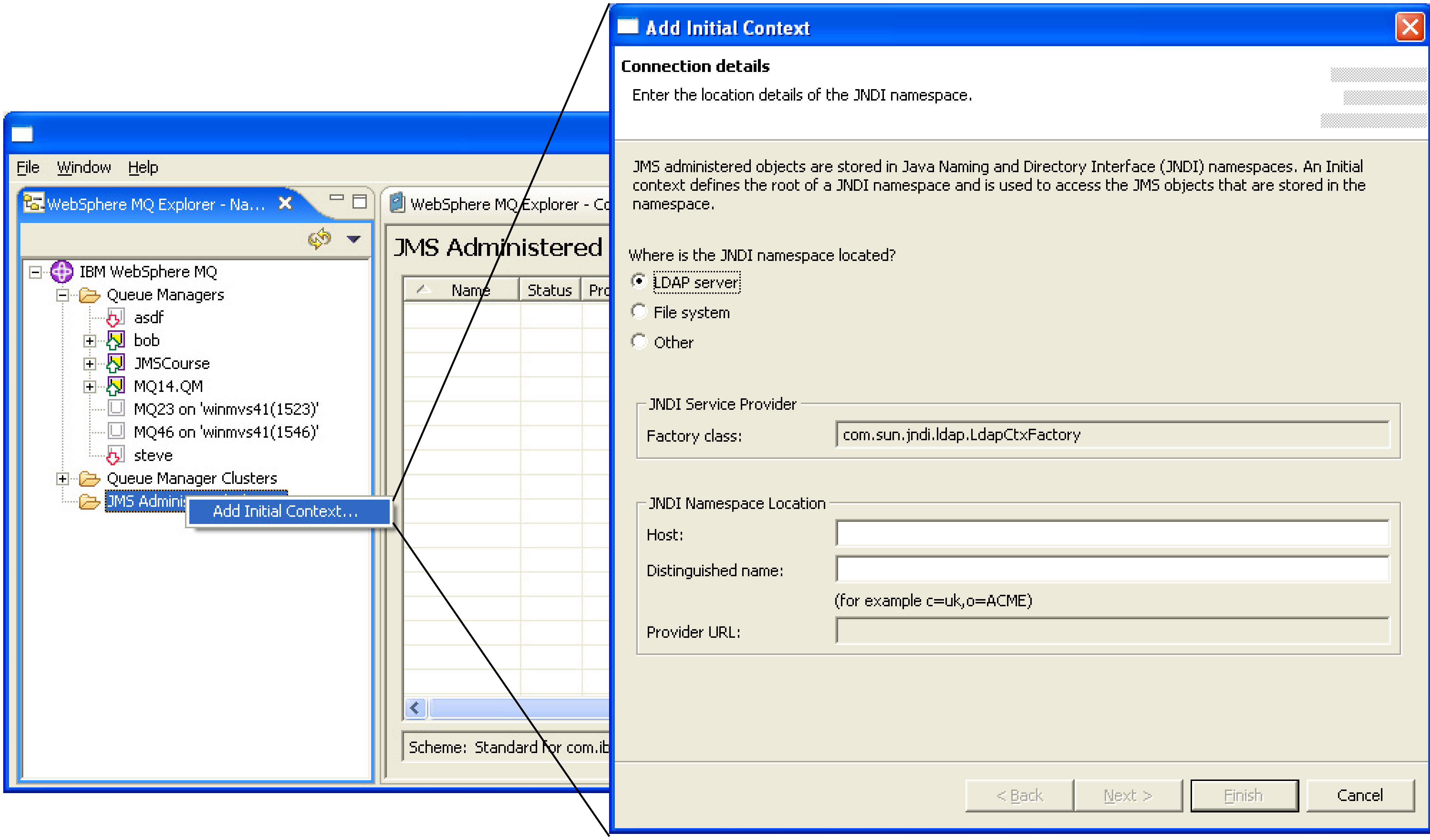

Command Prompt - JMSAdmin

```
InitCtx> display q(q)

    FAILIFQUIESCE(YES)
    QUEUE(ANN_QUEUE)
    QMANAGER()
    PERSISTENCE(APP)
    CCSID(1208)
    TARGCLIENT(JMS)
    ENCODING(NATIVE)
    PRIORITY(APP)
    EXPIRY(APP)
    VERSION(1)

InitCtx> display t(t)

    FAILIFQUIESCE(YES)
    BROKERDURSUBQ(SYSTEM.JMS.D.SUBSCRIBER.QUEUE)
    TOPIC(ANN_TOPIC)
    BROKERVER(V1)
    PERSISTENCE(APP)
    CCSID(1208)
    TARGCLIENT(JMS)
    ENCODING(NATIVE)
    BROKERCCDURSUBQ(SYSTEM.JMS.D.CC.SUBSCRIBER.QUEUE)
    MULTICAST(ASCF)
    PRIORITY(APP)
    EXPIRY(APP)
    VERSION(1)

InitCtx>
```

## JMS Admin extension to the MQ Explorer

- Graphical management of JNDI resource will be provided as a plug-in to the MQ Explorer.
  - Will provide management for initial contexts
    - File system based JNDI
    - LDAP
    - “other”

# Add an initial context



**Connection details**

Enter the location details of the JNDI namespace.

