

Der Java Message Service (JMS)

von

Robert Switzer

1 Präambel

Die JMS-Spezifikation ist recht lang (140 Seiten) und komplex. So werde ich in diesem Vortrag unmöglich alle Feinheiten von JMS besprechen können. Ich versuche nur, die Highlights hervorzuheben.

Ein Modell für die Entwicklung verteilter Systeme, das sehr erfolgreich wurde und auch heute sehr beliebt ist, war (und ist) die Message Oriented Middleware (MOM). Wie der Name nahelegt, basiert MOM auf dem (asynchronen) Austausch von Nachrichten.

MOM hat Vorteile gegenüber “klassische” RPC-basierte verteilte Systeme, wenn folgende Merkmale vorliegen:

- die Komponenten sollten *lose gekoppelt* sein. D. h. keine Komponente besitzt eine Referenz für eine andere. Die Kommunikation basiert auf standardisierten oder zumindest gemeinsam vereinbarten Nachrichtenformate anstatt auf stark typisierten Schnittstellen.
- die Komponenten sind nicht nur verteilt sondern ihre Verfügbarkeit ist nicht immer garantiert – z. B. wenn einige Komponenten auf Laptops laufen.
- die Kommunikation zwischen den Komponenten kann in einem asynchronen Modus ablaufen. Dies bedeutet, Komponente *A* kann eine Nachricht an Komponente *B* schicken und eine Antwort zu einer Zeit erhalten, die evtl. viel später liegt – oder sogar gar keine Antwort.

MOM basiert auf einer Softwareschicht, die den Nachrichtenaustausch regelt. Davon gibt es mehrere konkurrierenden Angebote; hier einige Beispiele:

- das MSMQ von Microsoft.
- das MQSeries (heute Websphere MQ) von IBM.
- im Grunde ist der CORBA Notification Service von der OMG auch ein Beitrag zu dieser Kultur.

Diese Systeme haben alle recht unterschiedliche (und nicht kompatible) Schnittstellen. Die Autoren von JMS haben sich die Aufgabe gestellt, dem Java-Programmierer eine einheitliche und hoffentlich leicht verständliche Schnittstelle zu all diesen Systemen zu bieten.

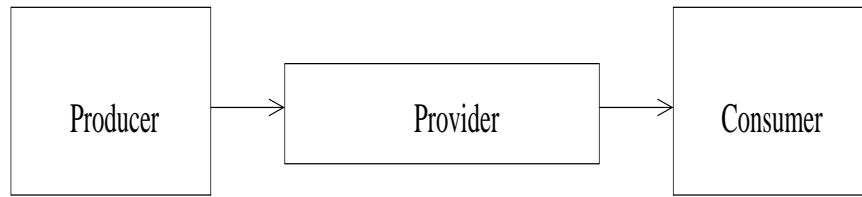


Abbildung 1: JMS-Clients und -Provider

Die JMS-Autoren haben insofern eine schwierige Aufgabe gehabt, als sie die Bühne erst betraten, als die Systeme, denen Sie eine einheitliche Schnittstelle verpassen wollten, sich ziemlich weit auseinander etwickelt hatten. Das erklärt die Komplexität der JMS-Spezifikation.

Dennoch ist der Gebrauch von JMS erstaunlich einfach, wenn man Anwendungen ohne allzu hohe Anforderungen entwickeln will. Erst fortgeschrittene Entwickler brauchen die Feinheiten von JMS. Denen ist allerdings die sorgfältige Lektüre von [jms] dringend zu empfehlen.

2 Clients und Provider

Alle Nutzer eines Nachrichtentransportservice werden als Clients betrachtet. Wenn es in dieser Konfiguration so etwas wie einen Server gibt, dann ist das der Nachrichtentransportservice, der bei JMS als *Provider* bezeichnet wird; das ist die JMS-Implementation.

Beispiel. Wenn das MSMQ von Microsoft eingesetzt wird, um die Nachrichten zu transportieren, dann ist der Provider eine Softwareschicht, die zwischen den Clients und MSMQ liegt. Auf der einen Seite bietet der Provider den Clients die in JMS spezifizierte Schnittstelle und auf der anderen Seite benutzt der Provider die proprietäre Schnittstelle von MSMQ, um JMS zu implementieren.

Abbildung 1 illustriert den Provider mit seinen Clients, von denen es mindestens zwei gibt: den *Producer*, der die Nachrichten generiert und abschickt, und den *Consumer*, der die Nachrichten empfängt und auf seine charakteristische Art verarbeitet.

3 Endpunkte alias Ziele

JMS arbeitet mit der Abstraktion *Endpunkt* (engl. endpoint) oder *Ziel* (engl. destination). Ein Producer schickt seine Nachrichten nicht an einen expliziten Consumer; das wäre eine zu *enge Kopplung*. Stattdessen schickt ein Producer seine Nachrichten an einen Endpunkt.

Umgekehrt empfängt ein Consumer seine Nachrichten nicht von einem expliziten Producer – aus demselben Grund. Stattdessen legt ein Consumer fest, von welchem Endpunkt er seine Nachrichten beziehen will.