JMS administered objects are stored in Java Naming and Directory Interface (JNDI) namespaces. An Initial context defines the root of a JNDI namespace and is used to access the JMS objects that are stored in the namespace.

Where is the JNDI namespace located?

[✓ LDAP server]
File system
Other

JNDI Service Provider

Factory class: com.sun.jndi.ldap.LdapCtxFactory

JNDI Namespace Location

Host:

Distinguished name:

(for example c=uk,o=ACME)

Provider URL:

## Add an initial context

N

O

T

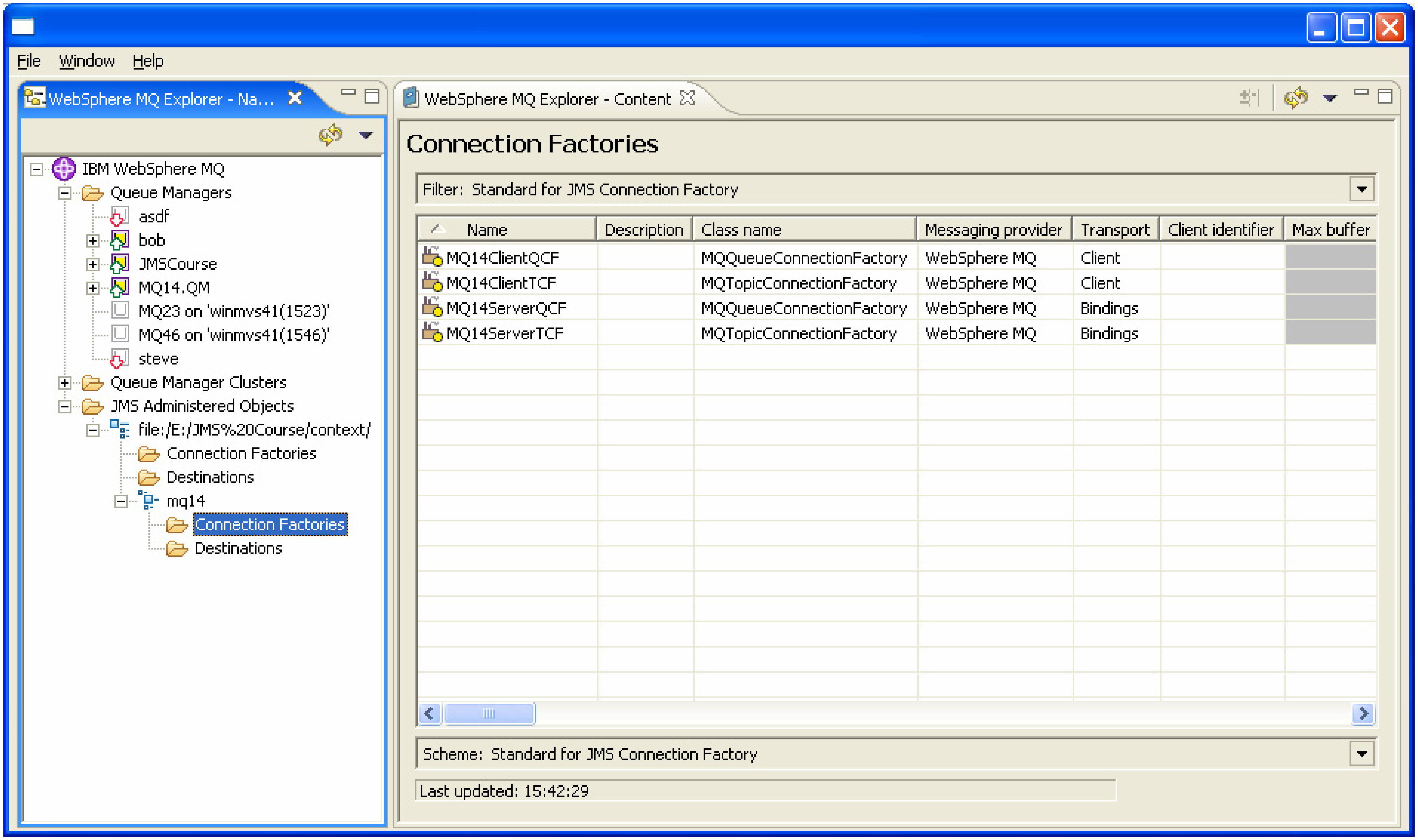
E

S

- The MQ Explorer will have a new tree node "JMS Administered Objects".

- Pressing right mouse button and selecting "Add Initial Context…" launches as wizard to allow the addition a new initial context.

# Viewing connection factories



Connection Factories

Filter: Standard for JMS Connection Factory

| Name | Description | Class name | Messaging provider | Transport | Client identifier | Max buffer |
|---|---|---|---|---|---|---|
| MQ14ClientQCF | | MQQueueConnectionFactory | WebSphere MQ | Client | | |
| MQ14ClientTCF | | MQTopicConnectionFactory | WebSphere MQ | Client | | |
| MQ14ServerQCF | | MQQueueConnectionFactory | WebSphere MQ | Bindings | | |
| MQ14ServerTCF | | MQTopicConnectionFactory | WebSphere MQ | Bindings | | |

Scheme: Standard for JMS Connection Factory

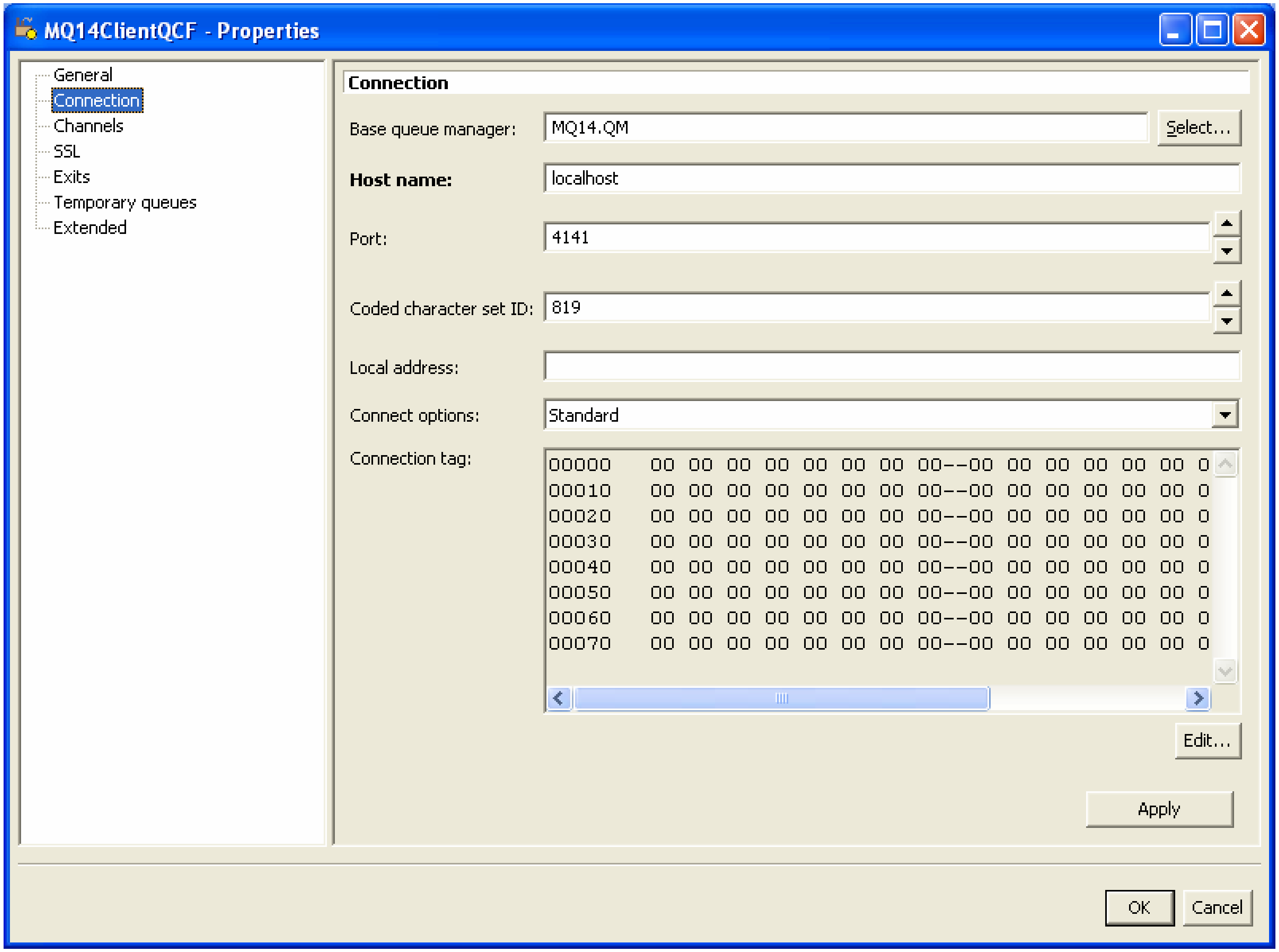
Last updated: 15:42:29

## Viewing connection factories

N O T E S

- After adding a new context, the Navigator view on the right hand side of the Explorer can be used to view the structure of the initial context and any sub contexts.
- Connection factories and Destinations are shown in separate folders and can be created, managed and deleted in a similar way to normal queue manager objects