So ist es möglich, daß mehrere Consumer Nachrichten vom selben Producer empfangen: sie müßten nur alle denselben Endpunkt als Quelle ihrer Nachrichten angeben. Davon braucht der Producer nichts zu wissen.

Genauso könnte jeder Consumer Nachrichten von mehreren Producern empfangen. Die verschiedenen Producer müßten nur alle denselben Endpunkt als Ziel ihrer Nachrichten angeben.

4 Nachrichtenmodelle

Wenn man die Möglichkeiten genauer betrachtet, die am Ende von Abschnitt 3 beschrieben wurden, erkennt man, daß sie insofern problematisch sind, als die Semantik noch unklar ist.

Beispiel. Nehmen wir das Beispiel, wo mehrere Consumer ihre Nachrichten vom selben Endpunkt beziehen. Heißt das, daß jeder Consumer nur einen Teil der vom Producer generierten Nachrichten erhält? Oder muß der Provider diese Situation erkennen und die Nachrichten duplizieren, damit jeder Consumer von jeder Nachricht eine Kopie erhält?

Es gibt in der Tat einige Provider, die die spezielle Semantik nicht unterstützen, die im letzten Satz unseres Beispiels beschrieben wird. Aus diesem Grund teilt JMS die Provider in zwei Bereiche (engl. domain) ein, die jeweils verschiedene *Nachrichtenmodelle* unterstützen:

Point-to-Point (kurz PTP) Hier heißt der Endpunkt *Queue* und hat die Semantik, die dieser Name nahelegt: hat ein Consumer die “nächste” Nachricht vom Endpunkt bekommen, ist diese weg. Ein anderer Consumer (wenn es denn einen anderen gibt) bekommt die Nachricht danach. Es ist nicht ausdrücklich verboten, daß mehrere Consumer ihre Nachrichten von derselben Queue beziehen – aber JMS spezifiziert die Semantik in diesem Fall nicht.

Publish-and-Subscribe (kurz Pub/Sub) Hier heißt der Endpunkt *Topic*; in der Regel *publizieren* (engl. publish) beliebig viele Producer ihre Nachrichten zu einem Topic und beliebig viele Consumer *abonnieren* (engl. subscribe) die Nachrichten von diesem Topic. Dabei wird garantiert, daß jeder Consumer eine Kopie von jeder Nachricht erhält, die an diesen Topic geschickt wurde. Hierbei gibt es gewisse Einschränkungen der Phrase “von jeder Nachricht”, die von [jms] ausdrücklich genannt werden.

JMS 1.0 hat diese beiden Bereiche (bzw. Nachrichtenmodelle) durch getrennte Interfacefamilien unterstützt. Das bedeutete, daß sich der Entwickler eines Client für den einen oder anderen Bereich entscheiden mußte; es war nicht möglich, Clients zu programmieren, die in beiden Bereichen funktionierten.

Ab JMS 1.1 haben die JMS-Autoren beschlossen, die beiden Interfacefamilien zu vereinheitlichen, wie wir in Abschnitt 5 sehen werden.

5 Die Architektur von JMS

5.1 JMS-Anwendungen

Eine JMS-Anwendung hat folgende Bestandteile:

- JMS-Clients – diese sind die Java-Programme, die Nachrichten versenden und empfangen.
- Nicht-JMS-Clients – diese sind die Clients, die die eigene Schnittstelle eines Nachrichtentransportsystems einsetzen und auf die JMS Schnittstellen verzichten. In der Regel stammen diese Clients aus der Zeit vor JMS (sog. *legacy* Clients).
- Nachrichten (engl. *message*) – jede Anwendung definiert eine Menge von Nachrichten, die benutzt werden, um Informationen zwischen Clients zu transportieren.
- JMS-Provider – dies ist das Nachrichtentransportsystem, das JMS implementiert und die Funktionalität bereitstellt, damit ein Administrator die Anwendung konfigurieren kann.
- administrierte Objekte (engl. *administered objects*) – diese Objekte werden vom Administrator erzeugt und vorkonfiguriert.

5.2 Die Verwaltung

Man muß davon ausgehen, daß JMS-Providers erheblich voneinander unterscheiden sowohl in der von ihnen eingesetzten Technologie als auch in der Art, wie sie installiert und verwaltet werden.

Damit JMS-Clients portabel hinsichtlich des eingesetzten Provider sein können, müssen sie von diesen proprietären Eigenschaften isoliert werden. JMS erreicht dies über die administrierten Objekte, die vom Administrator erzeugt und konfiguriert werden, damit sie später von Clients eingesetzt werden können.

Es gibt zwei Sorten von administrierten Objekten:

- **ConnectionFactory** – dies sind die Objekte, die ein Client verwendet, um eine Verbindung zum Provider aufzunehmen.
- **Destination** – dies sind die Endpunkte, die Producer und Consumer gemeinsam benutzen, um Nachrichten auszutauschen.

Administrierte Objekte werden im JNDI-Baum untergebracht, von wo jeder Client sie auf portable Art holen kann.

5.3 Die Interfaces

Wie wir seit Abschnitt 4 wissen, bot JMS 1.0 zwei separate Interfacefamilien zur Unterstützung der beiden Bereiche. Seit JMS 1.1 gibt es eine gemeinsame Familie `common` von deren Interfaces die Interfaces der älteren Interfacefamilien erben. Clients, die nur die Interfaces von `common` einsetzen, sind unabhängig vom Bereich des jeweils eingesetzten Provider – zumindest *syntaktisch*. Die *Semantik* wird natürlich je nach Modell anders sein.

Folgende Tabelle zeigt die Beziehungen zwischen den verschiedenen Interfaces.

gemeinsam	PTP-spezifisch	Pub/Sub-spezifisch
<code>ConnectionFactory</code>	<code>QueueConnectionFactory</code>	<code>TopicConnectionFactory</code>
<code>Connection</code>	<code>QueueConnection</code>	<code>TopicConnection</code>
<code>Destination</code>	<code>Queue</code>	<code>Topic</code>
<code>Session</code>	<code>QueueSession</code>	<code>TopicSession</code>
<code>MessageProducer</code>	<code>QueueSender</code>	<code>TopicPublisher</code>
<code>MessageConsumer</code>	<code>QueueReceiver</code>	<code>TopicSubscriber</code>

Die Bedeutung der Interfaces im Einzelnen ist wie folgt:

- `ConnectionFactory` – ein administriertes Objekt, mit dem ein Client eine Verbindung zum Provider herstellen kann.
- `Connection` – eine aktive Verbindung zu einem Provider.
- `Destination` – ein administriertes Objekt, das die Identität eines Endpunkts kapselt.
- `Session` – ein single-threaded Kontext für das Versenden und Empfangen von Nachrichten.
- `MessageProducer` – ein Objekt, das von einer `Session` erzeugt wird und mit dem man Nachrichten an einen vorbestimmten Endpunkt versenden kann.
- `MessageConsumer` – ein Objekt, das von einer `Session` erzeugt wird und mit dem man Nachrichten von einem vorbestimmten Endpunkt empfangen kann.

Abbildung 2 illustriert die Beziehungen zwischen den JMS-Objekten.

Der Ausdruck *consume* wird in [jms] als generische Bezeichnung für den Akt des Empfangs einer Nachricht durch einen Client verwendet. D. h. der Provider hat eine Nachricht erhalten und gibt diese weiter an den Client. Da JMS sowohl den synchronen als auch den asynchronen Empfang von Nachrichten unterstützt, wird der Ausdruck 'consume' da benutzt, wo man zwischen diesen beiden Möglichkeiten nicht unterscheiden muß oder will.

Genauso wird der Ausdruck *produce* als generische Bezeichnung für den Akt des Versendens einer Nachricht verwendet.

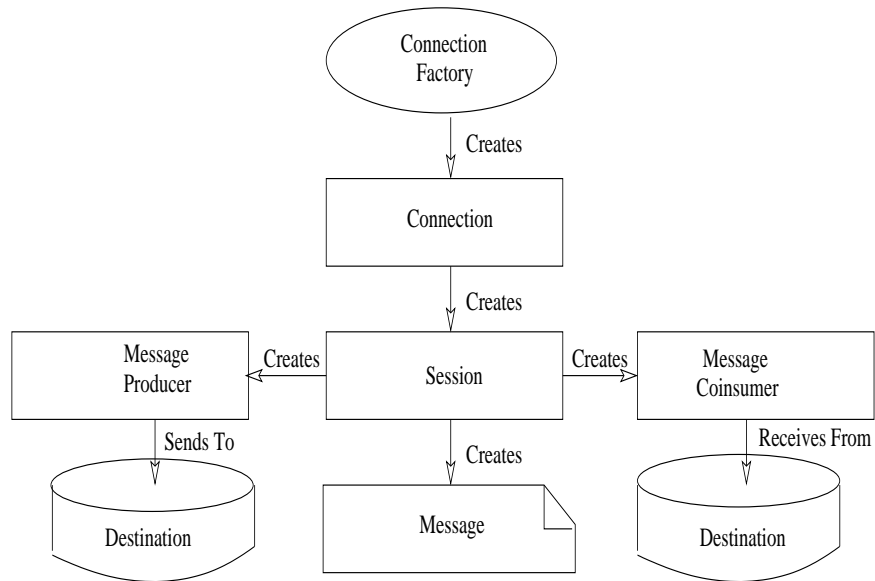


Abbildung 2: Beziehungen zwischen JMS-Objekten

6 Nachrichten

Bis jetzt haben wir noch gar nicht über die Hauptsache gesprochen: die Nachrichten, die die Clients austauschen. Anders als die Interfaces, die wir bisher besprochen haben, waren das Interface `Message` und dessen Subtypen schon immer in beiden Bereichen gleich. Der Supertyp `Message` hat fünf Subtypen:

```

TextMessage
MapMessage
ObjectMessage
StreamMessage
ByteMessage
  
```

von denen ich nur die ersten beiden näher beschreiben will.