# Properties of a connection factory



Connection

Base queue manager: MQ14.QM

**Host name:** localhost

Port: 4141

Coded character set ID: 819

Local address:

Connect options: Standard

Connection tag:

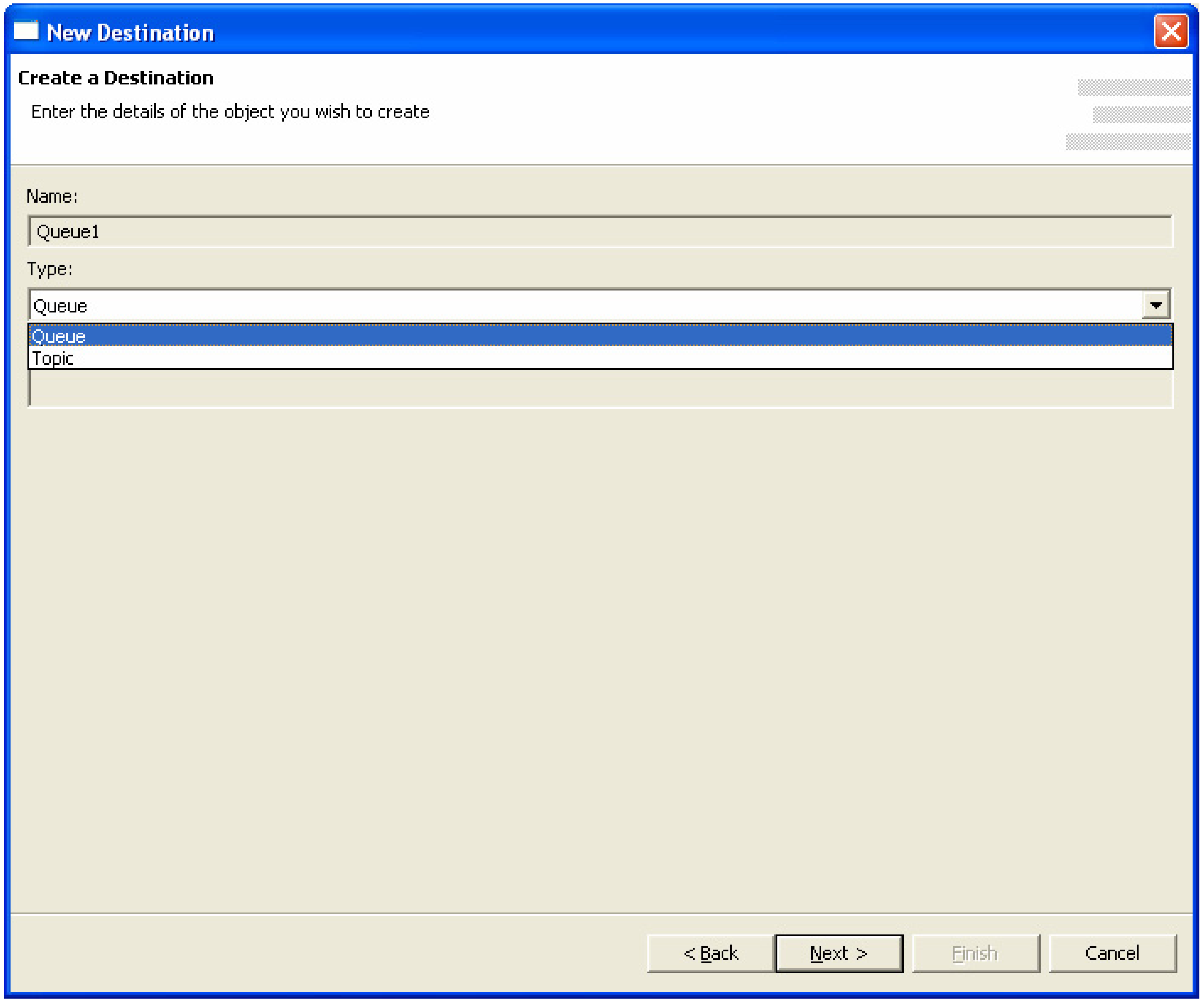
```
00000     00  00  00  00  00  00  00  00--00  00  00  00  00  00  0
00010     00  00  00  00  00  00  00  00--00  00  00  00  00  00  0
00020     00  00  00  00  00  00  00  00--00  00  00  00  00  00  0
00030     00  00  00  00  00  00  00  00--00  00  00  00  00  00  0
00040     00  00  00  00  00  00  00  00--00  00  00  00  00  00  0
00050     00  00  00  00  00  00  00  00--00  00  00  00  00  00  0
00060     00  00  00  00  00  00  00  00--00  00  00  00  00  00  0
00070     00  00  00  00  00  00  00  00--00  00  00  00  00  00  0
```