`TextMessage` – dies ist die einfachste Art von Nachricht; sie transportiert einen `String`. Dementsprechend hat dieser Typ Methoden `setText` und `getText`, mit denen man der Nachricht ihren Inhalt geben bzw. diese wieder extrahieren kann.

Dieser Typ entfaltet seine volle Nützlichkeit, wenn der transportierte Text ein XML-Dokument ist.

`MapMessage` – mit Nachrichten dieser Art kann man ganz bequem Listen von name/value-Paaren transportieren. Diesen Typ verwendet man gern, wenn man eine Liste von Aktualargumenten transportieren will. Die Werte der transportierten name/value-Paare müssen Basistypen sein:

```

boolean
  
```

```
byte
char
double
float
int
long
short
```

sowie `byte[]` oder `String`. Für jeden dieser Typen gibt es entsprechende `get/set`-Methoden. Dieser Typ nimmt gewisse automatische Typumwandlungen vor. Hierzu s. [jms].

6.1 Nachrichten-Header

Genauso, wie EMail- und HTTP-Nachrichten Header haben können, die für den Transport und die Verarbeitung dieser Nachrichten wichtig sein können, so besitzen auch JMS-Nachrichten Header.

JMS-Nachrichten können eine ganze Reihe von diversen Header-Einträgen besitzen, von denen ich auch hier nur einige herauspicken will:

`JMSMessageID` – dieser Eintrag identifiziert die Nachricht eindeutig. Dieser Eintrag wird nicht vom Clientprogrammierer festgelegt (weder direkt noch indirekt). Vielmehr ist der korrekte Wert eingetragen, wenn `send` zurückkehrt.

`JMSCorrelationID` – oft will man eine Nachricht mit einer anderen korrelieren – etwa Antwort mit Auftrag. Hierzu kann man diesen Eintrag verwenden.

Der Wert dieses Eintrags kann einer der folgenden sein:

- ein providerspezifischer Nachrichtenidentifikator.
- ein anwendungsspezifischer String.
- ein providernativer Wert vom Typ `byte[]`.

Die naheliegendste Möglichkeit ist die erste.

Man kann den Wert dieses Eintrags mit einer der folgenden Methoden der Klasse `Message` festlegen:

```
void setJMSCorrelationID(String value)
void setJMSCorrelationIDAsBytes(byte[] value)
```

`JMSReplyTo` – der Absender kann hier eine Referenz auf einen Endpunkt (Typ `Destination`) eintragen, zu dem der Empfänger eine Antwort schicken sollte.

Dies erledigt er, indem er folgende Methode der Klasse `Message` verwendet:

```
void setJMSReplyTo(Destination replyTo)
```

JMSExpiration – oft versendet man Nachrichten, deren Aktualität begrenzt ist. Man will nicht, daß solche Nachrichten erst dann zugestellt werden, wenn sie schon Schnee von gestern sind. Mit diesem Header-Eintrag kann man genau das verhindern. Der Inhalt dieses Eintrags signalisiert dem Provider, daß diese Nachricht zu einer bestimmten Zeit vernichtet und nicht mehr zugestellt werden soll.

Man kann den Wert dieses Eintrags mit folgender Methode des Interface **Message** setzen:

```
void setJMSExpiration(long expiration)
```

Man beachte, daß der Wert, den man hier als Argument angibt, nicht etwa die Zeit bis zum Ablauf der Gültigkeit ist, sondern die Zeit in Millisekunden, die sich als Summe aus der Greenwichzeit (GMT) plus die Lebensspanne der Nachricht ergibt. [jms] legt dem Provider-Hersteller nahe, diese Vorgabe so genau wie möglich zu erfüllen – gibt aber keine einzuhaltende Genauigkeit vor.

Alternativ kann man die defaultmäßige Überlebenszeit von Nachrichten mit folgender Methode der Klasse **MessageProducer** festlegen:

```
void setTimeToLive(long timeToLive)
```

In diesem Fall ist das Argument tatsächlich eine Zeitspanne. Ist dieser Wert 0, leben Nachrichten unbegrenzt.

JMSPriority – frei nach George Orwell: alle Nachrichten sind gleich aber manche sind gleicher. Sollte eine Nachricht besonders dringlich zugestellt werden, kann man ihr eine höhere Prioritätsstufe zuordnen. [jms] sieht zehn Prioritätsstufen von 0 bis 9 vor, wobei die Stufen 0–4 als Schattierungen der Stufe *normal* und die Stufen 5–9 als Schattierungen von *expedited* (d. h. eilig) angesehen werden.

Man kann den Wert dieses Eintrags mit folgender Methode der Klasse **Message** festlegen:

```
void setJMSPriority(int priority)
```

Es gibt eine Alternative: die Methode **send** der Klasse **MessageProducer** kommt in mehreren überladenen Varianten vor, wobei einige die Priorität als Argument besitzen.