## Properties of a connection factory

N O T E S

- Properties of objects within the JMS Admin tool are very similar to the standard MQ object.
- The major change is the ability to select MQ objects from with the dialog. In this case there is a select button next to the “Base queue manager” field, this allows the user to select a queue manager from those known to the Explorer

## Create a destination



**Create a Destination**

Enter the details of the object you wish to create

Name: Queue1

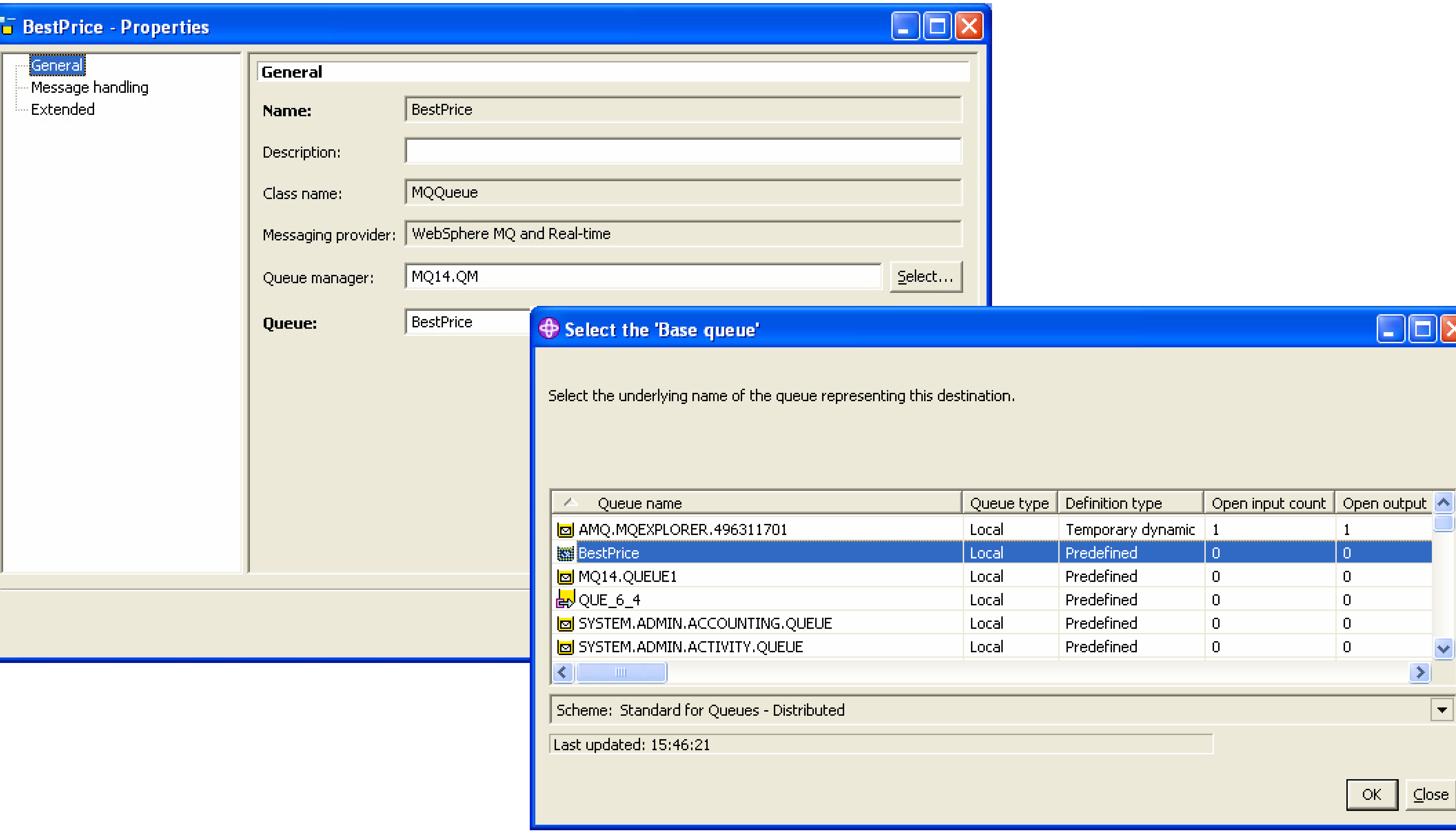
Type: Queue

## Create a destination

N O T E S

- Destinations can be created using the right mouse button on the Destinations folder, either a queue or topic destination can be created.

# Properties of a destination



**General**

| | |
|---|---|
| **Name:** | BestPrice |
| Description: | |
| Class name: | MQQueue |
| Messaging provider: | WebSphere MQ and Real-time |
| Queue manager: | MQ14.QM |
| **Queue:** | BestPrice |

Select the underlying name of the queue representing this destination.

| Queue name | Queue type | Definition type | Open input count | Open output |
|---|---|---|---|---|
| AMQ.MQEXPLORER.496311701 | Local | Temporary dynamic | 1 | 1 |
| BestPrice | Local | Predefined | 0 | 0 |
| MQ14.QUEUE1 | Local | Predefined | 0 | 0 |
| QUE_6_4 | Local | Predefined | 0 | 0 |
| SYSTEM.ADMIN.ACCOUNTING.QUEUE | Local | Predefined | 0 | 0 |
| SYSTEM.ADMIN.ACTIVITY.QUEUE | Local | Predefined | 0 | 0 |

Scheme: Standard for Queues - Distributed

Last updated: 15:46:21

## Properties of a destination

N

O

T

E

S

- When altering the properties of a destination, a Queue Manager from the Explorer.

- After a Queue Manager is selected, a list of queues available on that queue manager can be browsed and selected as the target queue for a queue destiniation

## WebSphere MQ Connection Pooling

- **WMQJMS implementation transparently pools underlying MQ connections**
- **JMS Session corresponds to an MQ HConn**
- **Sessions pooled in the scope of Connection**
- **Pool deactivated when Connection closed**

## WebSphere MQ Connection Pooling

N O T E S

WebSphere MQJMS support in WMQ 5.3 provides implicit pooling of underlying MQ connections.

This pooling is transparent to applications and can improve performance as MQ HConns can be reused by multiple JMS applications.

If required this can be disabled by setting the USECONNPOOLING property on the Connection Factory to No either by using JMSAdmin

```
InitCtx> alter cf(cf) USECONNPOOLING(NO)
```

or programatically

```
((MQConnectionFactory)connectionFactory.setUseConnectionPooling(false);
```

## WMQJMS and other WebSphere Products

- Publish Subscribe support
  - Product Extension MA0C
  - WebSphere MQ Integrator
  - WebSphere Business Integration Event/Message Broker
  - Built in support in WMQ V6 (MA0C built into the product)

- Application Server
  - WebSphere Application Server (Versions 4, 5, 5.1, 6)
  - WebSphere Studio Application Developer

## WMQJMS and other WebSphere Products

N O T E S

As mentioned previously Publish Subscribe support can be provided using  the WMQ MAOC product extension (Fuji Broker), or WebSphere MQ Integrator and Business Integration Message and Event broker products.

Support for J2EE compliant application servers can also be provided with WebSphere MQ JMS. For Version 5 of WebSphere Application Server, WMQJMS provides the default JMS Messaging capability.

For the latest release of WAS version6, this default functionality is now provided by WebSphere Platform Messaging with support to use WMQJMS instead as the JMS provider or in conjunction with.

## Publish Subscribe Options

- Queued
  - Uses Queue Manger and defined Queues to communicate with the Broker.
  - Client/Bindings
- Direct (IP, Real-Time)
  - Non-persistent, non-transactional, non-durable only
  - Connects directly to broker using fast TCP/IP protocol
  - Multicast, Reliable Multicast options

## Publish Subscribe Options

N O T E S

**Publish Subscribe Options :**

When connecting to a WebSphere Business Integration Event/Message Broker from a MQJMS application, you can either do so via an MQ Queue, or directly.

The transport type to be used is specified as a Transport option on the Connection Factory.

To use WMQ, you can specify Bindings or client as the transport type.

To connect direct to the broker you can use one of the Direct transport options.

Multicast support is also available with this type of connection but this must also be enabled on the topic via the Message Broker Toolkit.

These ‘direct/realtime’ options do not support transactions or persistence but they do offer a performance advantage.

**Publish/Subscribe Brokers :**

This is the set of in-service supported Pub/Sub brokers

WMQ MA0C brokers,

v2 MQ brokers,

WebSphere Business Integration v5 brokers.

Direct connect is only supported by the WMQ Event Broker v2.1, and the v5 brokers.

Multicast is only supported by the v5 brokers.

(As WMQ v5.2 fell out of service end Dec 2003, so the only supported installations of MA0C are now running on v5.3 queue managers.)