6.2 Nachrichten-Attribute (Properties)

Eine besondere Stärke von JMS ist, daß ein Empfänger mit einem *Filter* (cf. Abschnitt 8) angeben kann, welche Nachrichten er bekommen will.

Die Attribute (engl. property) sind der Schlüssel zu dieser Filterfunktion. Attribute sind Paare (Name, Wert), die der Absender mit der Nachricht verknüpfen kann, wobei die Typen der Werte nur Basistypen oder **String** sein dürfen.

Die Namen gibt es in drei Sorten:

- von JMS vordefinierte Attributnamen; diese beginnen stets mit dem Präfix **JMSX**. Das 'X' in diesem Präfix dient zur Unterscheidung der Attributnamen von den Headernamen.
- vom Provider vordefinierte Attributenamen; diese beginnen stets mit dem Präfix **JMS_<vendor_name>**, wobei <vendor_name> der Name des ProviderHerstellers ist.
- anwendungsspezifische Attributnamen; diese müssen den Regeln für Java-Identifizierer genügen und dürfen nicht eines der reservierten Worte der Sprache SQL92 sein. Bei Attributnamen ist die Groß/Kleinschreibung signifikant.

Für jeden der möglichen Attributtypen T gibt es entsprechende **get**- und **set**-Methoden

```
T get<T>Property(String name)
void set<T>Property(String name, T value)
```

Dabei muß der mit der **get**-Methode geholte Wert nicht immer denselben Typ haben als der mit der **set**-Methode gesetzte Wert; JMS nimmt gewisse Typkonversionen vor (z. B. float zu double). Hierzu konsultiere man [jms].

Beispiel.

```
setFloatProperty("price", 1.89);
double p = getDoubleProperty("price"); // OK
```

Die Bedeutung der meisten von JMS vordefinierten Attribute wird in folgender Tabelle erläutert:

Name	Type	Set By	Use
JMSXUserID	String	Provider on send	Identity of user sending message
JMSXAppID	String	Provider on send	Identity of application sending message
JMSXGroupID	String	Client	Identity of message group this message is part of
JMSXGroupSeq	int	Client	Sequence number of message within group
JMSXProducerTXID	String	Provider on send	Identifier of transaction message was produced in
JMSXConsumerTXID	String	Provider on receive	Identifier of transaction message was consumed in

Kommentar:

1. Die Methode `createConnection` der Klasse `ConnectionFactory` kommt in zwei überladenen Versionen, wobei die eine Benutzername und Paßwort als Argumente erwartet. Welche Version der Clientprogrammierer verwenden muß, hängt davon ab, wie der Administrator das `ConnectionFactory`-Objekt konfiguriert hat. Nur wenn beim Aufbau der Verbindung zum Provider diese Benutzerdaten angegeben wurden, wird das Attribut `JMSXUserID` einen sinnvollen Wert besitzen.
2. Bei `JMSXGroupSeq` beginnt die Zählung bei 1, nicht bei 0.

7 Die Entwicklung eines Client

Bei der Entwicklung eines Client (ob Absender oder Empfänger) sind generell folgende Schritte durchzuführen:

- man benutze JNDI `lookup`, um ein passendes Objekt vom Typ `ConnectionFactory` zu finden.
- man benutze JNDI `lookup`, um einen oder mehrere Endpunkte (Typ `Destination`) zu finden.
- man benutze das `ConnectionFactory`-Objekt, um eine `Connection` zu erzeugen, bei der das Senden zunächst blockiert ist.
- man benutze die `Connection`, um mindestens ein `Session`-Objekt zu erzeugen.
- man benutze die `Session` sowie die `Destination`, um einen `MessageProducer` oder einen `MessageConsumer` zu erzeugen.
- man benutze das `Session`-Objekt, um eine `Message` zu erzeugen. Dann fülle man diese mit Inhalt und übergebe sie an den `MessageProducer` (Methode `send`).
Dieser Schritt ist selbstverständlich nur im Absender sinnvoll.
- man instruiere die `Connection`, das Versenden von Nachrichten zu beginnen (Methode `start`).
Dies macht man sowohl im Absender als auch im Empfänger.
- man hole die nächste Nachricht vom `MessageConsumer` (Methode `receive`) und verarbeite sie auf die für diesen Client charakteristische Art.
Dieser Schritt ist selbstverständlich nur im Empfänger sinnvoll.
- Da Objekte der Typen `Connection` und `Session` relativ kostspielige Ressourcen sind, gibt man diese wieder frei (Methode `close`), wenn man sie nicht mehr benötigt.

Bemerkung. Diese Aufzählung geht davon aus, daß der Administrator administrierte Objekte der Typen `ConnectionFactory` und `Destination` erzeugt, konfiguriert und in den JNDI-Baum gehängt hat.

7.1 Beispiel eines Absenders

In diesem Abschnitt zeigen wir Beispielcode für einen Absender, der aktuelle Daten über Aktien an einen Endpunkt vom Typ `Topic` liefert.

Wir beginnen mit einer gemeinsamen Oberklasse für Absender und Empfänger; diese Klasse erledigt alle Schritte, die Absender und Empfänger beide ausführen müssen:

```
package stocks;

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

public class StockBase
{
    protected Destination stockTicker = null;
    protected Connection stockConnection = null;
    protected Session stockSession = null;

    protected void init() throws Exception
    {
        /*
         * Get root of JNDI tree.
         */

        Context initial = new InitialContext();

        /*
         * Lookup connection factory.
         */

        ConnectionFactory cFactory = (ConnectionFactory)
            initial.lookup("ConnectionFactory");

        /*
         * Lookup destination.
         */
        stockTicker = (Destination) initial.lookup("jms/stockTicker");

        /*
         * Create a connection to provider.
         */

        stockConnection = cFactory.createConnection();

        /*
```

```

        * Create a session -- not transacted; uses AUTO_ACKNOWLEDGE
        * for message acknowledgement.
        */

stockSession = stockConnection.createSession(false,
                                             Session.AUTO_ACKNOWLEDGE);

/*
 * And open the valve.
 */

stockConnection.start();
}

protected void cleanUp(MessageProducer sender,
                       MessageConsumer receiver)
{
    try
    {
        if (sender != null) sender.close();
        if (receiver != null) receiver.close();
        if (stockSession != null) stockSession.close();
        if (stockConnection != null) stockConnection.close();
    } catch (Exception e) {}
}
}

```

Und nun der eigentliche Absender:

```

package stocks;

import java.io.*;
import java.math.BigDecimal;
import java.util.*;
import javax.jms.*;

public class StockReporter extends StockBase
{
    public StockReporter()
    {
        MessageProducer sender = null;

        try
        {
            init();

```

```

    /*
    * Create a message producer.
    */

    sender = stockSession.createProducer(stockTicker);

    /*
    * Create a MapMessage.
    */

    MapMessage stockData = stockSession.createMapMessage();

    while (fetchInfo(stockData))
    {
        /*
        * Send message on its way.
        */

        sender.send(stockData);

        /*
        * And clear out stockData in
        * preparation for next iteration.
        */

        stockData.clearBody();
        stockData.clearProperties();
    }
}
catch (Exception e)
{
    System.err.println(e);
}
finally
{
    cleanUp(sender, null);
}
}

/**
 * Method fetchInfo fills a message with data
 * and sets appropriate properties.
 * @param stockData the message to be filled in.
 * @return true if stockData could be filled with
 * new data; otherwise false.

```

```

    */
private boolean fetchInfo(MapMessage stockData)
{
    boolean result = false;
    /*
     * Here we could read data from a database,
     * Instead we use a property file.
     * Properties duplicate information in body
     * so consumer can filter.
     */
    if (priceList == null)
        loadPriceList();
    if (!priceList.hasMoreElements())
        return result;

    try
    {

        String stockSymbol = (String) priceList.nextElement();
        stockData.setString("stockSymbol", stockSymbol);
        stockData.setStringProperty("stockSymbol", stockSymbol);
        BigDecimal price = new BigDecimal(props.getProperty(stockSymbol));
        stockData.setDouble("price", price.doubleValue());
        stockData.setDoubleProperty("price", price.doubleValue());
        result = true;
    }
    catch (Exception e)
    {
        System.err.println(e);
    }
    return result;
}

private Enumeration priceList = null;
private Properties props = null;

private void loadPriceList()
{
    props = new Properties();
    try
    {
        FileInputStream in = new FileInputStream("stock.properties");
        props.load(in);
    }
    catch (Exception e)

```

```

        {
            System.err.println(e);
        }
        priceList = props.propertyNames();
    }

    public static void main(String[] args)
    {
        StockReporter reporter = new StockReporter();
        // That's enough; the constructor does it all.
    }
}

```

Kommentar:

1. Nichts in diesem Code verrät, daß wir es hier mit einem Topic zu tun haben und nicht etwa mit einer Queue. Der Administrator nimmt uns diese Entscheidung ab. Hier sehen wir die Vorteile der Common-Schnittstelle in Kombination mit den administrierten Objekten.
2. `Session`- und `Connection`-Objekte konsumieren beträchtliche Ressourcen im Provider. Darum sollte man diese stets in einer `finally`-Klausel freigeben (`close`), wenn man sie nicht weiter benötigt. Dasselbe gilt für

```

    MessageProducer
    MessageConsumer

```

3. I. Allgem. ist es keine besonderes gut Idee, Properties mit einem Objekt zu verknüpfen, die Inhalte des Objekts duplizieren. Hier haben wir die Ausnahme, die die Regel bestätigt: beim Filtern wird der Body (Inhalt, Nutzlast) der Nachricht nicht inspiziert. Es werden lediglich die Attribute herangezogen; darum müssen diese alle Informationen transportieren, die ein Empfänger evtl. zum Filtern einsetzen möchte.