# IBM Message Service (XMS) Clients
## XMS and JMS shared features

- Provides common messaging API across IBM message servers and programming languages
  - Based on high level Message classes (JMS 1.1)
- API support for both Point-to-Point and Publish/Subscribe messaging
  - Send/Receive messages to/from queues or topics
- Decoupled Administration
  - Common Admin tooling to define Connection Factories and Destinations
  - Directory can be LDAP, COS naming, or file system based
- Synchronous/Asynchronous messaging receipt
  - Synchronous: Application waits on receive() method
  - Asynchronous: Application provides OnMessage() callback - called on background thread for each incoming message
- Message Selection
  - Received messages can be filtered with SQL92 selection criteria

# IBM Message Service (XMS) Clients

## XMS and JMS shared features

N O T E S

IBM has been asked by a number of customers for an API with the simplicity and power of JMS but in a non-JAVA environment. XMS is the response to those requests. XMS is not a replacement for the MQI nor is it even the preferred API. It is merely a new API which may be preferable for some customers in some environments.

The aim is to provide an API as close as possible to JMS subject to the rules and restrictions of the non-JAVA languages. This means that the exact spelling of some of the APIs will change from their JMS equivalents but the key features of JMS, administered objects, sessions, selectors, asynchronous consumers, are all present.

## IBM Message Service (XMS) Clients
## XMS specific features

- XMS API will be updated in lock step with JMS

- (Almost) Full message compatibility with IBM JMS
  - Both WMQ JMS and WPM JMS
  - (Object messages treated as binary data in C/C++)

- API initially provided for C, C++, and C#
  - Possibilities for COBOL, PL/1, RPG(?) etc.

- Beta SupportPac IA94 available
  - Subset of final API
  - C / C++ Interface
  - Servers
    - WebSphere MQ Queue Managers (Point to Point only)
    - WBI Broker Real-Time transport
    - WAS v6 Platform Messaging

# IBM Message Service (XMS) Clients
## XMS specific features

N O T E S

Clearly basing XMS on JMS and JMS concepts is trying to hit a moving target. However, it's a target which is moving fairly slowly. There have only been two major revisions of the JMS specification, 1.02 and the new, unified domain 1.1. It was felt that there wasn't sufficient need to support the 1.02 specification since the 1.1 specification is preferable and already well established. As future versions of the JMS specification are released the XMS API will be updated accordingly.

There will always be certain certain areas where 100% compatability is not possible. One of these is the notion of 'object' messages in JMS. Object messages are essentially serialized Java objects. Clearly these make little sense in a non-Java world. If XMS is sent an object message it will be treated as just binary data.

The intention is to support further language bindings over time - subject to user requests and requirements. The current set of C, C++ and C#  seem the obvious choices but clearly other languages such as COBOL would benefit from XMS.

The XMS Support is currently available as a Beta SupportPac essentially as a technology review.  In the near furture the intention is to make it a full supported, product extension, SupportPac.

## When to use XMS

- To use non-Java applications on any transport/servers
- To re-use JMS administered objects and programming skills
  - in a .NET or other non-Java environment
- To simplify Pub/Sub applications or decouple administration

- .......and when <u>not</u> to use XMS....

  - When all else is WMQI
    - Use the MQI to get widest range of languages & platforms
    - Use the MQI when performance is critical
  - If using MQe / SCADA on a very small footprint device

## When to use XMS

N O T E S

XMS will clearly be of most use to customers who have a significant investment in JMS applications and skills.  However, the JMS features such as asynchronous consumers and selectors may make it attractive to newcomers to messaging. The nature of JMS is to be provider independant and this is carried forward into XMS. The consequence of this is that not necessarily all the features of the backend server are exposed in the API. For example, JMS has not concept of message grouping/segmentation as WMQI has.

The plan for XMS is that the same client application can be used to connect to either WBI Broker, WPM or WMQI. The choice as to which server to connect to and 'how' can be hidden from the application in the administered object definition.

# XMS C# Publish Example

```
//  Create a connection factory and set properties for connection
xms.ConnectionFactory cf = new xms.ConnectionFactory();
cf.SetIntProperty( xms.XMSC.TRANSPORT_TYPE, xms.XMSC.TP_DIRECT_TCPIP );
cf.SetStringProperty( xms.XMSC.HOST_NAME, "localhost" );
cf.SetIntProperty( xms.XMSC.PORT, 1506 );

// Create Real-time Connection, Session and Topic
xms.IConnection conn= cf.CreateConnection();
xms.ISession sess = conn.CreateSession(false,  xms.AcknowledgeMode.AUTO_ACKNOWLEDGE);
xms.IDestination topic = new xms.Destination("topic://welcome");

// Create Producer and BytesMessage, then send it
xms.IMessageProducer pub = sess.CreateProducer( topic );
xms.IBytesMessage msg = sess.CreateBytesMessage();
msg.WriteUTF("Hello from XMS C#");
pub.Send(msg);
Console.WriteLine("Messages sent.");

//  Clean up
pub.Close();
sess.Close();
conn.Close();
```

## XMS C# Publish Example

N O T E S

Clearly languages such as C++ and C#, being object orientated, are a much closer fit to JMS than procedural languages such as 'C'.  Essentially the 'C' interface uses the notion of 'handles' which are returned and passed back on subsequent calls (the same way as done in the MQI).