7.2 Beispiel eines Empfängers

In diesem Abschnitt zeigen wir Beispielcode für einen Empfänger, der aktuelle Daten über Aktien von einem Endpunkt vom Typ `Topic` holt. Dieser Empfänger holt die Nachrichten synchron.

```

package stocks;

import java.math.BigDecimal;
import javax.jms.*;

public class StockReaderSync extends StockBase
{

```

```

public StockReaderSync()
{
    long timeout = 5000;
    MessageConsumer receiver = null;

    try
    {
        init();

        /*
         * Create a message consumer.
         */

        receiver = stockSession.createConsumer(stockTicker);

        /*
         * Fetch messages from provider synchronously.
         */

        while (true)
        {
            Message m = receiver.receive(timeout);
            if (m == null)
            {
                // We timed out.
                continue;
            }
            else
            {
                if (m instanceof MapMessage)
                {
                    MapMessage stockData = (MapMessage) m;
                    handleMessage(stockData);
                }
                else
                {
                    // Just ignore it.
                }
            }
        }
    }
    catch (Exception e)
    {

```

```

        System.err.println(e);
    }
    finally
    {
        cleanUp(null, receiver);
    }
}

private void handleMessage(MapMessage message)
{
    try
    {
        BigDecimal price = new BigDecimal(message.getDouble("price"));
        price = price.setScale(2, BigDecimal.ROUND_HALF_UP);
        System.out.println(message.getString("stockSymbol") +
            " costs $" + price);
    }
    catch (Exception e)
    {
        System.err.println(e);
    }
}

public static void main(String[] args)
{
    StockReaderSync reader = new StockReaderSync();
    // That's enough: the constructor does it all.
}
}

```

Kommentar:

1. Auch hier verrät nichts in unserem Code, daß wir es mit einem Topic zu tun haben.
2. Unser Konstruktor enthält eine Endlosschleife; ohne diese würde unser Empfänger terminieren, ohne eine einzige Nachricht zu empfangen. Aber sie bedeutet, daß wir unseren Empfänger gewaltsam killen müssen.

8 Nachrichten filtern

Wie wir in Abschnitt 6.2 erfuhren, muß ein Empfänger nicht alle Nachrichten annehmen, die in dem von ihm gewählten Endpunkt ankommen. Er kann sie *filtern* lassen, bevor sie an ihn übergeben werden.

Hierzu verwendet er die Methode `createConsumer` der Klasse `Session` mit folgender Signatur:

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

Der in diesem Kontext besonders interessante Parameter ist der zweite `messageSelector`. Dieser String ist ein Ausdruck (engl. expression), der der Syntax der Ausdrücke in der Sprache SQL92 genügt.

Diese Ausdrücke setzen sich aus folgenden Bestandteilen zusammen:

- Identifiers, wobei in unserem Fall die Identifiers entweder Nachrichtenheadereinträge oder Nachrichtenattribute bezeichnen.
- Literals (Konstanten), wobei diese denselben Regeln unterworfen sind wie in SQL92 (insbesondere werden Stringkonstanten in einfachen Hochkommata eingeschlossen: `'This is a string'`).
- Operatoren (logische und arithmetische sowie die speziellen Operatoren, die SQL92 definiert – wie z. B. `LIKE` und `BETWEEN`).

Beispiel. Wir könnten unseren `StockReader` so modifizieren, daß der `MessageConsumer` folgendermaßen erzeugt wird:

```
String selector = new String("price BETWEEN 20.00 AND 50.00");  
  
receiver =  
    stockSession.createConsumer(stockTicker, selector);
```

Nun wird unser Empfänger nur solche Nachrichten erhalten, die Aktien betreffen, die aktuell zwischen \$20 und \$50 kosten.

9 Synchron oder Asynchron

Wenn wir Nachrichten mit `receive` anfordern, arbeiten wir *synchron*; d. h. `receive` blockiert so lange, bis eine Nachricht im Endpunkt eingetroffen ist.

Es sei denn man verwendet die Version von `receive` mit folgender Signatur:

```
Message receive(long timeout)
```

In dem Fall blockiert `receive` so lange, bis entweder eine Nachricht eingetroffen ist oder das Zeitintervall `timeout` abgelaufen ist, je nachdem was zuerst passiert. Aber auch in diesem Fall arbeiten wir immer noch synchron – nur, daß es nicht passieren kann, daß unser Empfänger für immer blockiert.

Will man wirklich asynchron arbeiten, weil der Empfänger etwas sinnvolles tun kann, bis die nächste Nachricht eintrudelt, muß man mit einem `MessageListener` arbeiten. D. h. man schreibt eine konkrete Klasse, die die Schnittstelle `MessageListener` implementiert. Diese hat nur eine einzige Methode

```
void onMessage(Message message)
```

die vom Provider aufgerufen wird, wenn eine Nachricht für den `MessageListener` eintrifft. Damit dies funktionieren kann, muß das `MessageListener`-Objekt beim Provider angemeldet werden.

Beispiel. Wir zeigen, wie wir unseren `StockReader` asynchron machen könnten. Wir gestalten unsere Klasse wie folgt:

```
package stocks;

import java.math.BigDecimal;

import javax.jms.*;

public class StockReaderAsync extends StockBase
    implements MessageListener
{
    MessageConsumer receiver = null;

    public StockReaderAsync()
    {
        try
        {
            init();

            /*
             * Create a message consumer.
             * And register MessageListener
             * with provider.
             */

            receiver =
                stockSession.createConsumer(stockTicker);
            receiver.setMessageListener(this);

            while (true)
            {
                // Busy waiting;
                // just wait until we're killed.
            }
        }
        catch (Exception e)
        {
            System.err.println(e);
        }
    }
}
```

```

        finally
        {
            cleanUp(null, receiver);
        }
    }

    /*
     * Handle next message.
     */
    public void onMessage(Message message)
    {
        try
        {
            MapMessage infoMessage = (MapMessage) message;
            BigDecimal price = new BigDecimal(infoMessage.getDouble("price"));
            price = price.setScale(2, BigDecimal.ROUND_HALF_UP);
            System.out.println(infoMessage.getString("stockSymbol") +
                               " costs $" + price);
        } catch (Exception e)
        {
            System.err.println(e);
        }
    }

    public static void main(String[] args)
    {
        StockReaderAsync reader = new StockReaderAsync();
        // That's enough; the constructor does it all.
    }
}

```

Kommentar: Ohne die Endlosschleife am Ende unserer `main`-Methode würde unser Empfänger aussteigen, bevor er die erste Nachricht empfangen hat. Wir müssen unseren Empfänger gewaltsam killen.

10 JMS und Transaktionen

Wenn man wissen will, wie sich JMS bezüglich Transaktion verhält, muß man [jms] sehr sorgfältig lesen. Diese Information ist gewissermaßen zwischen den Zeilen zu finden. Die javadoc-Dokumentation, die mit dem J2EE-Paket mitgeliefert wird, ist da ein *wenig* auskunftsfreudiger.

Wir wissen bereits, daß die Methode `createSession` der Klasse `Connection` ein erstes Argument vom Typ `boolean` hat, mit dem man festlegt, ob die `Session` transaktionsbefähigt

(engl. transacted) sein soll. Die J2EE-Dokumentation erklärt diesen Begriff folgendermaßen:

Eine Session kann als transaktionsbefähigt deklariert werden. Eine solche Session unterstützt eine einzige Folge von Transaktionen. Jede Transaktion faßt eine Folge von **send**- und **receive**-Operationen zu einer atomaren Arbeitseinheit zusammen. Effektiv organisieren solche Transaktionen die Nachrichteneingabe- und Nachrichtenausgabeströme zu Folgen von atomaren Einheiten. Wird die Transaktion mit **commit** beendet, werden die Nachrichten der Einheit des Eingabestroms alle zusammen bestätigt (**acknowledge**) und die der Einheit des Ausgabestroms alle zusammen abgeschickt. Wird die Transaktion dagegen mit **rollback** beendet, werden die Nachrichten der Ausgabe vernichtet und die der Eingabe geborgen (**recover**).

Bemerkung. Die Klasse **Session** hat eine Methode **recover**, die die J2EE-Dokumentation folgendermaßen erklärt: die Nachrichtenzustellung dieser Session wird gestoppt und mit der ältesten noch unbestätigten Nachricht neu begonnen.

Die Erklärungen zu Transaktionen in Sessions setzen sich so fort:

Der Inhalt der Eingabe- bzw. Ausgabeeinheit einer Transaktion besteht aus denjenigen Nachrichten, die während der Dauer der Transaktion konsumiert bzw. produziert wurden.

Eine Transaktion wird beendet, mit einem Aufruf einer der beiden Methoden **commit** oder **rollback** der Klasse **Session**. Danach wird eine neue Transaktion automatisch begonnen – mit der Wirkung, daß eine transaktionsbefähigte Session *alle* seiner Aufgaben innerhalb einer aktuellen Transaktion erledigt.

Der Java Transaction Service (JTS) oder ein anderer Transaktionsmonitor kann eingesetzt werden, um verteilte Transaktionen zu realisieren, damit die Transaktionen einer Session mit den Transaktionen anderer Ressourcen kombiniert werden können (Datenbanken oder andere JMS Sessions, etc.). Die Transaktionen des JTS werden mit dem Java Transaction API (JTA) gesteuert. Der Gebrauch der **commit**- oder **rollback**-Methoden einer Session in diesem Kontext ist nicht erlaubt.

Die JMS-Spezifikation schreibt Unterstützung für JTA nicht vor; sie spezifiziert allerdings, wie diese Unterstützung auszusehen hat, wenn ein Provider sie bietet.

Obwohl es möglich ist, daß ein Client verteilte Transaktionen direkt einsetzt, ist es unwahrscheinlich, daß viele Clients dies tun werden. Unterstützung für JTA in JMS zielt eher auf Hersteller, die JMS in ihre Anwendungsserver integrieren wollen.

Literatur

[jms] <http://java.sun.com/products/jms/>. Hier findet man die jeweils aktuelle JMS-Spezifikation.