The XMS C# implementation is a fully managed client unless the client connection needs to be made of SSL. Currently Microsoft do not provide a fully managed SSL library.

## XMS C++ Publish Example

```
// Create ConnectionFactory and set up properties
xms::ConnectionFactory cf;
cf.setIntProperty( XMSC_TRANSPORT_TYPE, XMSC_TP_DIRECT_TCPIP);
cf.setIntProperty( XMSC_PORT, 1506);
cf.setStringProperty( XMSC_HOST_NAME, "localhost");

// Create Real-time Connection, Session and Topic
xms::Connection conn = cf.createConnection();
xms::Session sess = conn.createSession(xmsFALSE, XMS_AUTO_ACKNOWLEDGE);
xms::Destination topic = sess.createTopic(“welcome");

// Create Producer and BytesMessage, then send it
xms::MessageProducer pub = sess.createProducer(topic);
xms::BytesMessage msg = sess.createBytesMessage();
msg.writeUTF("Hello from XMS C++");
pub.send(msg);
std::cout << "Message sent."  << std::endl;

// Clean up
pub.close();
sess.close();
conn.close();
```

## XMS C Publish Example

```
xmsRC rc = XMS_OK;                                 xmsHErrorBlock xmsError;
xmsHConnFact hCf = XMS_NULL_HANDLE;                xmsHConn hConn= XMS_NULL_HANDLE;
xmsHSess hSess = XMS_NULL_HANDLE;                  xmsHDest hTopic = XMS_NULL_HANDLE;
xmsHMsgProducer hPub = XMS_NULL_HANDLE;   xmsHMsg hMsg= XMS_NULL_HANDLE;

/* Create Error block and ConnectionFactory */
rc = xmsErrorCreate(&xmsError);
rc = xmsConnFactCreate(&hCf,xmsError);
rc = xmsSetIntProperty((xmsHObj)hCf, XMSC_TRANSPORT_TYPE, XMSC_TP_DIRECT_TCPIP, xmsError
rc = xmsSetStringProperty((xmsHObj)hCf, XMSC_HOST_NAME, "localhost", 9, xmsError);
rc = xmsSetIntProperty((xmsHObj)hCf,XMSC_PORT, 1506, xmsError);

/* Create Real-time Connection, Session and Topic  */
rc = xmsConnFactCreateConnection(hCf, &hConn, xmsError);
rc = xmsConnCreateSession(hConn, xmsFALSE, XMS_AUTO_ACKNOWLEDGE, &hSess, xmsError);
rc = xmsDestCreateByType(XMSC_TOPIC, “welcome", &hTopic, xmsError);

/* Create Producer and BytesMessage, then send it */
rc = xmsSessCreateProducer(hSess, hTopic, &hPub,xmsError);
rc = xmsSessCreateBytesMessage(hSess, &hMsg,xmsError);
rc = xmsBytesMsgWriteUTF(hMsg, "Hello from XMS C", strlen("Hello from XMS C"), xmsError);
rc = xmsMsgProducerSend(hPub, hMsg, xmsError);
```

## Some Resources

- **Book : Java Message Service**
  - Richard Monson-Haefel & David Chappel
  - Publisher: O’Reilly

- **Book : Enterprise Messaging using JMS and IBM WebSphere**
  - Kareem Yusuf Ph.D.
  - Publisher: Prentice Hall 2004

- **Support Pacs**
  - http://www-306.ibm.com/software/integration/support/supportpacs/
  - MA0C - WMQ Pub/Sub Support
  - MA88 - WMQ MQI Java/JMS Support
  - IA94 - XMS Beta

- **Redbook : WebSphere MQ Publish/Subscribe Applications (SG24-6282)**
- **WebSphere MQ Using Java (V5.3.0.6 Revision)**

## Summary

- JMS is a vendor-independent messaging API
  - Point-to-Point and Publish/Subscribe
  - No Vendor interoperability

- WebSphere MQ implements JMS
  - Interoperability with non JMS applications
  - Publish/Subscribe support via MA0C and others.

- XMS Api
  - Very JMS like
  - Interoperates with WMQ and WPM JMS
  - Support for C,C++,C#
  - Connect to WMQ, WPM and WBI-Broker